Saarland University

Addressing Socio-Technical Blind Spots in Modern Software Systems

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Masudul Hasan Masud Bhuiyan

Saarbrücken, 2026

Tag des Kolloquiums: 11. Juni 2026

Dekan: Prof. Dr. Jan Reineke

Prüfungsausschuss:

Vorsitzende:	Prof. Dr. Martina Maggio
Berichterstattende:	Dr. Cristian-Alexandru Staicu
	Prof. Dr. Andreas Zeller
	Prof. Dr. Sarah Nadi
Akademischer Mitarbeiter:	Dr. Alexi Turcotte

Dedicated to the people whom the world failed

আমি-যে দেখেছি প্রতিকারহীন শক্তের অপরাধে
বিচারের বাণী নীরবে নিভৃতে কাঁদে
আমি-যে দেখিনু তরুণ বালক উন্মাদ হয়ে ছুটে
কী যন্ত্রণায় মরেছে পাথরে নিষ্ফল মাথা কুটে।

কণ্ঠ আমার রুদ্ধ আজিকে, বাঁশি সংগীতহারা,
অমাবস্যার কারা
লুপ্ত করেছে আমার ভুবন দুঃস্বপ্নের তলে,
তাই তো তোমায় শুধাই অশ্রুজলে--

যাহারা তোমার বিষাইছে বায়ু, নিভাইছে তব আলো,
তুমি কি তাদের ক্ষমা করিয়াছ, তুমি কি বেসেছ ভালো।

– রবীন্দ্রনাথ ঠাকুর

And meanwhile I see secretive hatred murdering the helpless
Under cover of night;
And Justice weeping silently and furtively at power misused,
No hope of redress.
I see young men working themselves into a frenzy,
In agony dashing their heads against stone to no avail.

My voice is choked today; I have no music in my flute:
Black moonless night
Has imprisoned my world, plunged it into nightmare. And this is why,
With tears in my eyes, I ask:
Those who have poisoned your air, those who have extinguished your light,
Can it be that you have forgiven them? Can it be that you love them?

– Rabindranath Tagore

(English translation by William Radice)

UNIVERSITÄT
DES
SAARLANDES

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, ***January/2026***

gez. / signed

Masudul Hasan Masud Bhuiyan

Zusammenfassung

Softwaresysteme haben sich von isolierten Artefakten zu großen, global verteilten Ökosystemen entwickelt. Dieser Wandel hat sozio-technische blinde Flecken offengelegt, die durch traditionelle Praktiken der Softwaretechnik und IT-Sicherheit nicht vollständig adressiert werden. Diese Dissertation argumentiert, dass die Sicherung und nachhaltige Gestaltung moderner Softwaresysteme einen sozio-technischen Ansatz erfordert, der demografische und sprachliche Vielfalt, architektonische Komplexität sowie sich wandelnde Threat Models berücksichtigt.

Zu diesem Zweck präsentieren wir zunächst eine vergleichende Studie von 200.000 Websites aus zehn Industrie- und zehn Entwicklungsländern, die zeigt, wie der wirtschaftliche Kontext Performanz-, Sicherheits-, Datenschutz- und Barrierefreiheitspraktiken im Web prägt. Zweitens untersuchen wir Barrierefreiheitsprobleme im Globalen Süden anhand der ersten länderübergreifenden Barrierefreiheitsstudie mit 100.000 Websites und zeigen dabei anhaltende Hürden für Nutzerinnen und Nutzer mit Behinderungen auf. Aufbauend auf dieser Analyse stellen wir `LangCrUX` und `Kizuki` vor, die weitverbreitete Diskrepanzen zwischen sichtbaren Inhalten und Barrierefreiheitsmetadaten auf mehrsprachigen Websites aufdecken. Aus Entwicklerperspektive präsentieren wir zudem eine zehnjährige Longitudinalstudie der GitHub-Aktivität. Diese zeigt, dass die nicht-englischsprachige Beteiligung zwar stetig zunimmt, jedoch mit einer geringeren Sichtbarkeit verbunden ist, welche die Reichweite von Projekten und die Gewinnung von Mitwirkenden begrenzt.

Ergänzend zu diesen sozioökonomischen Ergebnissen widmet sich der letzte Teil den technischen Herausforderungen, die sich aus neuartigen Schwachstellenklassen und zunehmender Systemabstraktion ergeben. Wir liefern eine Systematisierung des Wissens zu Regular Expression Denial of Service (ReDoS), identifizieren Lücken in Threat Models, Definitionen und Evaluationsmethoden und analysieren die in modernen Regex-Engines eingesetzten Abwehrmechanismen. Zur Verbesserung von Reproduzierbarkeit und Evaluation im JavaScript-Ökosystem stellen wir SecBench.js vor, eine Benchmark-Suite realer Schwachstellen und ausführbarer Exploits für das npm-Ökosystem. Abschließend präsentieren wir Graphia, einen auf Graph-Neuronalen Netzen basierenden Ansatz für das ganzheitliche Call-Graph-Reasoning in JavaScript, der komplexe Aufrufbeziehungen rekonstruiert, die von traditionellen statischen Analysen übersehen werden. Zusammengenommen zeigen diese Beiträge, dass die Bewältigung der Herausforderungen moderner Softwaretechnik einen Perspektivwechsel hin zu einem sozio-technischen Ansatz erfordert, der soziale und sprachliche Kontexte mit systemweiter Analyse von Komplexität, Skalierung und Sicherheit integriert.

Abstract

Software systems have evolved from isolated artifacts into large, globally distributed ecosystems. This shift has exposed socio-technical blind spots that traditional software engineering and security practices do not fully address. This dissertation argues that securing and sustaining modern software systems requires a socio-technical approach that accounts for demographic and linguistic diversity, architectural complexity, and evolving threat models.

To this end, we first present a comparative study of 200,000 websites across ten developed and ten developing countries, showing how economic context shapes performance, security, privacy, and accessibility practices on the web. Second, we examine accessibility challenges in the Global South through the first cross-national accessibility study of 100,000 websites, revealing persistent barriers for users with disabilities. Building on this analysis, we introduce `LangCrUX` and `Kizuki`, which expose widespread mismatches between visible content and accessibility metadata on multilingual websites. From the developer perspective, we present a ten-year longitudinal study of GitHub activity, showing that although non-English participation is steadily increasing, it is associated with a visibility penalty that limits project reach and contributor engagement.

Complementing these socio-economic findings, the final part addresses technical challenges arising from emerging vulnerability classes and increasing system abstraction. We provide a Systematization of Knowledge of Regular Expression Denial of Service (ReDoS), highlighting gaps in threat models, definitions, and evaluation methodologies, and analyze the defenses deployed in modern regex engines. To improve reproducibility and evaluation in the JavaScript ecosystem, we introduce SecBench.js, a benchmark suite of real-world vulnerabilities and executable exploits for the npm ecosystem. Finally, we present Graphia, a graph neural network–based approach for whole-program call graph reasoning in JavaScript, which recovers complex call relationships missed by traditional static analysis. Together, these contributions show that addressing the challenges of modern software engineering requires a shift toward a socio-technical perspective that integrates social and linguistic context with system-level reasoning about complexity, scale, and security.

Background of this Dissertation

This dissertation is based on the papers listed below [P1, P2, P3, P4, P5, P6, P7]. Six of these papers have been accepted and published at top peer-reviewed conferences, and one paper [P7] is currently available as a preprint. The author of this dissertation is the first author and primary contributor on six of the seven papers, and shares first authorship on one paper [P5]. For all papers, the author played a central role in formulating the research questions, designing the methodology, conducting the empirical analysis, and writing the manuscripts. In the following, we clarify the contributions of co-authors for each paper.

Chapter 3 is based on our work [P1], published at the ACM Web Conference 2025, which presents a large-scale measurement study of web development practices across economic contexts. Matteo Varvello and Yasir Zaki contributed to problem formulation, technical discussions, and the writing of the paper. Cristian-Alexandru Staicu contributed to discussions and the writing in his role as the author's supervisor.

Chapter 4 is based on our work [P2], published at the IEEE/ACM Symposium on Software Engineering in the Global South (SEiGS) 2025, which examines accessibility practices in the Global South through an empirical study. Matteo Varvello, Yasir Zaki, and Cristian-Alexandru Staicu contributed to the writing of the paper.

Chapter 5 is based on our work published at the ACM Internet Measurement Conference (IMC) 2025 [P3], which reports a large-scale measurement study of the multilingual web from an accessibility perspective. Matteo Varvello, Yasir Zaki, and Cristian-Alexandru Staicu contributed to discussions, methodology design, and the writing of the paper.

Chapter 6 is based on our work accepted at the IEEE/ACM International Conference on Software Engineering (ICSE) 2026 [P4], which presents a longitudinal study of non-English language use and participation patterns in open-source repositories. Manish Kumar Bala Kumar contributed to the analysis. Cristian-Alexandru Staicu contributed to the writing and discussions in his capacity as the author's supervisor.

We present our work published at the ASIA Conference on Computer and Communications Security (AsiaCCS) 2025 [P5] in Chapter 7, which provides a systematization of knowledge on Regular Expression Denial of Service (ReDoS). Berk Çakar contributed equally to the paper and conducted the regular expression engine analysis. Ethan H. Burmane analyzed CVE data from the NVD. James C. Davis and Cristian-Alexandru Staicu contributed to the writing of the paper.

Chapter 8 is based on our work [P6], published at the IEEE/ACM International Conference on Software Engineering (ICSE) 2023, which presents an executable benchmark suite for server-side JavaScript vulnerabilities. Nikos Vasilakis and Michael Pradel contributed to technical discussions and the writing process. Adithya Srinivas Parthasarathy contributed to the creation and curation of the benchmark. Cristian-Alexandru Staicu contributed to the analysis and to the writing of the paper in his role as the author's supervisor.

Finally, Chapter 9 is based on our work currently available as an arXiv preprint [P7], which introduces a learning-based approach for improving JavaScript call graph construction using graph neural networks. Giancarlo Pellegrino and Gianluca De Stefano contributed to technical discussions. Cristian-Alexandru Staicu contributed to technical

discussions and analysis, and guided the paper writing in his capacity as the author's supervisor.

[P1] Bhuiyan, M. H. M., Varvello, M., Staicu, C.-A., and Zaki, Y. Digital Disparities: A Comparative Web Measurement Study Across Economic Boundaries. In: *Proceedings of the ACM on Web Conference 2025*. WWW '25. Association for Computing Machinery, Sydney NSW, Australia, 2025, 1889–1900.

[P2] Bhuiyan, M. H. M., Varvello, M., Staicu, C.-A., and Zaki, Y. Non-Western Perspectives on Web Inclusivity: A Study of Accessibility Practices in the Global South. In: *2025 IEEE/ACM Symposium on Software Engineering in the Global South (SEiGS)*. IEEE. 2025, 59–64.

[P3] Bhuiyan, M. H. M., Varvello, M., Zaki, Y., and Staicu, C.-A. Not All Visitors are Bilingual: A Measurement Study of the Multilingual Web from an Accessibility Perspective. In: *Proceedings of the 2025 ACM Internet Measurement Conference*. IMC '25. Association for Computing Machinery, USA, 2025, 871–881.

[P4] Bhuiyan, M. H. M., Bala Kumar, M. K., and Staicu, C.-A. "Write in English, Nobody Understands Your Language Here": A Study of Non-English Trends in Open-Source Repositories. In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2026.

[P5] Bhuiyan, M. H. M., Çakar, B., Burmane, E. H., Davis, J. C., and Staicu, C.-A. SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS). In: *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '25. Association for Computing Machinery, 2025, 1659–1675.

[P6] Bhuiyan, M. H. M., Parthasarathy, A. S., Vasilakis, N., Pradel, M., and Staicu, C.-A. SecBench. js: An executable security benchmark suite for server-side JavaScript. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, 1059–1070.

[P7] Bhuiyan, M. H. M., De Stefano, G., Pellegrino, G., and Staicu, C.-A. Call Me Maybe: Enhancing JavaScript Call Graph Construction using Graph Neural Networks. *arXiv preprint arXiv:2506.18191* (2025).

Further Contributions of the Author

In addition to the primary works presented in this thesis, the author also contributed to several additional publications [S1, S2, S3, S4]. Among these, [S2] presents preliminary results that were later extended and refined in [P7]. Furthermore, [S1] extends research originally conducted as part of the author's master's thesis and was further developed after joining the PhD program.

[S1] Arifuzzaman, M., Bhuiyan, M., Gumus, M., and Arslan, E. Be SMART, Save I/O: A probabilistic approach to avoid uncorrectable errors in storage systems. In: *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2022, 256–266.

[S2] Bhuiyan, M. H. M. The Call Graph Chronicles: Unleashing the Power Within. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. Association for Computing Machinery, San Francisco, CA, USA, 2023, 2210–2212.

[S3] Bhuiyan, M. H. M. Developers' Practices, Web Accessibility, and Language Barriers: Analyzing Challenges and Opportunities for Inclusive Digital Development. In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, New York, NY, USA, 2025, 846–851.

[S4] Bhuiyan, M. H. M. and Staicu, C.-A. A Tale of Frozen Clouds: Quantifying the Impact of Algorithmic Complexity Vulnerabilities in Popular Web Servers. *arXiv preprint arXiv:2211.11357* (2022).

Acknowledgments

This dissertation marks the conclusion of a significant chapter in my life. It is the result of a long journey that would not have been possible without the guidance, support, and companionship of many people who shared this path with me.

First and foremost, I want to thank my advisor, Cristian-Alexandru Staicu. Thank you for trusting me, especially during moments when I doubted myself. Your guidance shaped both this dissertation and my growth as a researcher. I am grateful for the freedom you gave me to explore ideas, make mistakes, and find my own direction, while always stepping in with clear feedback and perspective when I needed it most. I am especially thankful for your support during some of the most difficult moments of my PhD, including personal loss and the uncertainty of the COVID period. Your patience, understanding, and continued belief in my work helped me move forward when both research and life felt uncertain. Working with you has been a privilege, and I will always value what I learned under your supervision.

Next, I owe special thanks to my colleague and friend, Abdullah AlHamdan. Your support went far beyond work. Thank you for our endless discussions during office hours, on the way back home, and over coffee, where conversations moved naturally from research to life and philosophy. Thank you for your support during difficult moments, for your steady help, and for being a friend I could always rely on. And my friend, always remember: "The night is darkest just before dawn."

I also want to give a big shout-out to my colleagues from RA6, Soheil Khodayari, Giada Stivala, Andrea Mengascini, Gianluca De Stefano, Aleksei Stafeev, Jannis Rautenstrauch, Carolyn Guthoff, Alexander Ponticello, Shubham Agarwal, Florian Hantke, Dañiel Gerhardt, and Dominic Troppmann, for making my PhD journey at CISPA a shared and enjoyable experience. I am especially grateful to Jannis Rautenstrauch for carefully reviewing this thesis and providing valuable feedback. Thank you all for the insightful discussions, coffee breaks, long lunches, and the random conversations that made even stressful days easier.

I would also like to thank Andreas Zeller for welcoming me into his group and for providing a supportive research environment. I thank the group members for the open discussions and for making the transition smooth. I am especially thankful to Alexi Turcotte for the help, conversations, and support since I joined the group, and for taking the time to review this thesis and provide thoughtful feedback.

I am grateful for the opportunity to work with my co-authors, Nikos Vasilakis, Yasir Zaki, Matteo Varvello, Michael Pradel, James C. Davis, Berk Çakar, Engin Arslan, and Md Arifuzzaman. Your input, feedback, and collaboration shaped my research in important ways, and I learned a great deal from working with you.

In addition, I want to thank my peers at CISPA and Saarland University, especially Hossein Hajipour, Bhupendra Acharya, Moonis Ali, and Fahad Hilal, for the many conversations and shared moments along the way. I am also grateful to the CISPA faculty, Ben Stock, Katharina Krombholz, Aurore Fass, and Giancarlo Pellegrino, for the fruitful discussions, the Zoom meetings during COVID that helped me stay sane, and the small moments over lunch that made the working environment enjoyable. I also thank the CISPA staff, including the front office, travel department, scientific writing and presentation support, and IT services, for their continuous support.

Living away from one's homeland is never easy, but the warmth of the SaarBangla community in Saarland made it feel like home. Thank you to everyone in the SaarBangla community for the gatherings, traditional food, and sense of belonging. You kept me connected to my roots and provided a welcome break from the pressures of academic life.

Now, I turn to my family, who have been my anchor through the storms. My heart carries a heavy weight knowing that my father is not here to witness this milestone. He passed away during this journey, but his spirit, his values, and his unshakeable belief in the importance of education are woven into every page of this dissertation. To my mother, words cannot fully express my gratitude. Thank you for your endless prayers, your sacrifices, and your unconditional love. Your strength has always been my inspiration to keep moving forward.

I also want to thank my brother, Mahmodul Hasan, my sisters, Ayesha Islam Bhuiyan, Farjana Islam Bhuiyan, and Jahanara Islam Bhuiyan, my nephews, and my extended family in Bangladesh. Even from far away, your love, encouragement, and constant check-ins meant more than you know. Your support reminded me where I come from and gave me strength throughout this journey.

And finally, to my wife, Saima, and my son, Khalid. Thank you for your immense patience during my late nights and tight deadlines. To my wife, thank you for holding the fort and for being my greatest cheerleader when I was exhausted. To my son, your smile is the best reward after a long day. You both bring joy to my life and gave me the ultimate purpose to complete this work.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction

Software engineering is undergoing a profound transformation. Classic accounts describe software development as driven by relatively small, co-located teams working in homogeneous technical and organizational settings, with an emphasis on correctness, modularity, and process control as primary objectives [290, 429, 453]. In contrast, modern software engineering practices, exemplified by Open Source Software (OSS) development, now involve millions of developers distributed across global ecosystems [333, 60], producing software that underpins essential infrastructure for billions of users worldwide. The *scale, diversity, and criticality* of today's systems mean that approaches adequate for earlier contexts no longer capture the full range of challenges. These changes are visible in both social and technical dimensions of modern practice.

One striking example is India's Aadhaar program [497]. Aadhaar is the world's largest biometric identification system, enrolling over 1.3 billion people[1] and serving as a mandatory credential for accessing a wide range of public services, from food rations to pensions and health care. In a country with 22 official languages and hundreds more local languages, however, linguistic variation has proven to be a critical point of failure. The Unique Identification Authority of India (UIDAI) [497] itself acknowledges that enrollment data is collected in English and then transliterated into local scripts, an error-prone process that often produces inconsistencies between Roman and Indic scripts in official records [496]. Civil society reports document cases where such mismatches in spelling [406, 254] or script prevented vulnerable citizens especially the rural poor from accessing subsidized food [483] or medical treatment, in some instances with fatal outcomes [95, 264]. What may seem like a minor orthographic discrepancy in a database is, in practice, a barrier to survival. The Aadhaar case underscores how linguistic issues can directly undermine security, reliability, and inclusion in digital systems that underpin basic rights.

While linguistic and demographic diversity exposes the social dimension of these evolving practices, the technical dimension faces its own set of challenges driven by architectural complexity and evolving threat models. On the technical front, new classes of vulnerabilities are emerging that challenge long-standing security assumptions and require system-level reasoning. A representative example is the *Regular Expression Denial of Service (ReDoS)* attack [114, 209, 426, 459], a form of algorithmic complexity vulnerability that exploits inefficiencies in regular-expression engines to exhaust computational resources. Our work shows that ReDoS is the fourth most common vulnerability in the npm ecosystem, and Snyk reports that it is one of the fastest-growing vulnerability classes[2]. At the same time, research activity on ReDoS has remained steady for more than a decade, with new papers continuing to appear in top venues in security, software engineering, and programming languages (Figure 1.1). However, unlike classical injection vulnerabilities such as SQL or code injection, which are uniformly exploitable regardless of deployment context, ReDoS is a *context-dependent vulnerability*: its exploitability and impact depend heavily on system architecture, input patterns, and

[1] `https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/inspired/biometrics`

[2] `https://snyk.io/blog/redos-vulnerabilities-in-npm-spikes-by-143-and-xss-continues-to-grow`

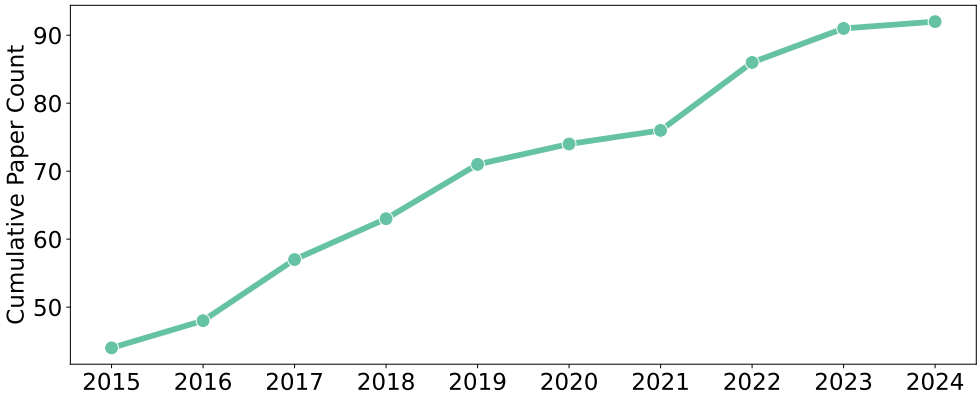


Figure 1.1: Number of research papers on ReDoS vulnerabilities published per year (2015–2024) across top venues in security (IEEE S&P, USENIX Security, ACM CCS, NDSS), software engineering (ICSE, FSE, ASE), and programming languages (PLDI, POPL, OOPSLA). The steady publication rate illustrates continued academic attention to ReDos.

deployment configuration. The same vulnerable regex that may merely slow down a single process in one system can, in another, block event loops, exhaust shared threads, or cascade into service outages. This dependency on architecture and workload means that understanding ReDoS requires end-to-end reasoning about runtime behavior, rather than pattern matching or static inspection of code fragments.

Despite its prevalence and continued academic attention, practitioners remain skeptical of ReDoS as a meaningful security threat. This skepticism is frequently voiced in developer forums, issue trackers, and CVE discussions. Engineers at GitHub describe it as *"not particularly serious, but easy to create by accident, obscure to understand, and sometimes tricky to fix"* [33], while others dismiss such vulnerabilities as *"indistinguishable from malicious noise"* [533] and a source of security fatigue. A review of public vulnerability reports further illustrates this attitude: developers often reject reported ReDoS findings as unlikely to matter in practice, arguing that such cases are artifacts of testing rather than real exploits.

> *"Most ReDoS vulnerabilities are self-attacks, meaning not a real vulnerability."*[3]
>
> *"None of these seem exploitable in real-world scenarios."*[4]

Consistent with these perceptions, Snyk identifies ReDoS as the *most neglected vulnerability type* in its industry reports, noting that it receives the lowest prioritization among reported security issues[5]. These reactions indicate a clear gap between the academic and practitioner communities. Addressing this divide requires rethinking how emerging classes of vulnerabilities are identified, modeled, and mitigated within modern software engineering and security practice.

[3] `https://github.com/sarbbottam/eslint-find-rules/issues/349#issuecomment-1863661519`

[4] `https://github.com/facebook/react-native/issues/46996#issuecomment-2410140810`

[5] `https://snyk.io/reports/open-source-security/`

One may hope that traditional security tools and engineering methods would suffice to address such issues. However, this is not the case: modern software engineering must confront a set of structural challenges that reflect how systems are designed, who builds them, and how they operate in practice. Addressing these challenges requires adopting a *socio-technical systems* perspective [351, 44]. Below, we discuss four such challenges in detail.

C_1**: Demographic Diversity** The fastest growth on the web is taking place in regions where English is not dominant, including China, Brazil, Indonesia, and Egypt [271]. As English's share of web content has declined from around 80% in 1998 to between 20–50% today [149, 387, 389], millions of new contributors participate using diverse languages [189, 118]. Surveys show nearly a quarter of open-source developers report limited English proficiency[6]. In practice, this creates barriers in collaborative platforms like GitHub, where communication depends on issue discussions, pull requests, and documentation. Studies of projects such as OpenStack and GNOME show that non-native English speakers are often restricted to translation or peripheral tasks, while struggling to advance into core code contributions due to communication difficulties[7] [467]. These disruptions highlight that regional and linguistic diversity is not just a social factor but a structural challenge for modern software engineering.

C_2**: Unicode Adoption** Multilingualism is no longer a peripheral concern in modern software engineering; it directly affects how systems are designed, analyzed, and secured. Many development tools and analysis pipelines still assume that identifiers, comments, and documentation are written in English and limited to ASCII [46, 222, 132]. However, the rapid growth of web users and developers from the Global South has brought Chinese, Portuguese, Hindi, Arabic, and many other languages into software artifacts and user-facing content [387, 389]. Until recently, even major software ecosystems reflected these monolingual assumptions. Google's Gmail only began recognizing non-Latin email addresses in 2014[8], Microsoft extended support to Office 365 and Exchange Server in 2018[9], and the Indian company XgenPlus became the world's first full EAI (Email Address Internationalization) mailbox provider[10].

This shift introduces new blind spots across software systems. In security, tools trained primarily on English data struggle to detect multilingual attacks such as phishing, while Unicode-based exploits like Trojan Source demonstrate how script variation can be exploited to bypass both automated analysis and human review [233, 22, 70]. In accessibility, metadata such as alt text or ARIA labels is often written in a different language than the surrounding interface, leading to mispronunciations or silent failures in screen readers [435, 183, 546]. Treating multilingualism as an edge case rather than a system-level constraint risks overlooking real failures in both security and inclusion.

C_3**: Evolving Threat Models and Systemic Risks** Modern software engineering increasingly follows a compositional model, in which applications are built by integrat-

[6] https://opensourcesurvey.org/2017/

[7] https://chaoss.github.io/website/CHAOSScon/2020EU/slides/language_barriers.pdf

[8] https://techcrunch.com/2014/08/05/gmail-now-works-with-addresses-with-non-latin-characters

[9] https://uasg.tech/2020/11/support-for-email-addresses-in-local-languages-grows-with-major-

[10] https://uasg.tech/2017/02/universal-acceptance-india/

ing large numbers of third-party libraries, frameworks, cloud services, and reusable components. This shift has fundamentally altered the security landscape. Instead of vulnerabilities arising solely from local implementation errors, many security risks now emerge from how components interact, how trust is delegated across dependency chains, and how systems behave under adversarial or worst-case conditions. This compositional nature enables new classes of vulnerabilities that are difficult to reason about using traditional threat models. Software supply chain attacks exploit implicit trust in external dependencies and transitive package relationships. Availability and logic-based attacks exploit algorithmic behavior, shared resources, or assumptions about benign inputs. Regular Expression Denial of Service (ReDoS) is a representative example: the vulnerability often does not stem from an incorrect implementation, but from worst-case interactions between input patterns, regex engines, and runtime execution models. In many cases, no single component is faulty in isolation; the vulnerable behavior only manifests at the system level, when independently developed components are combined and exercised in unexpected ways.

Despite their growing relevance, these systemic risks remain poorly understood and inconsistently defined. Existing security frameworks and benchmarks were largely designed for vulnerabilities with clear triggers, localized effects, and well-scoped attacker models. In contrast, modern threats such as ReDoS or supply chain attacks depend heavily on deployment context, workload characteristics, dependency structure, and evolving ecosystems. As a result, there is limited agreement on how to model attackers, define exploitability, or evaluate the severity of these vulnerabilities in a consistent and reproducible manner. This mismatch between evolving threat models and existing security abstractions leaves both researchers and practitioners without reliable tools to reason about risk in modern software systems. Addressing this challenge requires rethinking how vulnerabilities are defined, modeled, and evaluated, with a stronger emphasis on system-level behavior and real-world execution contexts.

C_4**: Rising Complexity and Abstraction** Modern software engineering is also marked by growing complexity and abstraction, where layered frameworks, dynamic execution models, and asynchronous control flow obscure program structure and runtime behavior. Understanding and securing such systems, therefore, requires comprehensive program analysis capable of reasoning across modules, libraries, and runtime behaviors rather than isolated code fragments. At the core of such analyses lies call graph construction, the process of identifying possible function calls and data flows within a program, which supports tasks such as dependency resolution, vulnerability detection, and performance profiling. In highly abstracted systems, however, call relationships are often implicit or resolved only at runtime, making them difficult to recover accurately.

Existing analysis techniques struggle in this setting, particularly for dynamic languages such as JavaScript. Traditional static analysis approaches often miss many call edges while introducing large numbers of spurious ones due to JavaScript's dynamic and asynchronous nature [25, 501]. Dynamic analysis can provide higher precision but is computationally expensive, coverage-dependent, and difficult to scale across real-world projects. Recent learning-based approaches have improved parts of this picture but remain limited in scope. Machine learning and pre-trained models such as CodeBERT [158] improve local reasoning at the function level, yet fail to capture

whole-program context or long-range dependencies. Large language models offer greater flexibility but remain constrained by limited context windows and high computational cost. Graph-based learning approaches, including graph neural networks (GNNs), show promising progress in modeling full-program structure and semantics [14, 220], yet their adoption in practical analysis tools remains limited. As a result, a significant gap persists between the complexity and abstraction of modern software systems and the analytical methods currently used to reason about and secure them.

Considering all these challenges, this dissertation supports the following thesis:

> Modern software engineering practices present unique challenges and opportunities that demand socio-technical approaches accounting for linguistic and demographic diversity, alongside the integration of machine learning techniques for whole-program and system-level reasoning.

While the challenges identified above are not specific to any single platform, they are especially visible in web-based systems, which combine global reach, heterogeneous users, third-party dependencies, and rapid evolution [165, 559, 394]. Accordingly, much of this dissertation studies these challenges through the lens of the modern web, using it as a concrete and widely deployed setting for empirical and technical analysis.

Concretely, we support this thesis (i) through large-scale empirical studies of web ecosystems, including a comparative analysis of 200,000 websites across 20 countries and the first large-scale cross-national accessibility study of 100,000 websites from the Global South. Together, these studies show how economic context, cultural norms, and legal frameworks shape performance, security, privacy, and accessibility practices, (ii) by examining the impact of linguistic factors on both users and developers: we conduct the first cross-national accessibility study in the Global South, build `LangCrUX` to analyze multilingual accessibility practices, design the language-aware auditing tool `Kizuki`, and perform a longitudinal study of multilingual participation on GitHub, (iii) by analyzing academic and engineering progress on Regular Expression Denial of Service (ReDoS) through a systematization of knowledge, and by developing SecBench.js, the first executable benchmark suite of JavaScript vulnerabilities, and (iv) by advancing program analysis with Graphia, a GNN-based approach to whole-program reasoning in JavaScript.

1.2 Outline of the Thesis

This dissertation is organized into ten chapters that together address the four challenges of modern software engineering outlined above. Chapter 1 introduces the motivation, scope, and central thesis of the dissertation, while Chapter 10 concludes the dissertation and outlines directions for future work. Chapter 2 establishes the technical foundations of the dissertation, covering web ecosystems and performance metrics, accessibility standards, regular expressions, and graph representation learning.

The core contributions are presented in three parts. The first part (Chapter 3 and 4) addresses demographic diversity (challenge C_1) by examining the state of the web ecosystem through large-scale measurement, focusing on economic context and

accessibility practices across regions, including a cross-national study of the Global South. The second part (Chapter 5 and 6) addresses Unicode adoption and linguistic diversity (challenge C_2) through studies of multilingual accessibility and inclusion. This part presents cross-national analyses and introduces datasets and tools that capture multilingual accessibility practices, as well as examining how linguistic diversity shapes collaboration and visibility in global open-source communities. The third part (Chapter 7, 8, and 9) focuses on emerging vulnerabilities and shortcomings of program analysis techniques. It addresses evolving threat models (challenge C_3) and rising complexity and abstraction (challenge C_4) by systematizing knowledge of Regular Expression Denial of Service (ReDoS), introducing SecBench.js, an executable benchmark suite of server-side JavaScript vulnerabilities, and presenting Graphia, a graph neural network–based approach to whole-program reasoning for JavaScript.

In Chapter 3, we systematically analyze web development practices across different economic contexts. We analyze 200,000 websites drawn from the most populated countries, comparing 10 developed and 10 developing nations. This large-scale comparison provides a unique opportunity to examine how design and development practices differ across regions with very different levels of connectivity, infrastructure, and regulatory environments. The analysis focuses on several aspects that directly shape user experience. We look at *page size and complexity*, which influence both load times and data consumption. Smaller and simpler pages are often more suitable for mobile-first access, which dominates in developing regions [244]. *Performance practices*, such as the use of optimized image formats and the removal of unused JavaScript or CSS, show whether developers adapt sites to the constraints of resource-limited networks [286]. *Security* is another important dimension. The adoption of HTTPS, the avoidance of outdated libraries, and the use of Content Security Policy reflect how well websites protect users from interception or exploitation [428]. *Privacy practices*, including the use of tracking scripts and cookies, reveal how business models and regulation affect user protection across contexts [151]. We also study *accessibility*, since adherence to guidelines such as WCAG is necessary for inclusive participation [292]. Finally, we examine the *adoption of modern protocols and APIs*, which indicates whether countries lag behind or leapfrog older technologies [363]. Together, these aspects provide a systematic picture of how economic conditions shape both the structure and quality of the web.

Complementing this analysis, Chapter 4 address the unique accessibility challenges of the Global South by conducting the first large-scale cross-national study of 100,000 websites across ten countries. We focus on mobile platforms because they represent the dominant mode of access in these regions, yet accessibility support remains limited. Our analysis identifies common challenges such as missing alternative text, poor ARIA usage, and low-contrast design elements. We also look into how diverse disability groups face different issues, as not all users are impacted equally by accessibility violations. For example, blind and low-vision users are disproportionately excluded when alt text is missing, while motor-impaired users encounter barriers with small or poorly designed interaction targets.

While examining these accessibility issues, we found a particularly important phenomenon: a frequent mismatch between the visible text on websites and the information provided in accessibility metadata such as alt text and *lang* attributes. To study this

problem systematically, we needed large-scale data on websites in diverse languages, but no such dataset existed. We therefore developed a new dataset of 120,000 websites in twelve non-Latin-script languages in Chapter 5, enabling the first cross-linguistic analysis of accessibility practices. We leverage this dataset, `LangCrUX`, to conduct the first large-scale analysis of multilingual web accessibility, focusing on how assistive technologies interact with real-world multilingual content. This analysis reveals that a large share of websites lack accessibility text in the native language, even when the visible content is predominantly presented in that language. We also found a critical limitation of existing automated accessibility testing tools, which typically check only for the presence of accessibility hints but fail to detect whether they are accurate or consistent with visible content. To address this gap, we designed `Kizuki`, a language-aware auditing tool that evaluates the alignment between accessibility metadata and surrounding text.

In the same vein of examining the impact of language on the developers' side, in Chapter 6 we conduct the first longitudinal analysis of multilingual participation on GitHub, covering 9.14 billion messages and 62,500 repositories across a ten-year period. This scale allows us to capture how language shapes communication, collaboration, and visibility in open-source projects. These repositories span five widely used programming languages: JavaScript, Python, TypeScript, Java, and C#. Using language detection and classification tools, we identify the presence of 30 natural languages across different parts of open-source collaboration, including code elements such as identifiers and string literals, documentation, and user-generated messages in issues and pull requests. Prior research has largely focused on communication, with only a single study examining non-English content in code, leaving a major gap in understanding how linguistic diversity manifests in the artifacts of software development. This gap motivates us to examine how multilingual participation has changed over time, how it manifests differently across communication channels and code artifacts, and how language use affects project visibility and developer engagement. We also look into how different communities face distinct challenges: projects that use non-English languages often receive less attention and participation, while developers with limited English proficiency encounter obstacles in mentoring, onboarding, and collaboration. These findings highlight a tension between the desire to use native languages for expression and the pressure to conform to English as the lingua franca of software development. By systematically studying this dynamic, we provide the first large-scale evidence of how language both enables inclusion and reinforces barriers in the global open-source ecosystem.

Next, we turn to emerging vulnerabilities that require clearer system-level understanding. A prominent example is Regular Expression Denial of Service (ReDoS), a long-recognized vulnerability class that has attracted significant research attention but remains insufficiently systematized. To address this gap, in Chapter 7 we conduct the first comprehensive literature review of academic work on ReDoS, covering 35 papers published in top security conferences. We systematically categorize research directions, examine the threat models assumed, analyze the vulnerability definitions employed, and assess the evaluation methodologies used across prior work. One of our key observations is that nearly all prior studies rely on two classical models of regex engine behavior: Thompson's automata-based construction from 1968 and Spencer's backtracking engine

from 1994. While foundational, these models no longer fully capture modern regex engines, many of which now incorporate partial or full defenses against catastrophic backtracking. Motivated by this disconnect, we complement the literature review with an engineering review of regex engines used in major programming languages. This review examines how engines in ecosystems such as JavaScript, Java, Python, and Rust implement defenses, including algorithmic changes, resource caps, and non-backtracking execution modes. By analyzing engine designs, deployment practices, and adoption of defenses, we provide the first consolidated view of how ReDoS is handled in practice today. This dual perspective, combining a synthesis of academic knowledge with an engineering survey of deployed systems, clarifies where ReDoS remains a real risk, where mitigations are effective, and where academic research has moved out of alignment with practice.

Building on this system-level perspective, in Chapter 8 we also address the absence of reproducible evaluation, a persistent gap in JavaScript security research. To fill this gap, we build SecBench.js, the first executable benchmark suite of server-side vulnerabilities in the npm ecosystem. SecBench.js contains 600 real-world cases across major vulnerability classes, each paired with an exploit and a validated fix. A reusable benchmark suite of this kind alleviates long-standing problems of fragmented datasets and helps improve both experimental techniques and evaluation metrics. We design the benchmark around four key properties. It is *realistic*, as it is built from a diverse set of real-world packages with minimal modifications to the original code, ensuring that results generalize beyond the benchmark itself. It is *executable*, including inputs that trigger the vulnerabilities and runtime oracles that confirm successful exploitation, enabling systematic evaluation of runtime defenses. It is *two-sided*, containing both vulnerable and fixed versions of each package, which allows researchers to measure not only detection but also false positive rates and remediation strategies. Finally, it is *vetted*, with each entry manually checked to confirm the vulnerability, its class, and the correctness of metadata. By combining these properties, SecBench.js provides a robust and reusable foundation for reproducible security research in JavaScript.

Lastly, in Chapter 9, to address limitations in program analysis, we propose Graphia, a learning-based approach to improve call graph construction for JavaScript. Call graphs are central to interprocedural analysis, yet existing static tools struggle with JavaScript's dynamic features such as higher-order functions, dynamic property accesses, and prototype inheritance, often leaving many call sites unresolved. Graphia tackles this challenge by representing the entire program as a graph that combines both syntactic and semantic edges, linking not only structural relationships from the abstract syntax tree but also connections between identifiers and data flows. We then model call graph augmentation as a link prediction task and apply graph neural networks to infer likely but missing call edges. This holistic representation allows the model to capture non-local patterns, such as the passing of anonymous or higher-order functions, that traditional static analysis tools often miss. Unlike prior approaches that rely on embeddings of local code fragments, Graphia operates directly on the whole-program graph, closer to the spirit of interprocedural analysis. This design enables it to handle complex contextual relationships and reduce the number of unresolved call sites without overwhelming developers with false positives. By integrating machine learning with program analysis,

Graphia demonstrates how modern techniques can overcome long-standing obstacles in reasoning about dynamic languages at scale.

1.3 Contributions

This dissertation brings together several empirical studies, datasets, and analysis techniques that examine both the social and technical dimensions of the modern software engineering. As further discussed in §1.4, this work builds on a series of peer-reviewed studies, each contributing specific insights recognized by the research community. At a high level, the dissertation advances three broad research themes, each developed in later chapters.

Digital Disparity Across Economic Contexts We establish a new empirical basis for understanding differences in the global web. By measuring 200,000 websites from twenty countries, we compare core aspects of web development across developed and developing regions. The study contrasts page size, complexity, performance, security, privacy practices, and protocol adoption. One observation is that websites in developing regions are generally smaller and structurally simpler, yet still waste more bandwidth through oversized images and unused code. These results provide the first systematic view of how economic conditions shape everyday web experience.

We also conduct the first cross-national study of mobile accessibility across ten countries in the Global South. Evaluating 100,000 websites, we identify recurring issues such as inadequate touch targets, low-contrast elements, and missing text descriptions. These problems affect users unevenly: blind and low-vision users encounter the most severe barriers, and compliance varies with national policy environments.

Multilinguality in Web Accessibility and Development To study accessibility in multilingual contexts, we build `LangCrUX`, a dataset of 120,000 websites spanning twelve non-Latin-script languages. This enables the first large-scale, language-aware analysis of how textual content is presented to assistive technologies. A key finding is the frequent mismatch between visible content and accessibility metadata. Many sites present native-language interfaces while providing English-only accessibility text. To address the limitations of existing checkers, we introduce `Kizuki`, a tool that inspects both the presence and the linguistic consistency of accessibility metadata.

We further examine how linguistic diversity shapes open-source collaboration. Analyzing 9.14 billion GitHub messages across 62,500 repositories over ten years, we show that non-English content has increased sharply, with an overall rise of about 164%, driven mostly by Korean, Chinese, and Russian contributions. Much of this growth appears in human-facing code elements such as comments and string literals rather than in structural identifiers. At the project level, repositories dominated by non-English languages attract fewer stars and contributors, indicating a visibility gap, yet they resolve issues faster than English-dominant projects. This provides the first large-scale longitudinal view of how multilingual activity affects engagement and coordination in open-source ecosystems.

Advancing Program Analysis for Emerging Vulnerabilities We systematize the state of Regular Expression Denial of Service vulnerabilities through a comprehensive review of 35 research papers. The study compares how prior work defines vulnerability conditions, evaluates exploitability, and models attacker capabilities. A key outcome is the identification of gaps in the literature, providing a basis for more practical and impactful research. Complementing this, an engineering review of regex engines used in JavaScript, C#, Ruby, and other ecosystems shows that many now integrate linear-time or partially linear-time defenses such as memoization, bounded backtracking, and non-backtracking execution modes. These changes weaken or eliminate several classical attack strategies. The combined review clarifies where ReDoS remains a practical threat and where long-standing threat models no longer reflect deployed systems.

To address long-standing issues around reproducibility in JavaScript security, we introduce SecBench.js, an executable benchmark suite of 600 real-world vulnerabilities. Each entry contains the vulnerable version, an exploit that triggers the issue, and the corresponding patched version. The benchmark covers major vulnerability classes such as command injection, prototype pollution, path traversal, and ReDoS. The two-sided nature of the benchmark supports evaluating both detections and false positives. Applying SecBench.js led to concrete findings: we identified 168 package versions whose vulnerability labels in public databases were incorrect, and 18 cases where published fixes were incomplete or faulty, yielding 20 previously unknown zero-day vulnerabilities across npm packages.

We present Graphia, a graph neural network (GNN) approach that augments JavaScript call graphs. Graphia models call-graph enhancement as a link prediction task over a holistic, multi-file program graph and resolves call sites that static tools leave unresolved by leveraging both structural and semantic information. To our knowledge, this is the first work to demonstrate that GNN-based link prediction is feasible at whole-program scale for JavaScript. Through extensive experiments, we show that Graphia can learn from imperfect ground truths, using either the static edges produced by baseline analysis tools or a combination of static and dynamically extracted edges.

We evaluate Graphia on 50 popular npm libraries and find that, for statically unresolved call sites, it ranks the correct callee first in 42% of cases and within the top five in 72%. An ablation study highlights the importance of our design: removing the program representation reduces performance by 61%, and removing node features leads to a 13% drop. These results show that Graphia can learn from imperfect data and effectively support existing call graph construction methods in performing interprocedural inferences.

1.4 List of Publications and Open-Source Implementations

This dissertation is based on several peer-reviewed publications, from which it reuses material where appropriate. The following works form the core foundation of the chapters presented in this document:

1. [P1] **Digital Disparities: A Comparative Web Measurement Study Across Economic Boundaries.** Bhuiyan, Masudul Hasan Masud, Matteo

Varvello, Cristian-Alexandru Staicu, and Yasir Zaki. In Proceedings of the ACM on Web Conference 2025, pp. 1889-1900. 2025.

2. [P2] **Non-Western Perspectives on Web Inclusivity: A Study of Accessibility Practices in the Global South.** Bhuiyan, Masudul Hasan Masud, Matteo Varvello, Cristian-Alexandru Staicu, and Yasir Zaki. In 2025 IEEE/ACM Symposium on Software Engineering in the Global South (SEiGS), pp. 59-64. IEEE, 2025.

3. [P3] **Not All Visitors are Bilingual: A Measurement Study of the Multilingual Web from an Accessibility Perspective.** Bhuiyan, Masudul Hasan Masud, Matteo Varvello, Yasir Zaki, and Cristian-Alexandru Staicu. In Proceedings of the 2025 ACM Internet Measurement Conference, pp. 871-881. 2025.

4. [P4] **"Write in English, Nobody Understands Your Language Here": A Study of Non-English Trends in Open-Source Repositories** Bhuiyan, Masudul Hasan Masud, Manish Kumar Bala Kumar, and Cristian-Alexandru Staicu. In 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE), IEEE, 2026.

5. [P5] **SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS).** Bhuiyan, Masudul Hasan Masud, Berk Çakar, Ethan H. Burmane, James C. Davis, and Cristian-Alexandru Staicu. In Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, pp. 1659-1675. 2025.

6. [P6] **SecBench. js: An executable security benchmark suite for server-side JavaScript.** Bhuiyan, Masudul Hasan Masud, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1059-1070. IEEE, 2023.

7. [P7] **Call Me Maybe: Enhancing JavaScript Call Graph Construction using Graph Neural Networks.** Bhuiyan, Masudul Hasan Masud, Gianluca De Stefano, Giancarlo Pellegrino, and Cristian-Alexandru Staicu. arXiv preprint arXiv:2506.18191 (2025).

In Table 1.1, we show the mapping between these publications and the chapters of this dissertation.

In order to encourage future work, support open science, and enable the community to build upon our results, this dissertation also releases the following open-source artifacts:

- SecBench.js : A public benchmark suite of executable JavaScript vulnerabilities. `https://github.com/cristianstaicu/SecBench.js`
- Graphia : A GNN-based system for enhancing JavaScript call graphs. `https://github.com/kal-purush/ml4call-graph`

Table 1.1: Mapping between publications and dissertation chapters.

Related Publications	Chapter
Digital Disparities: A Comparative Web Measurement Study Across Economic Boundaries [P1]	Chapter 3
Non-Western Perspectives on Web Inclusivity: A Study of Accessibility Practices in the Global South [P2]	Chapter 4
Not All Visitors are Bilingual: A Measurement Study of the Multilingual Web from an Accessibility Perspective [P3]	Chapter 5
"Write in English, Nobody Understands Your Language Here": A Study of Non-English Trends in Open-Source Repositories [P4]	Chapter 6
SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS) [P5]	Chapter 7
SecBench. js: An executable security benchmark suite for server-side JavaScript [P6]	Chapter 8
Call Me Maybe: Enhancing JavaScript Call Graph Construction using Graph Neural Networks [P7]	Chapter 9

- `LangCrUX` : A multilingual dataset of non-Latin-script websites. `https://github.com/kal-purush/LangCrux`
- `Kizuki` : A language-aware extension of Lighthouse for accessibility evaluation. `https://github.com/kal-purush/accessibility_study`
- Bhasha-Nirikhhon : A dataset and analysis framework for measuring multilingual trends in open-source software development. `https://github.com/kal-purush/bhasha-nirikhhon`
- WebContext : Code and data supporting the large-scale comparative study of web ecosystems across economic contexts. `https://github.com/kal-purush/digital-disparities-www25`
- ReDoSSoK : An open repository accompanying the systematization of knowledge on ReDoS. `https://github.com/PurdueDualityLab/SoKReDoS-ASIACCS25`

2
Technical Background

In this chapter, we present the foundational knowledge required to understand the socio-technical analyses and engineering contributions of this work. We begin by introducing the web ecosystem and the metrics used to characterize performance, security, and privacy across different economic contexts (§2.1). We then focus on web accessibility, describing the DOM, the accessibility tree, assistive technologies, and the standards used to evaluate accessible web content (§2.2). Next, we present the foundations of regular expressions and regex engines, including the matching algorithms used in practice and Regular Expression Denial of Service (ReDoS) (§2.3). Finally, we introduce graph representation learning, covering Graph Neural Networks (GNNs), Gated Graph Convolutional Networks (GatedGCNs), and the link prediction formulation used for program analysis tasks (§2.4).

2.1 Web Ecosystems and Performance Metrics

A major contribution of this dissertation is the analysis of web ecosystem across economic contexts. This section introduces the economic classifications and technical metrics used to characterize differences in web performance, security, and privacy across these ecosystems.

2.1.1 Economic Contexts: Developed vs. Developing

We adopt the economic classification defined by the International Monetary Fund (IMF) [252]. The IMF groups countries into two categories: *Advanced Economies* and *Emerging and Developing Economies*, which we refer to as developed and developing, respectively. We use this terminology consistently throughout this dissertation.

Economic classifications are not uniform across institutions. Other organizations, such as the World Bank [534] and the United Nations [499], rely on different criteria, including income per capita, lending eligibility, or broader development indicators. In particular, the World Bank classifies countries using income groups (low, lower-middle, upper-middle, and high income), which do not map directly to the IMF's advanced versus emerging distinction [534]. These classifications are revised regularly, typically on an annual basis, and a country's category may change over time. For example, based on updated GNI per capita thresholds, the World Bank reclassified Russia from an upper-middle-income economy to a high-income economy in 2023 [1], while the IMF categorizes Russia as an emerging and developing economy.

To ensure internal consistency and reproducibility, all analyses in this dissertation follow the IMF classification exclusively, regardless of alternative categorizations used by other institutions.

2.1.2 Global North and Global South

The terms *Global North* and *Global South* are commonly used to describe broad patterns of economic, technological, and infrastructural inequality across regions of the world [117,

[1] https://www.worldbank.org/en/about/leadership/directors/eds23/brief/russia-was-classified-as-high-income-country

328]. They do not refer strictly to geographic location, but rather to historical and structural differences in development, access to resources, and participation in global systems.

The Global North generally includes regions with higher income levels, more mature digital infrastructure, and greater access to high-bandwidth networks and powerful computing devices. In contrast, the Global South includes regions that often face constrained technological conditions, such as limited bandwidth, higher data costs, lower-end devices, and less reliable infrastructure [247]. Under these conditions, design decisions related to performance, accessibility, security, and language have a stronger impact on user experience and inclusion.

The Global North and Global South distinction is not a formal economic classification and does not replace institutional groupings such as those defined by the IMF or the World Bank. Instead, it serves as a descriptive lens for reasoning about structural inequalities in access and capability. In this dissertation, the term *Global South* is used in Chapter 4 to contextualize accessibility and inclusion challenges observed in regions facing persistent infrastructural and economic constraints.

2.1.3 Core Web Vitals

To quantify user experience in an objective and reproducible way, we rely on Core Web Vitals, a set of performance metrics defined by Google [515, 200]. These metrics capture aspects of page loading and interactivity that directly affect how users perceive a website. In this dissertation, we focus on two Core Web Vitals:

- **Largest Contentful Paint (LCP):** LCP measures perceived load speed. It marks the point in the timeline when the page's main content (the largest image or text block) has finished rendering. An LCP below 2.5 seconds is generally considered good. In low-bandwidth environments, large and unoptimized assets can delay LCP substantially, preventing users from accessing the main content of a page.

- **Total Blocking Time (TBT):** TBT serves as a proxy for interactivity. It measures the total amount of time during which the browser's main thread is blocked by long tasks, typically JavaScript execution, that exceed 50 milliseconds. While the main thread is blocked, user interactions such as clicks, taps, or scrolling cannot be processed. High TBT is especially problematic on low-end devices, where complex JavaScript takes longer to parse and execute than on high-end hardware.

2.1.4 Page Weight and Complexity

The cost of a webpage is determined not only by the number of bytes transferred over the network, but also by the computational effort required to process and render those bytes on the client device.

DOM Size The Document Object Model (DOM) is the in-memory tree representation of a webpage created by the browser after parsing the HTML markup [418]. DOM size captures the structural complexity of a page by counting the number of nodes in this tree. Large DOMs, for example those exceeding roughly 1,500 nodes as commonly flagged by browser performance tooling [202], increase memory usage and slow down style recalculation and layout. On devices with limited memory, such as low-end mobile phones, this can lead to poor responsiveness or browser instability.

Resource Composition Resource composition captures how different types of resources contribute to page cost. We distinguish between a static footprint and a dynamic footprint. The static footprint consists of images, fonts, and stylesheets (CSS) and primarily affects download time and data consumption. The dynamic footprint is dominated by JavaScript, which must be downloaded, parsed, compiled, and executed [291, 372]. This process imposes significant CPU and battery costs and directly affects responsiveness, particularly on resource-constrained devices.

2.1.5 Network Protocols and Transport Security

Transport Layer Security (TLS) is a cryptographic protocol that provides confidentiality and integrity for data exchanged between a client and a server [416]. On the web, TLS is used to secure HTTP connections and forms the basis of HTTPS.

HTTP defines how requests and responses are exchanged between browsers and servers. HTTP/1.1 is an older version of the protocol that processes requests largely sequentially [164]. Newer versions, such as HTTP/2 and HTTP/3, introduce multiplexing, allowing multiple requests and responses to be transmitted concurrently over a single connection [48, 62]. In modern browsers, HTTP/2 and HTTP/3 are typically deployed only over encrypted connections using TLS. As a result, transport security and HTTP protocol versions are closely related in practice, with HTTPS serving as the foundation for newer protocol features.

2.1.6 Content Security Policy

Content Security Policy (CSP) is a declarative web security mechanism that allows website operators to restrict which resources a browser is permitted to load for a given page. CSP is delivered via an HTTP response header and defines policies over resource types such as scripts, stylesheets, images, and fonts [529].

CSP is primarily used to mitigate cross-site scripting (XSS) attacks. By specifying an explicit allowlist of trusted sources, CSP prevents the execution of injected or malicious scripts that originate from unapproved domains. CSP can also restrict or disable high-risk features such as inline scripts and dynamic code evaluation, reducing the attack surface for client-side code injection [528].

2.1.7 Third-Party Tracking and Cookies

Modern websites commonly integrate resources from multiple domains. The domain that a user intentionally visits is referred to as the *first party*, while external services

embedded through scripts, images, or iframes are referred to as *third parties*.

Third-party tracking refers to the collection of user behavior by these external services across different websites [319, 425]. This tracking is commonly implemented through third-party scripts used for analytics, advertising, or social media integration, enabling the construction of cross-site user profiles without direct user interaction [151]. Cookies are a primary mechanism used for state management and tracking on the web [42]. *First-party cookies* are set by the visited domain and are typically required for essential functionality such as authentication or session management [425, 42]. *Third-party cookies* are set by external domains and allow trackers to recognize users across multiple websites. Because of their role in cross-site tracking, third-party cookies have significant privacy implications and are a central focus of web privacy regulation and browser policy.

To mitigate these privacy and security risks, including attacks such as Cross-Site Request Forgery (CSRF) [272], modern browsers support the `SameSite` cookie attribute [59]. This attribute controls whether cookies are sent with cross-site requests. The adoption of `SameSite`-by-default policies represents a significant shift in the web ecosystem, effectively restricting unrestricted third-party cookies unless explicitly configured with `SameSite=None` [341].

2.2 Web Accessibility and Assistive Technologies

This section introduces the technical concepts and standards relevant to web accessibility, including assistive technologies, accessibility metadata, and WCAG principles, providing the background information required to comprehend the accessibility analyses examined in this dissertation.

2.2.1 The DOM and the Accessibility Tree

As introduced in §1.1.3, the Document Object Model (DOM) provides the structural representation of a webpage and serves as the foundation for rendering and scripting in the browser. While the DOM encodes both visual and interactive elements, it is not consumed directly by assistive technologies.

Instead, modern browsers derive a separate abstraction known as the *Accessibility Tree* from the DOM [337, 45]. This tree exposes only the semantic and interactive information necessary for accessibility and is the primary interface through which assistive technologies interact with web content. The accessibility tree is derived from, but not identical to, the DOM tree: it consists of accessibility objects rather than DOM nodes and may omit, restructure, or enrich elements based on their semantic relevance.

Each object in the accessibility tree is characterized by four core properties:

- **Role:** The semantic category of the element, indicating the type of interface component it represents (e.g., button, link, heading).
- **Name:** The primary textual label exposed to assistive technologies, computed from visible text and accessibility attributes such as `alt` or `aria-label`.

- **Description:** Optional supplementary text that provides additional context beyond the name, typically used for complex or non-obvious elements.
- **State:** The current interactive condition of the element, such as whether it is focused, disabled, checked, or expanded.

By separating the structural DOM from the accessibility tree, browsers provide assistive technologies with a stable and semantically focused view of the page [427]. At the same time, this separation introduces the possibility of mismatches between visible content and the information exposed to users of assistive tools, which are central to the accessibility analyses conducted in Chapter 5.

2.2.2 Alt Text and ARIA Metadata

Accessibility information exposed through the accessibility tree is derived from both native HTML semantics and explicit accessibility metadata. Two central mechanisms for providing such metadata are alternative text and WAI-ARIA attributes [2]. Alternative text, provided via the `alt` attribute on `img` elements, supplies a textual substitute when visual content cannot be perceived. When present, `alt` text directly contributes to the accessible name of an image and is announced by screen readers in place of the visual content [141, 338]. Missing, empty, or uninformative `alt` attributes therefore result in a loss of semantic information for users of assistive technologies.

WAI-ARIA (Accessible Rich Internet Applications) attributes allow developers to annotate elements with additional semantic information when native HTML alone is insufficient [141]. Attributes such as `aria-label`, `aria-labelledby`, and `aria-describedby` influence how the accessible name and description of an element are computed. When multiple naming mechanisms are present, `aria-labelledby` has the highest priority, followed by `aria-label`. If neither is provided, the accessible name is derived from native markup, such as `alt` text or associated `label` elements, or from the element's visible text content, following the Accessible Name and Description Computation algorithm [182]. ARIA is particularly important for custom interface components built using generic elements such as `div` or `span`.

2.2.3 Screen Readers and Accessibility APIs

Screen readers are assistive software applications that provide non-visual access to digital content by translating on-screen information into alternative output modalities, most commonly synthesized speech or braille. Rather than parsing raw HTML directly, screen readers obtain information through platform-specific accessibility APIs exposed by the operating system [340]. Commonly used screen readers include NVDA [361] and JAWS [179] on Windows, VoiceOver on macOS and iOS [26], and TalkBack on Android.

Interaction with web pages typically follows a linear navigation model, where users traverse the accessibility tree sequentially. To support this interaction, screen readers maintain a virtual buffer often referred to as a *virtual cursor* in JAWS or *Browse Mode* in NVDA [362], which allows users to navigate static content such as text paragraphs

[2] `https://www.w3.org/WAI/standards-guidelines/aria/`

that do not natively receive keyboard focus [68, 178]. When expected accessibility information is missing, screen readers rely on fallback mechanisms, such as inferring names from visible text. For example, standard links and buttons without explicit labels may still be announced using their text content. In contrast, non-text content, such as images without `alt` text or custom interactive widgets built from generic elements (e.g., `div` or `span`), often lack reliable fallbacks. Moreover, such generic elements typically do not trigger the automatic mode switching from browse mode to focus mode that is required for interaction, which can render them unusable even when they are technically perceptible.

2.2.4 WCAG Principles

The Web Content Accessibility Guidelines (WCAG) define the primary technical standard for web accessibility and are developed by the Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C) [255, 256]. WCAG provides a normative framework for evaluating whether web content is accessible to users with diverse abilities, including users of assistive technologies such as screen readers.

WCAG organizes accessibility requirements around four high-level principles, commonly referred to as POUR:

- **Perceivable:** Information and user interface components must be presented in ways that users can perceive, such as providing text alternatives for non-text content.
- **Operable:** User interface components and navigation must be operable through different input methods, including keyboard-only interaction.
- **Understandable:** Information and user interface behavior must be clear and predictable.
- **Robust:** Content must be robust enough to be interpreted reliably by a wide range of user agents, including assistive technologies.

WCAG has evolved over time to address changes in web technologies and usage patterns. WCAG 2.0 established the core structure of the guidelines [538], while WCAG 2.1 extended them with additional success criteria focused on mobile accessibility, low-vision users, and cognitive accessibility, including requirements related to touch interaction, screen orientation, and input modalities [255]. WCAG 2.2 further refines these guidelines by adding criteria that improve accessibility for users with cognitive and motor impairments, while maintaining backward compatibility with earlier versions [256].

WCAG success criteria are grouped into three conformance levels: A, AA, and AAA, with increasing strictness. Level A captures the most fundamental accessibility requirements, Level AA adds criteria that address common and significant barriers to access, and Level AAA includes additional enhancements that improve accessibility but are often difficult to satisfy across all content. In practice, Level AA of WCAG 2.1 and later versions is widely adopted as the baseline for regulatory compliance and accessibility evaluation, particularly for mobile and responsive web content.

2.3 Regular Expressions and Regex Engines

In this section, we describe the foundations of regular expressions and their execution engines. We first present the syntax and semantics of regular expressions, followed by the regex matching algorithms used by modern engines. We then introduce Regular Expression Denial of Service (ReDoS) as a vulnerability that arises from the interaction between specific regex patterns and backtracking-based engines.

2.3.1 Regexes

2.3.1.1 Kleene's regexes (K-regexes)

Regexes are a popular technology for string matching tasks [180]. Regexes are used in two basic operations: recognition ("Does this regex match this string?") and parsing ("Extract relevant substrings from this match") [248].

Regexes began with Kleene's [274] "regular" notation (*K-regexes*), which specified a language as a set of strings using core constructs of concatenation (.), disjunction (|), and unbounded repetition ($*$). These operations define whether a given string belongs to the set of strings described by the regex. Formally, a K-regex R is:

$$R \to \emptyset \;\Big|\; \epsilon \;\Big|\; \sigma \;\Big|\; R* \;\Big|\; R_1 \mid R_2 \;\Big|\; R_1 \cdot R_2$$

where the empty language is $\emptyset$, the empty string is ϵ, and the alphabet is Σ. A regex's semantics are defined by its *language*, the set of strings it describes. The language function $L : R \to 2^{\Sigma^*}$ is:

$$\begin{array}{ll} L(\phi) = \phi & L(R*) = L(R)* \\ L(\epsilon) = \{\epsilon\} & L(R_1|R_2) = L(R_1) \cup L(R_2) \\ L(\sigma) = \{\sigma\} & L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2) \end{array}$$

K-regexes can be modeled with nondeterministic or deterministic finite automata (NFA and DFA) [484, 322]. These automata are 5-tuples $\langle Q, q_0, F, \Sigma, \delta \rangle$, where Q is the set of states, q_0 is the start state, F is the set of accepting states, Σ is the input alphabet, and δ is the transition function. Each regex operator—character matching, concatenation, repetition, and disjunction—has a corresponding NFA representation, as illustrated in Figure 2.1. These NFAs can be further converted into DFAs, providing deterministic evaluation at the cost of potentially higher state complexity [403].

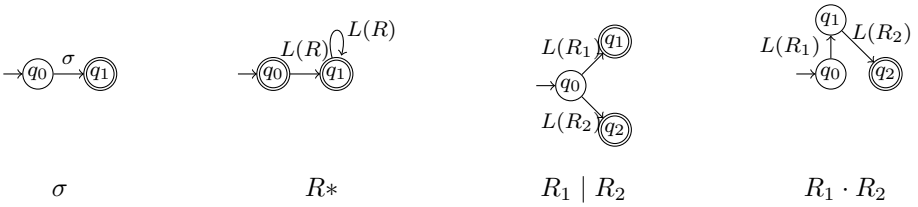


Figure 2.1: K-regex operators and NFA equivalents, following the Thompson-McNaughton-Yamada construction (484, 322).

2.3.1.2 Extended regexes (E-regexes)

Over time, to simplify the representation, the K-regexes were "extended" (*E-regexes*) with additional operators and constructs. *Syntax sugar* notations, like character ranges (e.g., `[A-Z]` for `A|B|...|Z`) and one-or-more-repetition (`R+`), are still regular and can be transformed into equivalent K-regexes. Some more complex extensions, such as atomic groups [51] and lookarounds (e.g., [90, 50, 181]), have also been shown to be regular. Many other extensions have been made [180, 235], including irregular features such as backreferences [7]. These extensions increase expressive power but exceed the boundaries of regular languages (*i.e.,* they cannot be directly modeled with finite automata), complicating theoretical analysis, implementation, and matching performance.

Table 2.1 provides a comprehensive reference of regex features and notation, summarized from the PCRE2 specification [235]. The table categorizes regex features into two groups: K-regexes and E-regexes. E-regexes are also subcategorized. Each feature is associated with its syntactic notation.

2.3.2 Regex Engines

Regexes are supported in all mainstream programming languages [180] and are part of many other systems (e.g., intrusion detection and prevention systems [424, 478], web application firewalls [19], and logging tools [80, 31]). These systems all implement or integrate a component called a *regex engine* that provides support for the regex tasks of recognition and parsing. As summarized by Davis [131], typical regex engine implementations follow one of two algorithms. The first is a depth-first, backtracking search based on a library by Spencer [455]. The second is a breadth-first, lockstep search proposed by Thompson [484]. The characteristics and trade-offs of these two engine styles are summarized in Table 2.2. For the simple case of K-regexes, both can be modeled as NFA simulations.[3] These approaches differ in their handling of *ambiguity*, where multiple transitions in the NFA are possible [15].

2.3.2.1 Spencer's NFA-Based Backtracking Algorithm

Spencer's backtracking algorithm is the standard implementation used in most modern programming languages, including JavaScript, Python, and Java [128]. The algorithm simulates a nondeterministic finite automaton (NFA) using a depth-first search (DFS) strategy. When the engine encounters ambiguity, where the NFA permits multiple transitions for the same input symbol, it resolves this nondeterminism by selecting one transition to explore immediately while storing alternative transitions on a *backtracking stack*. If the chosen path fails to reach an accepting state, the engine *backtracks* by restoring a previously saved state from the stack and exploring an alternative path. This process continues until a match is found or all possible execution paths are exhausted.

[3] Implementations of Thompson's algorithm typically have a direct mapping to an NFA simulation, while implementations of Spencer's algorithm may be implemented as recursive descent parsers. Here, we treat K-regexes, where both amount to NFA simulations.

Table 2.1: Full set of features and notation of regexes, compiled from (235). R denotes a sub-pattern.

Abbreviation	Feature	Notation
	K-regexes	
CAT	X followed by Y	R_1R_2
KLE	Zero-or-more repetition	$R*$
OR	Logical OR	$R_1\|R_2$
	E-regexes	
	Capture Groups	
CG	Capture group	(R)
NCG	Non-capture group	$(?:R)$
PNG	Named capture group	(?<name>R)
	Quantifiers	
ADD	One-or-more repetition	$R+$
QST	Zero-or-one repetition	$R?$
DBB	Range repetition	$R\{m,n\}$
LWB	At-least-m repetition	$R\{m,\}$
SNG	Exactly-n repetition	$R\{n\}$
	Character Classes	
CCC	Custom character class	[aeiou]
RNG	Character range	[a − z]
NCCC	Negated class	[ˆaeiou]
ANY	Any character	.
WSP	Whitespace	\s
DEC	Numeric	\d
WRD	Word	\w
NWSP	Non-whitespace	\S
NDEC	Non-numeric	\D
NWRD	Non-word	\W
VWSP	Vertical space	\v
	Zero-width Assertions	
STR	Start-of-string/line	$\hat{}R$, \A$R$
END	End-of-string/line	R\$, R \ Z
WNW	Word/non-word boundary	\b
NWNW	Negated WNW boundary	\B
PLA	Positive lookahead	$(?=R)$
NLA	Negative look-ahead	$(?!R)$
PLB	Positive lookbehind	$(?<=R)$
NLB	Negative look-behind	$(?<!R)$
	Backreferences	
BKR	Numeric backreference	$(R)\ldots$ \ 1
BKRN	Named backreference	(?<name>R)$\ldots$ \ k<name>
	Backtracking Controls	
LZY	Non-greedy repetition	$R*?$, $R+?$, $R\{m,n\}?$
ATM	Atomic group	$(?>R)$
POS	Possessive quantifier	$R++$, $R*+$

Table 2.2: Tradeoffs of Spencer and Thompson algorithms for regex engines (111, 131). Thompson's algorithm uses an efficient algorithm over an automata representation to obtain linear time complexity (in input string length), but this formal approach reduces its expressiveness. Spencer's algorithm gains broader E-regex support at the cost of exponential match times.

Algorithm	**Strategy**	**Match time**	**Expressiveness**
Thompson's	BFS NFA sim.	Linear match	Limited E-regex support
Spencer's	Backtracking	Exponential match	Easy E-regex support

Complexity and Catastrophic Backtracking Constructing the NFA from a regular expression typically takes $\mathcal{O}(|R|^2)$ time, where $|R|$ denotes the length of the regex pattern. The simulation phase, however, can exhibit highly variable performance. For unambiguous patterns, matching proceeds in near-linear time, on the order of $\mathcal{O}(|Q|\cdot|w|)$, where $|Q|$ is the number of NFA states and $|w|$ is the input length.

In contrast, for *pathological* NFAs with extensive ambiguity, the backtracking mechanism may explore an exponential number of execution paths. In such cases, the number of explored paths grows as $\mathcal{O}(b^{|w|})$, where $b > 1$ reflects the branching factor induced by ambiguous subexpressions. This exponential blow-up is known as *catastrophic backtracking* and constitutes the fundamental algorithmic weakness exploited by Regular Expression Denial of Service (ReDoS) attacks (§2.3.3).

2.3.2.2 Thompson's NFA-Based Lockstep Algorithm

Thompson's algorithm evaluates regular expressions by simulating the NFA in a breadth-first manner, commonly referred to as a *lockstep* simulation. Rather than exploring one execution path at a time, the algorithm tracks all possible active states simultaneously.

Let $\Phi_{\text{cur}} \subseteq Q$ denote the set of NFA states currently active. For each input symbol, the algorithm computes the next set of active states Φ_{next} by applying the transition function δ to every state in Φ_{cur}. If multiple transitions lead to the same state, that state is added to Φ_{next} only once. This deduplication step avoids redundant exploration and prevents the exponential behavior observed in backtracking engines.

Complexity and Limitations Thompson's algorithm guarantees polynomial-time matching. In practice, the matching time is $\mathcal{O}(|Q| \cdot |w|)$, as each input symbol advances the set of active states with work proportional to the automaton size. Accounting for transition bookkeeping and state management, the worst-case complexity remains polynomial and linear in the input length $|w|$.

This efficiency comes at the cost of expressiveness. Because the lockstep approach merges all concurrent execution paths into a single set of active states, it discards the path-specific history required to support extended regex features such as backreferences and complex lookarounds [131].

(a) NFA for $a*a*$ (b) NFA for $(aa*)*$

Figure 2.2: Example regexes that, in Spencer's algorithm, exhibit time complexity that is either: (a) polynomial or (b) exponential; in the length of the input string $\boldsymbol{w = aa...ab}$.

2.3.2.3 Performance Trade-offs and Worst-Case Behavior

In typical usage scenarios, for most regexes, inputs, and application contexts, both approaches run quickly enough and do not form a performance bottleneck for the surrounding application. However, while Thompson's algorithm guarantees match times that are linear in the input length, Spencer's algorithm exhibits worst-case polynomial or exponential match times on certain regex–input combinations.

Figure 2.2 gives two examples where Spencer's algorithm exhibits polynomial and exponential time complexity, respectively. In each case, on an input $w = aa...ab$, the regex engine encounters ambiguity when processing an "a" symbol. At the ultimate mismatch (caused by the final "b"), the backtracking will take $|w|^2$ time for example (a) and $2^{|w|}$ time for example (b). This worst-case behavior is colloquially termed *catastrophic backtracking*. Note that the actual cost of the match depends on the length of the input; the term *pumping* refers to the repetition of the problematic substring of the input (each "a" in this example).

Historically, regex engine maintainers accepted this worst-case behavior as the cost of supporting extended regular expression (E-regex) features. Such features are relatively easy to implement in a Spencer-style backtracking engine, which often takes the form of a recursive descent parser [129]. New E-regex constructs can be added by extending the set of parse operators, or equivalently by extending the meaning of transitions in the underlying automaton [131]. For example, zero-width assertions such as `\b` can be implemented using specialized ϵ-transitions that depend on contextual tests. Backreferences introduce transitions that validate previously matched group content while advancing the input position accordingly. Lookaround assertions similarly rely on recursive evaluation of sub-patterns within the backtracking framework [129]. These implementations are feasible because Spencer's algorithm processes execution paths sequentially, simplifying the management of path-specific state.

2.3.3 Regular Expression Denial of Service (ReDoS)

Regular Expression Denial of Service (ReDoS) is an algorithmic complexity attack that exploits worst-case behavior in regular expression matching engines [113, 209, 426]. By crafting inputs that trigger excessive computation, an attacker can cause regex evaluation to consume disproportionate CPU time, potentially rendering services unresponsive. The most severe and widely studied form of ReDoS arises in Spencer-style backtracking engines [129]. As discussed in §2.3.2, these engines resolve ambiguity by exploring alternative execution paths sequentially. For certain regex patterns, this

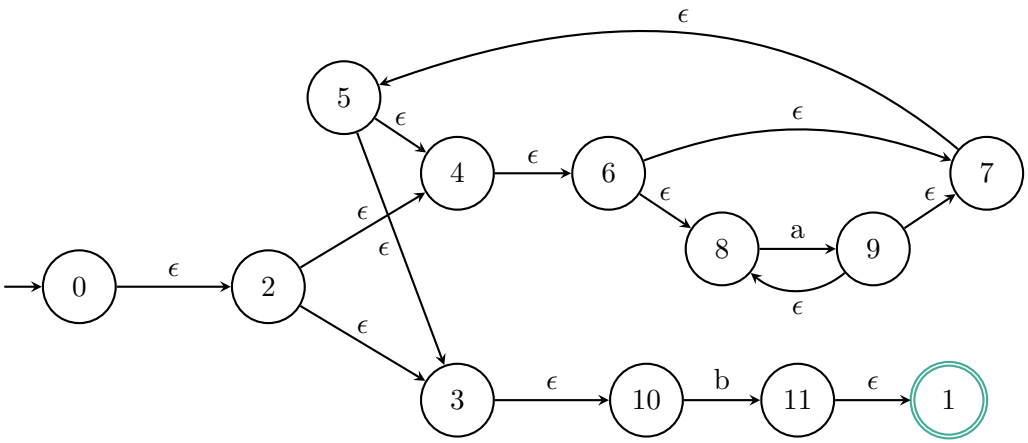


Figure 2.3: Automaton for the regular expression $(a^*)^*b$. 0 is the starting state, and 1 is the accepting state.

strategy can lead to exponential-time behavior when the engine is forced to exhaust many equivalent matching paths before failing.

As a simple example, consider the regex $(a^*)^*b$ applied to the input `aaa`. Using the automaton shown in Figure 2.3, the engine repeatedly decides how to distribute each `a` between the inner a^* (States 6, 8, and 9, with exit via 7) and successive iterations of the outer $(\cdot)^*$ (controlled by States 5 and 7). When the input is exhausted and the required trailing `b` cannot be matched, the backtracking engine revisits these earlier choices, exploring all possible partitions of the input across the nested repetitions. The number of such alternatives grows exponentially with the input length, resulting in catastrophic backtracking.

2.4 Graph Representation Learning

This section introduces the theoretical foundations of deep learning on graphs, which underpin the program analysis tasks presented in Chapter 9. We first define the general framework of Graph Neural Networks (GNNs), focusing on the message-passing paradigm and permutation invariance. We then describe the specific architecture used in this dissertation, the Gated Graph Convolutional Network (GatedGCN), and conclude with the link prediction formulation used to infer missing relationships in software graphs.

2.4.1 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of deep learning models designed to operate on graph-structured data. Unlike traditional neural networks (e.g., CNNs or RNNs) that assume grid-like or sequential inputs, GNNs operate on non-Euclidean domains. They leverage the connectivity structure of graphs to learn low-dimensional vector representations (embeddings) for nodes, edges, or entire graphs. Figure 2.4 illustrates a typical GNN architecture, where node representations are iteratively updated through

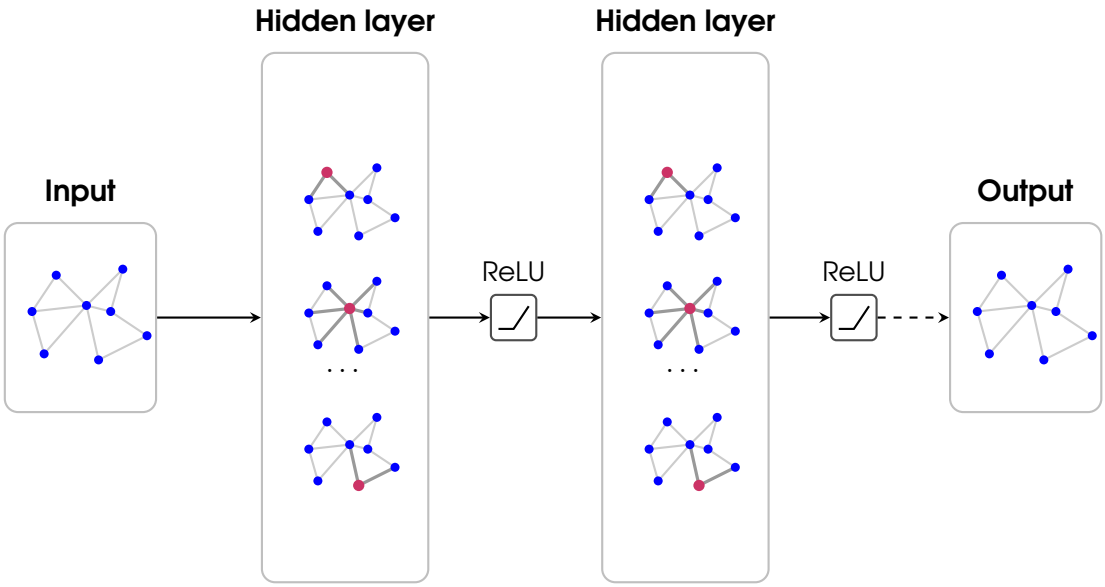


Figure 2.4: Illustration of a multi-layer Graph Neural Network. Each layer performs message passing over the same underlying graph structure. Highlighted nodes and edges indicate the local neighborhood involved in aggregation for a given target node. Nonlinear transformations (ReLU) are applied between layers, and the final output is produced after multiple rounds of neighborhood aggregation.

multiple layers of neighborhood aggregation over a shared graph structure.

A defining characteristic of GNNs is their permutation symmetry, ensuring that the model's output depends on the graph topology rather than the arbitrary indexing of its nodes. This property is achieved through permutation-invariant aggregation functions (such as sum, mean, or max) that treat neighboring nodes as an unordered set [185, 545]. As a result, GNN layers are permutation equivariant: a permutation of the input node ordering leads to a corresponding permutation of the output representations. For graph-level tasks, permutation invariance can be obtained by applying global pooling operations, ensuring that the final output remains unchanged under any node reordering [77].

Message Passing and Receptive Fields Most modern GNNs follow the Message Passing Neural Network (MPNN) framework [185]. Each node $v \in V$ is associated with a feature vector $\mathbf{h}_v^{(0)}$. The model updates these representations iteratively. A single layer k consists of two distinct phases: *aggregation* of messages from the local neighborhood $\mathcal{N}(v)$, and *update* of the node's internal state:

$$\mathbf{m}_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{\phi^{(k)}\left(\mathbf{h}_v^{(k)}, \mathbf{h}_u^{(k)}, \mathbf{e}_{u,v}\right) : u \in \mathcal{N}(v)\right\}\right), \tag{2.1}$$

$$\mathbf{h}_v^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_v^{(k)}, \mathbf{m}_v^{(k)}\right), \tag{2.2}$$

where ϕ is a message function (often a Multi-Layer Perceptron) and AGGREGATE is a permutation-invariant function (e.g., sum, mean, or max). By stacking K layers, a

node accumulates information from its K-hop neighborhood. This effectively defines the node's *receptive field*, allowing local features to propagate and capture broader structural context.

2.4.2 Gated Graph Convolutional Networks (GatedGCN)

While standard Graph Convolutional Networks (GCNs) aggregate neighborhood information using fixed, isotropic weighting schemes (e.g., degree-based normalization), they often fail to capture the varying importance of different edges. *Anisotropic* graph models address this limitation by learning edge-specific weights. Gated Graph Convolutional Networks (GatedGCNs) [73] are a prominent example of this class, integrating learnable edge gates into the message-passing framework to modulate information flow.

Edge Gating Mechanism Gated Graph Convolutional Networks model edges through learnable gating mechanisms that explicitly control the flow of information during message passing. Each edge is assigned a latent representation that is updated jointly with node representations and is used to modulate the contribution of neighboring nodes.

At layer ℓ, each node i is associated with a hidden representation $\mathbf{h}_i^\ell$, and each directed edge (i, j) is associated with a hidden edge state $\hat{\mathbf{e}}_{ij}^\ell$. Edge states are updated by combining the current edge representation with contextual information from the incident nodes:

$$\hat{\mathbf{e}}_{ij}^\ell = \hat{\mathbf{e}}_{ij}^{\ell-1} + \mathrm{ReLU}\Big(\mathrm{BN}\Big(\mathbf{A}^\ell \mathbf{h}_i^{\ell-1} + \mathbf{B}^\ell \mathbf{h}_j^{\ell-1} + \mathbf{C}^\ell \hat{\mathbf{e}}_{ij}^{\ell-1}\Big)\Big), \tag{2.3}$$

where $\mathbf{A}^\ell$, $\mathbf{B}^\ell$, and $\mathbf{C}^\ell$ are learnable parameters and BN denotes batch normalization.

The resulting edge representations are converted into normalized gating coefficients across the neighborhood $\mathcal{N}_i$:

$$\mathbf{e}_{ij}^\ell = \frac{\sigma(\hat{\mathbf{e}}_{ij}^\ell)}{\sum_{j' \in \mathcal{N}_i} \sigma(\hat{\mathbf{e}}_{ij'}^\ell) + \varepsilon}, \tag{2.4}$$

where σ is the sigmoid function and ε is a small constant for numerical stability.

Node representations are then updated using gated message aggregation:

$$\mathbf{h}_i^{\ell+1} = \mathbf{h}_i^\ell + \mathrm{ReLU}\left(\mathrm{BN}\left(\mathbf{U}^\ell \mathbf{h}_i^\ell + \sum_{j \in \mathcal{N}_i} \mathbf{e}_{ij}^\ell \odot \left(\mathbf{V}^\ell \mathbf{h}_j^\ell\right)\right)\right), \tag{2.5}$$

where $\mathbf{U}^\ell$ and $\mathbf{V}^\ell$ are learnable weight matrices and $\odot$ denotes the Hadamard product. This formulation enables anisotropic, context-sensitive weighting of neighboring nodes during message passing.

Handling Heterogeneous Features In program analysis, edges often encode distinct semantic relationships, such as *ControlFlow* or *DataFlow*. GatedGCN naturally accommodates such heterogeneity by initializing edge features $\hat{\mathbf{e}}_{ij}^0$ with encodings of edge types. The learned gating mechanism can then prioritize specific relationships during message passing, allowing the model to focus on semantically relevant paths in the program graph.

Link Prediction and Negative Sampling Beyond node-level tasks, GNNs are often used for link prediction to infer missing edges in a graph. In this setting, the GNN encoder produces node embeddings, and a decoder assigns a score $s(u, v)$ that reflects the likelihood of an edge between nodes u and v:

$$s(u, v) = f\left(\mathbf{h}_u^{(K)}, \mathbf{h}_v^{(K)}\right), \tag{2.6}$$

where f is a scoring function such as a dot product or a bilinear form ($\mathbf{h}_u^\top \mathbf{W} \mathbf{h}_v$). Training typically relies on *negative sampling*: the model is optimized to assign high scores to observed edges $(u, v) \in E$ while assigning low scores to sampled non-edges $(u, v') \notin E$, using a loss such as binary cross-entropy. This formulation is used in Chapter 9 to predict missing call edges in program graphs

Part I

Digital Disparity Across Economic Contexts

3

Web Ecosystems Across Economic Contexts

In this chapter, we empirically investigate web development practices across different economic contexts to address demographic diversity of the modern web (see challenge C_1 in the introduction). While internet usage is growing globally, the quality of the user experience varies significantly due to the tension between historical inertia and the potential for developing countries to leapfrog legacy technologies. To understand this dynamic, we analyze 200,000 websites from 10 developed and 10 developing nations to examine how differing levels of connectivity and infrastructure shape the digital ecosystem. Our findings reveal a complex reality in which webpages in developing regions are generally smaller and simpler. This structural simplicity acts as an adaptation that improves performance over slower networks, but it often masks poor optimization practices such as the use of inefficient image formats and unnecessary JavaScript or CSS code. Furthermore, our security assessment reveals contrasting patterns: while developing regions lag in HTTPS adoption, websites in developed regions more frequently include vulnerable third-party libraries. Interestingly, these websites also include a higher number of third-party trackers, despite being subject to stronger privacy regulations.

This chapter shares material with the corresponding publication [P1].

3.1 Motivation

According to the most recent 2025 data from the International Telecommunication Union [257] and the World Bank [535], 74% of the global population uses the internet, with a 3.3% growth in 2025 alone. However, there are large discrepancies in internet adoption based on the economic situation of a country, *i.e.,* while 93% of the developed world is online, only 60% of the developing world is [1]. Moreover, due to the limited availability of fixed broadband services and the prohibitive costs associated with them, users from developing countries mostly rely on mobile internet [464, 218]. This measurable inequality – also called digital disconnect [241] or digital divide [311] – is a missed opportunity for economic growth in developing countries [36].

It is commonly believed that digital inequality also extends to the web. For example, a recent work by Ruth et al. [430] finds that most websites visited in a country are specific to that country, with only few popular global websites. Moreover, the World Bank [36] notes that developed countries tend to host more web-based services. On the contrary, the lack of broadband [216, 36] and modern devices in developing countries might lead to web development optimizing for low-bandwidth conditions, e.g., with more simplistic designs and supported features. Still, websites in these regions are often criticized for poor optimization, causing slow load times and high data usage [87, 549]. However, developing countries can leapfrog [74, 167] and adopt the latest technologies, such as WebAssembly, without the ossification observed for older webpages [278]. Also, users, developers, and regulators in developing countries might be insufficiently concerned with security and privacy.

To the best of our knowledge, no large-scale empirical study has systematically analyzed web development practices across different economic contexts [244]. Most existing research focus on macro-level indicators such as internet penetration rates,

[1] https://en.wikipedia.org/wiki/Global_digital_divide

smartphone adoption, or general technological infrastructure [495, 363]. To address this research gap, we conduct a large-scale analysis comparing webpages from both developed and developing countries. Our study involves identifying and crawling 200,000 popular webpages, *i.e.,* 10,000 from the top 10 most populated developed and developing nations, using tools like Google Lighthouse for performance audits and Puppeteer to capture detailed network data. This analysis considers multiple web metrics, which we use to explore websites' composition and performance, security, privacy, technology adoption, and accessibility compliance as summarized below:

Webpage Size and Complexity: We assessed webpage size, HTTP requests, and multimedia usage to evaluate load times and data consumption. Websites in developing countries are generally smaller and simpler, thus more suitable for slower networks. Inefficiencies in image formats and JavaScript persist, highlighting opportunities for further optimization despite their reduced size.

Performance Optimization: We analyzed websites' performance by assessing the use of optimized image formats or responsive images or the inclusion of unnecessary JavaScript and CSS files. Despite smaller sizes, websites in developing countries waste more bandwidth due to poorly optimized images and unused code, lacking advanced optimizations, which might hinder user experience.

Security Measures: We evaluated security practices by examining HTTPS usage, outdated libraries, and CSP implementation. Developed countries have adopted HTTPS more widely, but surprisingly, they also contain a higher number of severe security vulnerabilities. Both groups show a widespread lack of CSP policies, leaving websites vulnerable to threats like cross-site scripting.

Privacy Practices: We analyzed tracking scripts and third-party cookies to assess privacy concerns. Websites in developed countries use more trackers and cookies, likely due to data-driven business models. Despite stricter privacy regulations, tracking remains widespread via consent mechanisms. Developing countries have fewer trackers, but weaker privacy regulations remain a concern [488].

Technology Adoption: We examined the adoption of modern web APIs and protocols to assess if developing countries leapfrog older technologies. Surprisingly, some developing countries adopt HTTP/2 and HTTP/3 at rates comparable to or higher than developed ones, likely due to fewer legacy systems. Both groups, however, show similar web API adoption.

Accessibility Compliance: We assessed websites for compliance with accessibility standards, including adherence to the Web Content Accessibility Guidelines (WCAG). Our analysis covers features such as alternative text for images, keyboard navigation support, and other mechanisms that enhance usability for individuals with disabilities [255, 288].

3.2 Related Work

Multiple studies have quantified cellular and broadband connectivity in developing regions, performing measurements, and proposing solutions to improve connectivity. Hassan et al. [231, 232, 230] investigated cellular connectivity in rural countries and

how solutions such as Software Defined Networking can provide affordable connectivity. Similarly, Marshini et al. [93] explored mobile broadband connectivity in South Africa, while Zaki et al. [549] investigated connectivity challenges in Ghana. Fanou et al. [155] further highlighted the disparities in web content infrastructure across Africa. Research by Koradia et al. [281], Sharma et al. [444], and Bischof et al. [61] focused on cellular network issues in India and Cuba.

Google's AMP [199] and Facebook Lite [324] are product designed to to address web complexity in developing regions. AMP provides a framework for faster web page construction, while Facebook Lite is optimized for low-end phones and slow networks. Google also launched Web Light [201], which transcoded webpages to lighter versions for slow connections. Some research papers also developed concrete web optimizations: Lite-Web [87], for example, aims to reduce web complexity through tools such as SlimWeb [88], JSCleaner [89], and Muzeel [286].

Web performance in developing regions are worsened by the diffusion of low-end devices. Naseer et al. [349, 348] noted that JavaScript execution often triggers memory constraints, while rising web complexity and JavaScript-based tracking inflate costs for users. Amjad et al. [21] showed that selective JavaScript blocking can help, though with challenges. Additional research has tackled web accessibility in these areas. Raza et al. [413] proposed enhanced caching to compensate for limited cloud infrastructure, while Ahmad et al. [5] advocated for a new content ecosystem tailored to local needs, essential for users in resource-constrained environments.

Finally, Habib et al. [226] introduced a fairness metric to redesign the web with a focus on affordability and inclusivity. Unlike [226, 225], we analyze and explore 200k popular web pages, from the most populated developed and developing regions, in terms of their performance, security, privacy, and technology adoption. Additionally, in comparison to [430], which focuses on web traffic distribution and the most visited categories, we delve into the finer details of these pages, comparing factors such as image formats, unnecessary JavaScript usage, and security measures.

3.3 Methodology

Our methodology combines Google Chrome User Experience (CrUX) dataset [139] with the International Monetary Fund (IMF)'s classification, and with web data (webpage composition, protocol usage, etc.) we crawled using Google Lighthouse and Puppeteer. IMF classifies countries as "Advanced Economies" and "Emerging and Developing Economies" based on per capita income, export diversification, and financial system integration. CrUX data provides real-world web performance metrics like page load times and responsiveness, offering insights into website performance across regions.

3.3.1 Country Selection Criteria

The countries in this study are selected to ensure a balanced representation of both developing and developed economies, following the classifications provided by the IMF. Specifically, we select a sample of 10 developing and 10 developed countries for a comparative analysis of their web ecosystems. The selection process was guided by two

primary criteria: population size and the availability of 10,000 websites per country that met our inclusion criteria (§3.3.2) within the CrUX database [139].

For developing countries, we identified the ten most populous nations from a cohort of 152 economies classified as "developing" by the IMF [252]. Population size is chosen as the primary criterion to ensure that the selected countries represent a significant portion of the global population within the developing world. Ethiopia, although the 10^{th} most populous developing country [541], was excluded because we could not identify at least 10,000 websites that met our criteria (§3.3.2). To complete our sample, we included the Philippines in place of Ethiopia. The ten developing countries selected, in order of population, are: China, India, Indonesia, Pakistan, Brazil, Nigeria, Bangladesh, Russia, Mexico, and the Philippines.

For the developed countries, we focused on the most populous nations classified as "Advanced Economies" by the IMF. This selection ensured that major global economies were adequately represented in our dataset. The ten developed countries, ordered by population, are: the United States, Japan, Germany, France, the United Kingdom, Italy, South Korea, Spain, Canada, and Australia.

Together, the developing countries in our dataset account for a combined population of 4.3 billion people, representing 54.33% of the global population [37]. In comparison, the developed countries in our sample account for 906.01 million people, or 11.3% of the world's population. By using population size and CrUX data availability as selection criteria, our study captures a significant portion of the world's population across both economic classifications, providing a robust foundation for comparing the web ecosystems of developing and developed nations.

3.3.2 Website Selection

To investigate the digital landscape in each country, we retrieve data on 10,000 websites per country from the CrUX dataset. A website is categorized under a particular country if it either has a country code top-level domain (ccTLD) for that country (e.g.,.de for Germany) or if its Whois [119] data indicates a physical registration address in that country. This criterion was used to mitigate the over-representation of globally dominant websites, which could skew the analysis of local web ecosystems. We utilized the `python-whois` package [382] and the `who.is` API [530] to retrieve registration information for the websites. We start from the 10,000 most popular websites for each country – as indicated by the CrUX list – and then apply our selection criteria to filter the relevant websites for analysis. This approach ensures a uniform number of websites for each country, and a comparable distribution of website rankings. In most cases, our 10,000 websites were within the top 50,000 of that country. However, for Bangladesh, Pakistan, Nigeria, and the Philippines, we had to include websites that were less popular to meet our criteria and target sample size. Figure 3.1 displays the distribution of website rankings across different countries in our dataset.

3.3.3 Crawler Configuration

We crawl the selected websites using Google Lighthouse [203] for performance audits, while simultaneously employing Puppeteer [399] to intercept and log network requests

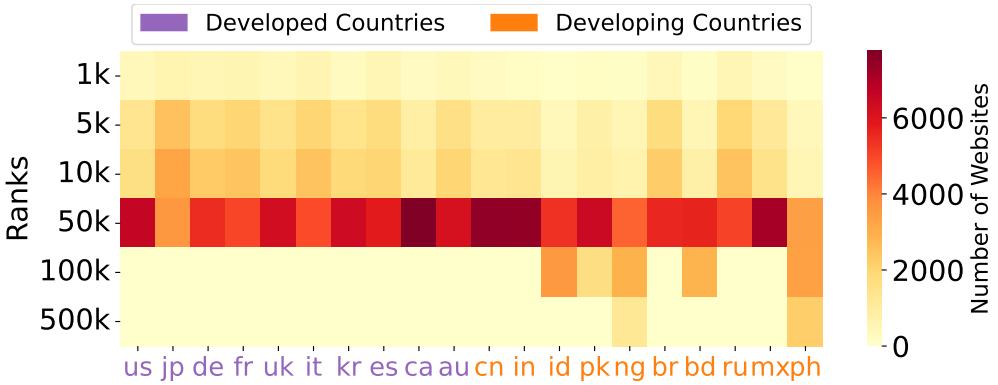


Figure 3.1: Distribution of website rankings in our dataset per country. The figure shows that in most cases, the 10,000 websites per country are within the top 50,000 websites of that country. For Bangladesh, Pakistan, Nigeria, and the Philippines, less popular websites were included to meet the target sample size.

and responses, capturing detailed data such as request headers, responses, and metadata. In the presence of unreachable websites, e.g., due to temporary outages, Lighthouse timeouts, certificate errors, or other technical problems, we supplemented the list with lower-ranked (less popular) websites. For the Lighthouse audits, we emulated an iPhone X mobile environment, with a screen resolution of 375x812 pixels and a device scale factor of 3, ensuring that the mobile version of each page was loaded. This configuration was chosen to reflect common mobile browsing conditions and to avoid bias due to default client configuration as observed in [265]. We run our crawlers across three server machines hosted at two university campuses (Germany and UAE). This selection was motivated by the minimal crawling bias observed for university IP addresses when compared to, for instance, cloud IP addresses [265].

3.3.4 Library Detection

To analyze the usage of JavaScript libraries across the selected websites, we employ multiple methods for library detection. We use the retirejs Python library [153], custom regular expressions to extract library information from request URLs, and data from Google Lighthouse audits. This approach helps capture a broader range of library usage across websites.

3.3.5 Limitations

Our crawler operates from Germany and UAE for scalability, though Germany's inclusion as a test country may introduce bias. Ideally, crawlers would run within each studied country, but logistical constraints made this unfeasible. VPNs proved unreliable, as VPN detection could skew results, and many rely on cloud servers that lack tracking domains and JavaScript libraries [265]. Since crawlers don't operate within most studied countries or simulate realistic mobile networks, they cannot fully capture Web performance tied to user experience. To mitigate this, we use CrUX data, offering real-world metrics from actual devices and networks.

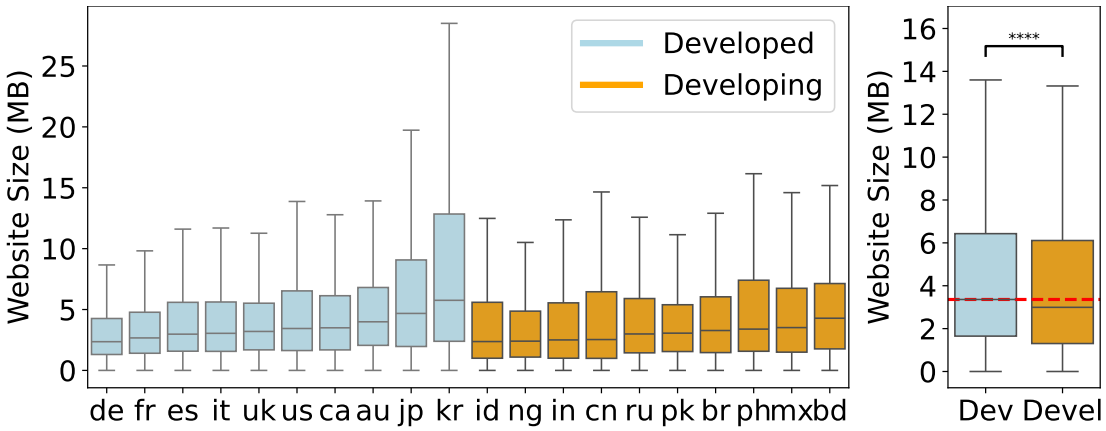


Figure 3.2: Website size comparison between developed and developing countries.

Google Lighthouse also has limitations. By default, it doesn't handle cookie banners, which may prevent full webpage rendering. While solutions exist [204], their reliability is untested. Lighthouse also cannot handle user authentication, affecting performance and security metrics [410]. Additionally, retirejs may miss libraries, especially in bundled files, limiting library detection accuracy [404].

3.4 Size and Complexity

3.4.1 Static Footprint

We measure a webpage *size* as the total number of bytes required to load all its content. Figure 3.2 shows that the median website size in developed countries is 3.35 MB, *i.e.,* slightly larger than the 2.99 MB median size in developing countries, both of which exceed the global median of 2.6 MB as per the HTTP archive [27]. Similarly, websites in developed countries require a median of 108 requests to load (Figure 3.3), compared to 89 requests in developing countries – while the global median is 70 requests. These differences between developed and developing countries are statistically significant($p < 0.001$), highlighting the larger footprints of websites in developed countries. South Korea stands out with exceptionally large websites – median size of 5 MB and upper whisker close to 30 MB – due to higher multimedia content and interactive features supported by the country's fast internet speeds [253].

This result contrasts with earlier findings that focused on public service websites [225], where websites in developing countries were significantly larger and less optimized. In our broader comparison of general websites, however, the opposite is true: websites in developing countries have smaller footprints and fewer resource requests, indicating they are more suitable for slower networks. Despite this, both developed and developing countries exceed the global median in terms of website size and number of requests, suggesting further optimization could reduce data costs, especially for users with limited access to affordable internet services.

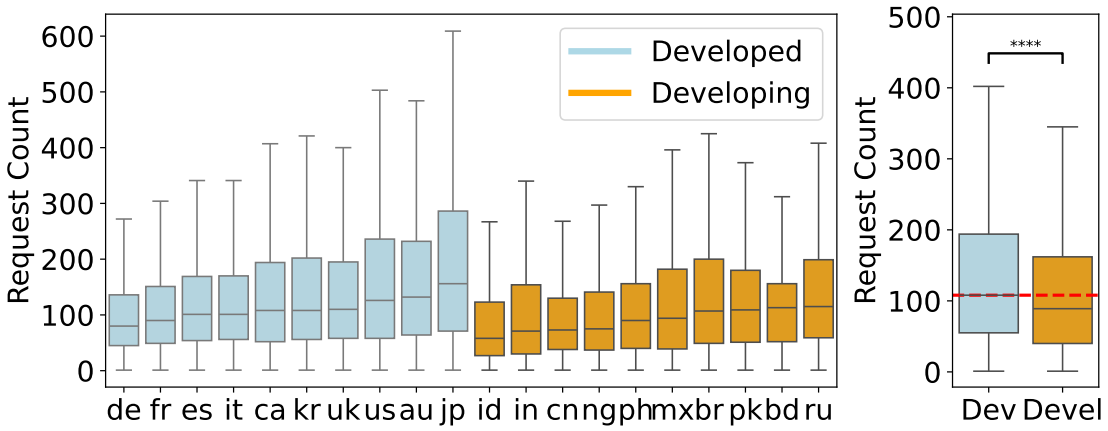


Figure 3.3: Request count comparison between developed and developing countries.

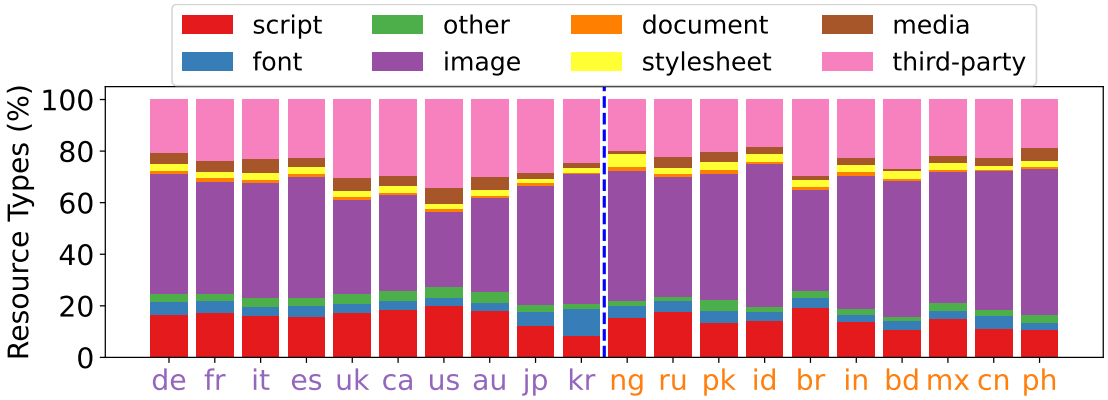


Figure 3.4: Bandwidth allocation for different resource types.

3.4.2 Breakdown by Resource Type

To understand which webpage characteristics contribute to the larger sizes of websites in developed countries, we analyze the sizes of different resources across pages. We distinguish between HTML documents, images, JavaScript, and third-party content, *i.e.,* any resource imported from a third party domain [559]. Figure 3.4 shows a clear contrast in how websites from these two regions allocate bandwidth across webpage components, particularly images. Websites in developing countries dedicate a much larger share of their page size to images, ranging from 48% to 57%. For instance, websites in Indonesia allocate 55.3% of their page size to images, compared to websites in developed countries like the US (28.8%), the UK (36.3%), and Canada (37.2%), where image bandwidth usage is $< 40\%$.

Figure 3.5 further focuses on image data usage, revealing that websites in developing countries devote a larger proportion of their bandwidth to images, even though they make fewer image requests (19 on average) than websites in developed countries (24 on average). Notably, websites in developing regions allocate more than 50% of their bandwidth to images, averaging 822.75 KB, compared to websites in developed regions, which allocate less than 40% and average 839.62 KB. This indicates that the higher

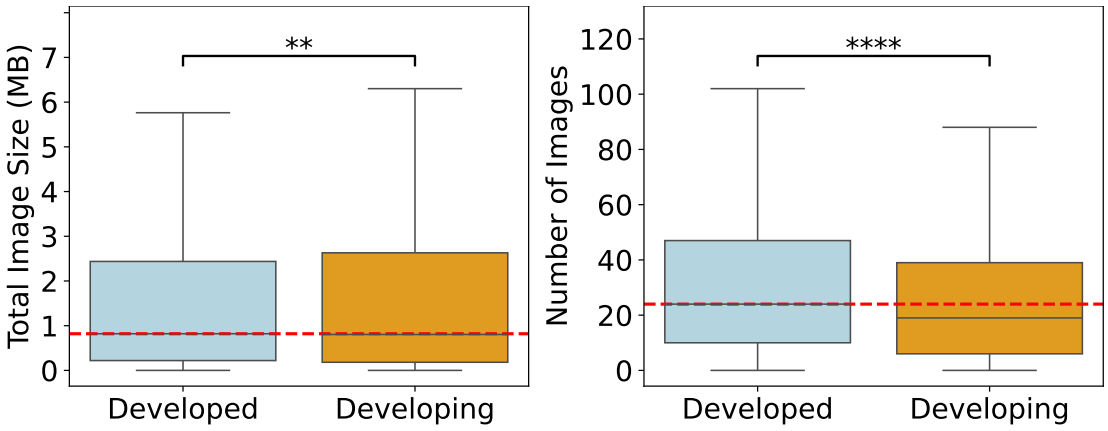


Figure 3.5: The sum of all image sizes and the number of image requests across developed and developing regions.

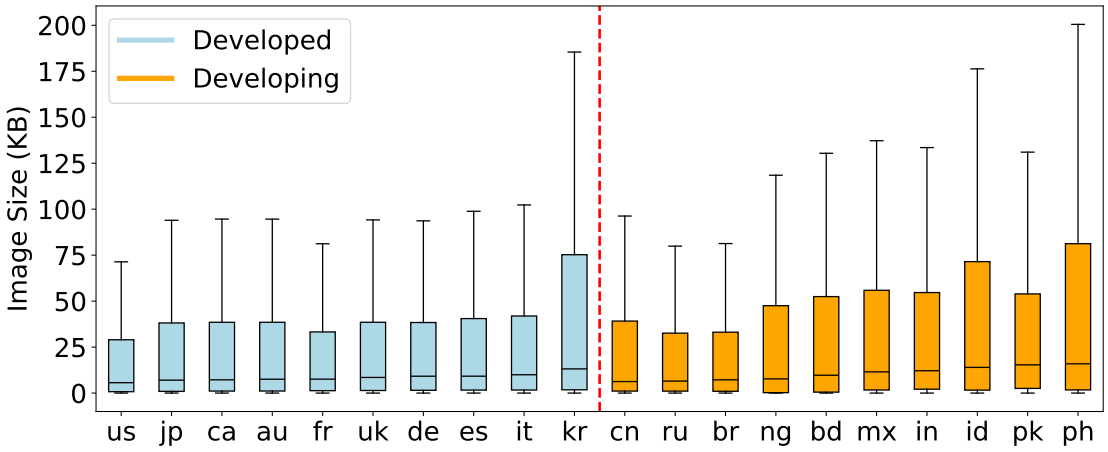
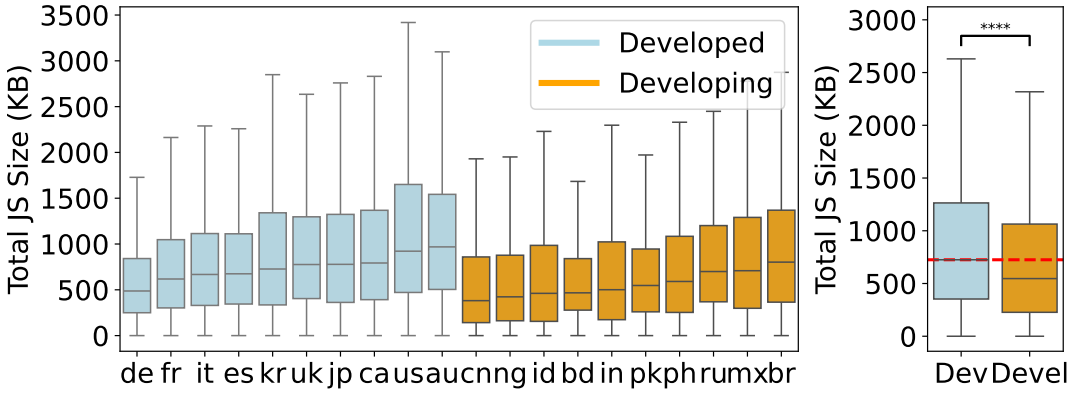


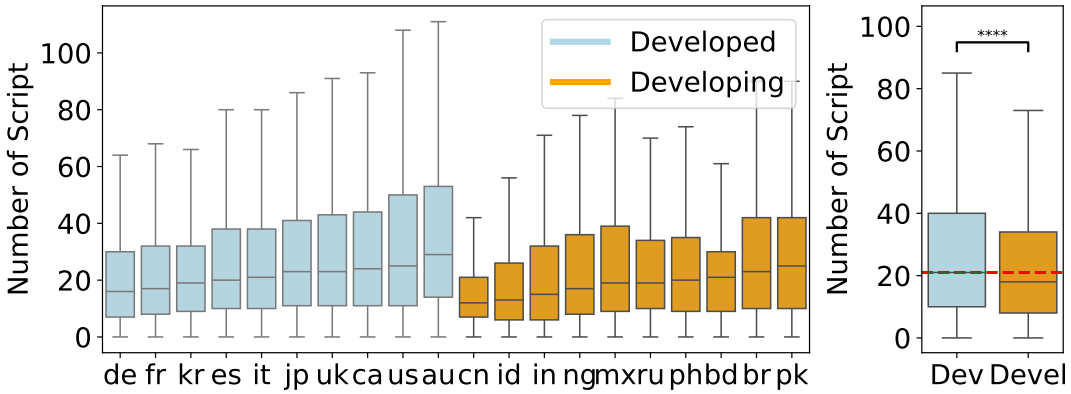
Figure 3.6: Distribution of image sizes across countries.

share of image-related bandwidth in developing countries stems from larger image files rather than a larger number of image requests.

By analyzing the distribution of image sizes, we confirm that websites in developing countries use significantly larger image files. Figure 3.6 illustrates these variations across different countries. Specifically, 15% of images on websites in developing countries exceed 100 KB, compared to only 12% in developed countries. This reinforces the observation that the bandwidth burden in developing regions is driven by unoptimized, heavy images rather than the number of assets hosted. Consequently, this practice increases the overall website size and bandwidth consumption, leading to potential negative effects for users in regions often characterized by slower internet speeds and limited data plans.



(a) Total JS Size Across Countries



(b) Number of JS Requests Across Countries

Figure 3.7: Comparison of total JS size (a) and number of JS requests (b) between developed and developing countries.

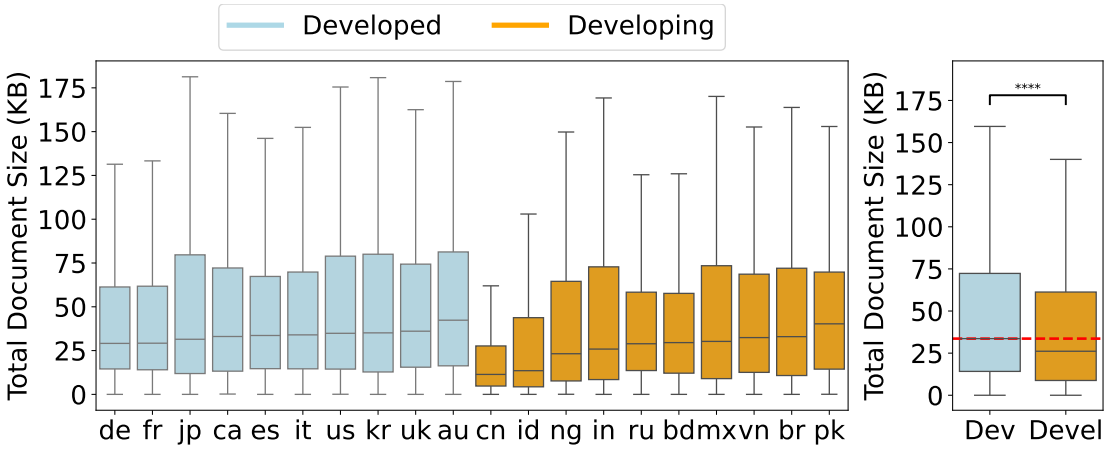


Figure 3.8: Distribution of HTML document sizes across developed and developing regions.

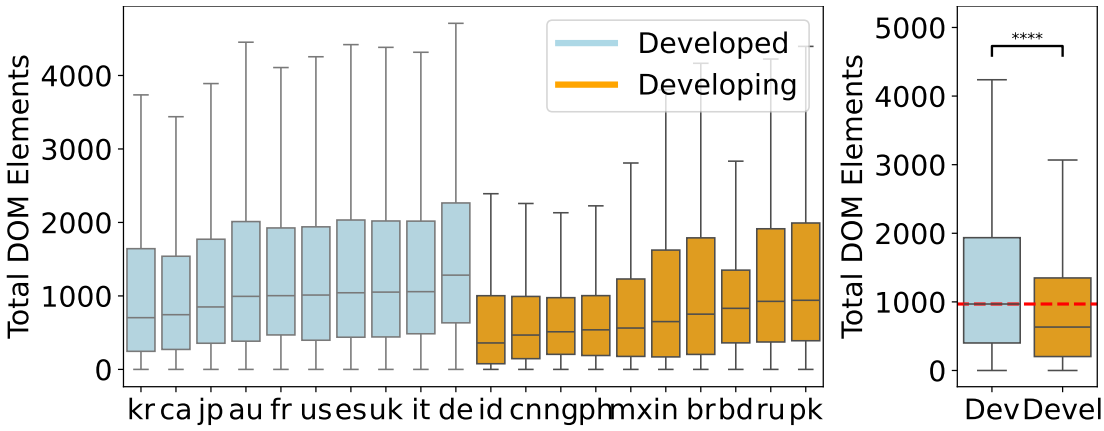


Figure 3.9: Total number of DOM elements across countries.

3.4.3 Dynamic Footprint

We here focus on JavaScript (JS), the dominant factor in a website's performance [89], with Chrome spending over 30% of its processing time on it [372]. Figure 3.7a shows the difference in the total size of JS usage across various countries. The median JS size for websites in developed countries such as the US and Australia is 921.71 KB and 968.61 KB, respectively, while in developing countries like Bangladesh and Pakistan, the median JS size is significantly lower, at 466 KB and 547 KB. This trend is consistent across regions, with developed countries generally using more JavaScript code. The overall median JS size for developed countries is 725 KB, compared to 546 KB for developing countries. This shows a clear difference in JS usage, suggesting that websites in developed countries rely more on JS to create richer, more interactive user experiences.

Further supporting this trend, Figure 3.7b shows that websites in developed countries make more JS network requests than those in developing regions. For example, the median number of requests for websites in the US and Australia is 25 and 29, respectively, while in developing nations like Bangladesh and Pakistan, the median number of requests is 21 and 25, respectively. Overall, the median number of JS requests is 21 for developed and 18 for developing countries. This difference highlights again the greater reliance on JS in developed countries. When combined with the median document size (which includes all HTML files) shown in Figure 3.8, where developed countries have a median size of 33.6 KB compared to 26.1 KB in developing regions. It becomes evident that the increased JS usage contributes to the overall websites' size difference.

Figure 3.9 shows that this complexity is also reflected in the DOM size, or number of DOM elements in a webpage, which represents the structure and complexity of the page. A larger DOM size generally indicates a more complex webpage with more interactive elements, often driven by extensive JS usage. For example, the median number of DOM elements in developed countries like Germany (1,283) and Italy (1,058) is significantly higher than in developing countries such as Pakistan (940) and Russia (926). The overall median DOM size for developed countries is 969 elements, while in developing countries, it is much lower at 633 elements. This larger DOM size in developed countries highlights the increased complexity and interactivity of websites, which is often due to the heavier

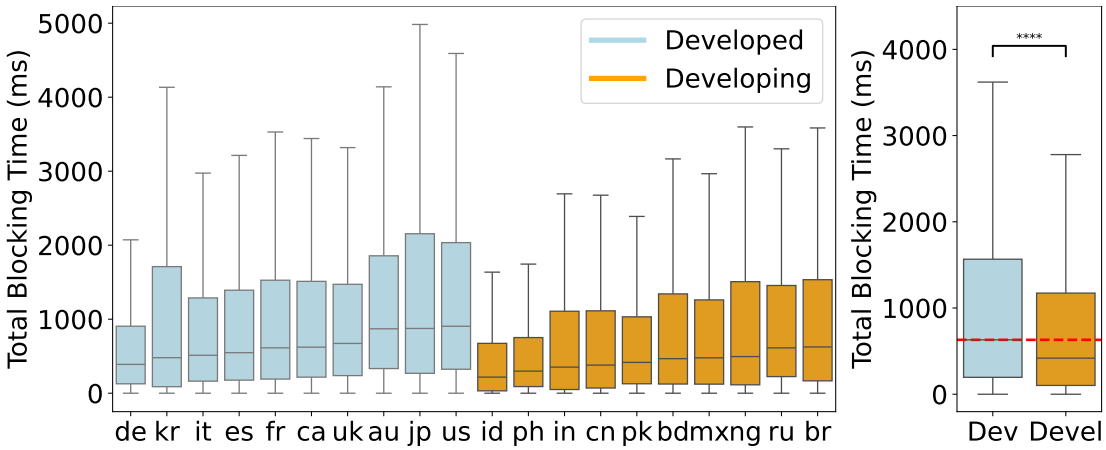


Figure 3.10: Comparison of Total Blocking Time (TBT) distribution between developed and developing countries.

use of JS to enhance dynamic content and features.

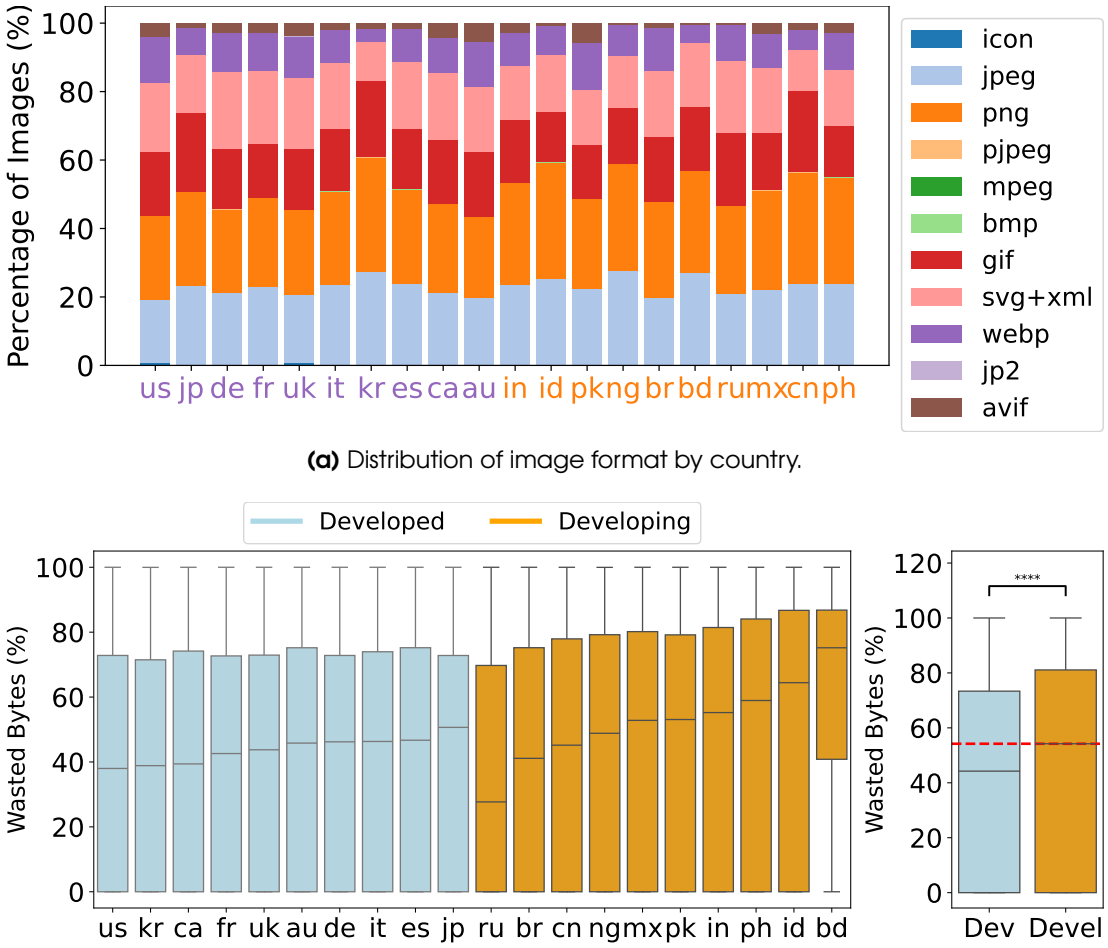
The higher complexity of webpages in developed countries, as indicated by larger document sizes and DOM elements, significantly impacts browser performance, particularly in terms of *Total Blocking Time (TBT)*. Figure 3.10 illustrates the distribution of TBT across developed and developing countries. TBT measures the duration during which the browser's main thread is blocked by tasks, preventing user interactions such as clicks, scrolling, or typing. The data reveals a significant disparity: websites in developed countries exhibit a median TBT of 631.0 ms, compared to 418.5 ms in developing countries. This longer blocking time is a direct consequence of the heavier use of JavaScript and larger HTML code in developed regions, which must be interpreted before the website becomes fully interactive.

3.5 Performance Optimizations

3.5.1 Image Compression

Optimizing image formats and using properly sized images are critical for improving web performance, particularly in regions with bandwidth constraints. Modern optimized formats such as WebP, AVIF, and SVG are recommended by best practices due to their superior compression capabilities and ability to maintain high image quality with smaller file sizes [205]. While JPEG 2000 (JP2) is not recommended, studies have shown that this format outperforms traditional JPEG in terms of compression efficiency [4].

Figure 3.11a shows the analysis of image formats across countries, which reveals disparities in the adoption of these optimized formats. Developed countries have a higher usage of modern formats like WebP, AVIF, and SVG, contributing to faster page loads and more efficient data usage. For instance, in the United States, modern formats account for approximately 37% of image usage, with SVG at 20.03%, WebP at 13.50%, and AVIF at 3.81%. Traditional formats like JPEG and PNG account for 17.90% and 24.52%, respectively. Similarly, Germany shows substantial adoption of modern



(a) Distribution of image format by country.

(b) Wasted image bytes (%) from compression inefficiencies by country.

Figure 3.11: Comparison of image format distribution (a) and percentage of wasted bytes due to compression inefficiencies (b) across different countries.

formats, with SVG at 22.45%, WebP at 11.33%, and AVIF at 2.69%, totaling about 36% of modern format usage. In contrast, developing countries have lower adoption rates of modern formats; in India, modern formats constitute approximately 28% of image usage—SVG at 15.91%, WebP at 9.64%, and AVIF at 2.68%—while traditional formats like JPEG and PNG dominate at 23.24% and 29.62% respectively. Nigeria exhibits even lower modern format usage at around 24%, with SVG at 14.95%, WebP at 9.02%, and AVIF at 0.52%, whereas JPEG and PNG account for 27.48% and 31.13%.

Next, we analyze image compression wastage, which refers to the proportion of image file sizes that exceed what is necessary for maintaining visual quality. Lighthouse computes this by comparing the size of the rendered image against the actual image size, accounting for the device's pixel ratio to assess whether images are optimally sized. Figure 3.11b shows that developed countries generally have lower image compression wastage rates compared to developing countries. Developed countries like the United States, South Korea, and Canada, which have median wastage rates of approximately

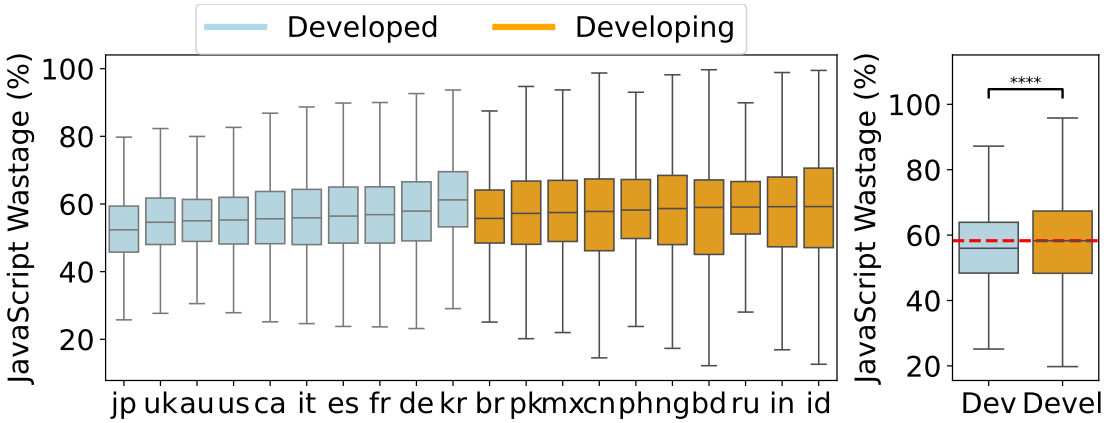


Figure 3.12: Distribution of unused JavaScript across developed and developing countries.

38-39%, also exhibit higher usage of modern formats such as WebP and AVIF in Figure 3.11, contributing to better management of image sizes and reducing wastage. In contrast, developing countries like India and Bangladesh, with higher wastage rates of 55% and 75%, respectively, show a lower adoption of optimized formats, indicating that inefficient image formats and large file sizes are more commonly used, particularly on smaller devices. This correlation between format usage and wastage emphasizes the importance of adopting optimized formats to improve performance.

3.5.2 Unused JavaScript and CSS Code

The unused JavaScript (JS) audit highlights code that is downloaded and parsed but never executed, contributing to significant performance inefficiencies. Although this code isn't directly used during page load, it still incurs costs as browsers must download, parse, and sometimes compile it. While tools like Lighthouse help identify this issue, they may overestimate the amount of unused JS by not fully simulating user interactions that might trigger the code later.

Several factors contribute to the presence of unused JS, including the use of pre-built libraries, legacy code, and a lack of regular optimization. Developers often include entire libraries or modules even when only small portions are needed. Additionally, outdated or redundant code can accumulate on websites, remaining in the assets due to limited maintenance or the complexity of refactoring. The increasing reliance on large JS frameworks exacerbates this issue, as these frameworks are often not tailored to the specific needs of individual webpages. Unused JS significantly impacts performance, especially on mobile devices. Mobile phones, particularly low-end models common in developing countries, require up to seven times more processing power than desktop devices to handle JS [89], further slowing down page loads. In regions with slower networks and higher data costs, the burden of downloading and processing unnecessary code is even more problematic.

On average, 55% of JS is unused across all countries, as shown in Figure 3.12. Developed nations like the United States (82.65%) and Japan (79.76%) show high

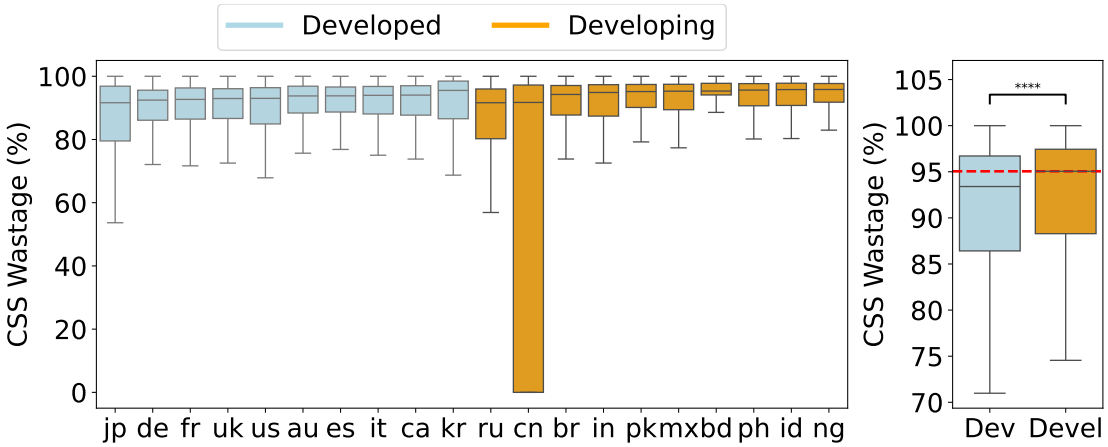


Figure 3.13: Percentage of unused CSS rules across developed and developing countries.

inefficiencies, while countries like South Korea (93.69%) and Germany (92.65%) fare even worse. The problem is most severe in developing nations such as Bangladesh (99.68%) and China (98.72%), where limited network speeds and strained resources amplify the negative impact of unused JS.

CSS wastage is also prevalent, with over 91% of global CSS rules remaining unused. Figure 3.13 presents the percentage of unused CSS rules across various countries, illustrating that developing nations like Pakistan and Bangladesh exhibit wastage levels above 95%. In comparison, developed countries like Japan and Germany report slightly lower rates, around 92%. This widespread wastage is likely due to outdated frameworks and poor mobile optimization.

It is crucial to address these inefficiencies, especially in developing regions where slow networks and low-end devices exacerbate the issue. Reducing unused JS and CSS through better development practices will improve performance, particularly for users in regions with limited network and device capabilities.

3.5.3 Webpage Performance

We conclude the section by analyzing the quality of browsing experience for users in both developed and developing regions. Using the CrUX dataset [139], which anonymously reports web performance metrics from mobile users, we focus on the Largest Contentful Paint (LCP) [516]. LCP measures how quickly the main content of a webpage becomes visible, a key metric recommended by Google as a reliable indicator of page load performance [516]. LCP performance is classified into three categories: Good ($\leq$ 2.5 sec), Needs Improvement (2.5-4 sec), and Poor ($\geq$ 4*sec*) [200]. CrUX reports the percentage of mobile visits in each country that fall into these categories for the pages under test. Although CrUX does not disclose the total number of reports for privacy reasons, we estimate a minimum of 70.2 million reports [163]. This provides valuable insights into real-world user experiences across different regions.

For visibility reasons, Figure 3.14 aggregates the results across developing and

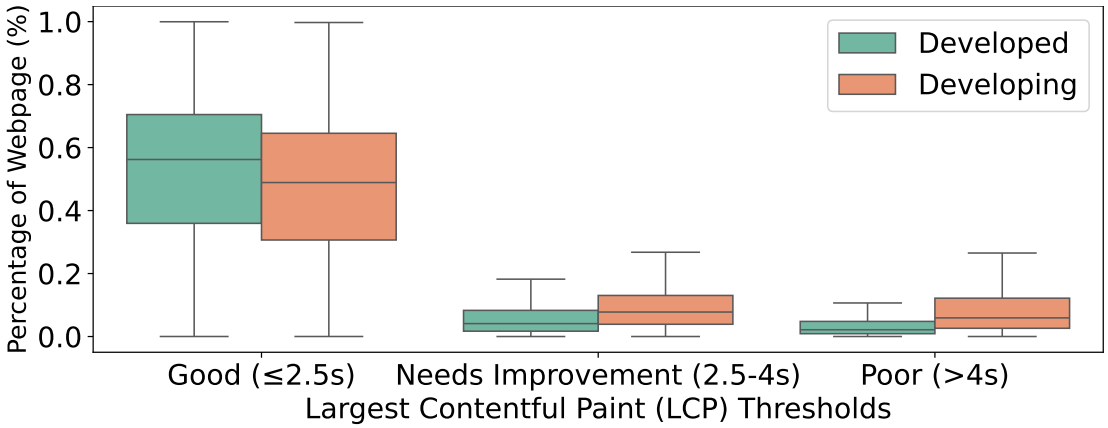


Figure 3.14: Distribution across developing and developed regions of the percentage of mobile webpage loads which were Good, Needs Improvement, or Poor.

developed regions, and shows the percentage of mobile visits which were Good, Needs Improvement, or Poor. For example, 50% of the developed webpages have close to 60% of their mobile visits, which were "Good", whereas this number drops to 50% for developed regions. Conversely, developing regions have overall more webpages characterized by either "Needs Improvement" or "Poor" performance. This analysis confirms that users in developing regions experience a slower web; however, the impact is less dramatic than expected. This is likely due to the overall smaller sizes, and lack of complex JS functionalities, previously discussed.

3.6 Privacy & Security Analysis

Trackers and Third-party Cookies They play a crucial role in how websites collect data on user behavior, preferences, and interactions. These tools enable personalized advertising, content customization, and detailed analytics. Trackers work by embedding scripts in websites to monitor user actions, while third-party cookies allow tracking across multiple domains. However, these practices raise privacy concerns, as they allow companies to build detailed user profiles. Surprisingly, despite stricter regulations like GDPR, our analysis, as shown in Figure 3.15, indicates a counter-intuitive trend: websites in developed countries tend to use more trackers despite the presence of these regulations. For example, 70.86% of websites in Germany have trackers, while this percentage increases to 93.07% in the United Kingdom and 92.06% in Australia. Other developed countries, such as Canada (89.50%), France (83.96%), and Japan (90.41%), also show high tracker prevalence. On average, websites in developed countries employ approximately nine trackers per site. In contrast, the prevalence of trackers in developing countries shows more variability. In China, 68.01% of websites have trackers, while Bangladesh (90.77%) and Pakistan (90.26%) exhibit similarly high tracker usage. Countries such as Brazil (89.55%) and Russia (91.52%) also have widespread tracker deployment. On average, websites in developing countries use six trackers, significantly fewer than their counterparts in developed countries. The higher tracker prevalence in

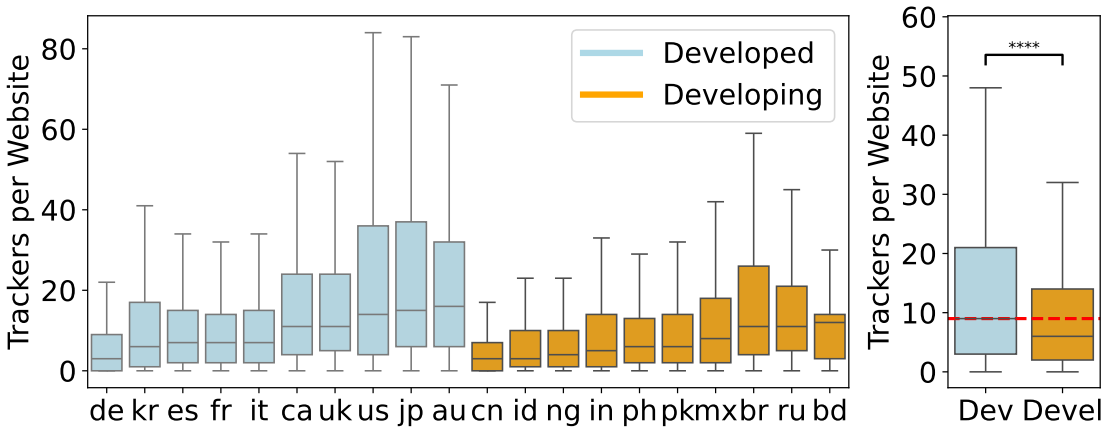


Figure 3.15: Comparison of the prevalence of trackers across countries.

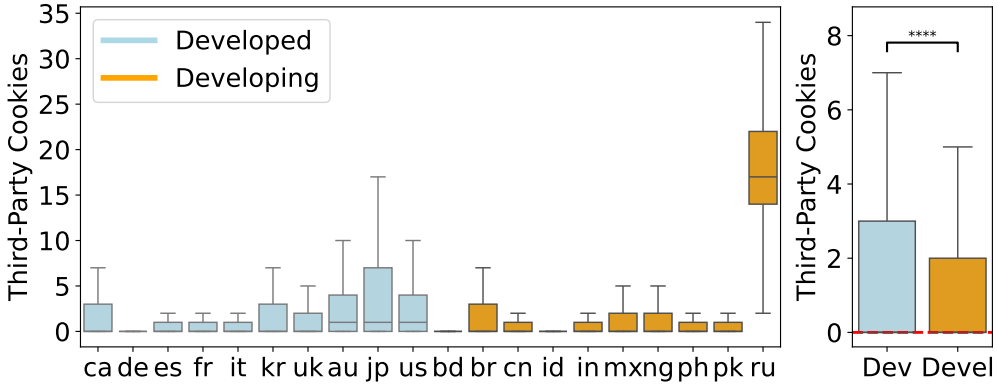


Figure 3.16: Comparison of third-party cookie usage across developed and developing countries.

developed nations can be explained by the fact that these countries have more mature advertising industries and e-commerce platforms that rely heavily on user data for targeted marketing and sales optimization. Additionally, data-driven business models are more prevalent, making user data a valuable asset.

In Figure 3.16, we show the distribution of third-party cookies across both developed and developing countries, showing a low overall prevalence, likely driven by changes in browser policies, e.g., Google Chrome is planning to soon drop support for such cookies. However, Russian websites stand out with a particularly high prevalence of both trackers and third-party cookies. The median number of trackers on Russian websites is 17, and 86% of these websites employ third-party cookies. This extensive use of third-party cookies and trackers in Russia is likely driven by a strong reliance on data-driven advertising models and lack of strict privacy-related legislation. These results are consistent with those of Gotze et al. [206], who also show that almost 90% of Russian websites extensively use both trackers and third-party cookies.

HTTPS Adoption HTTPS [415] encrypts web content protecting it from interception and manipulation. Figure 3.17 shows the adoption of HTTPS across both developed

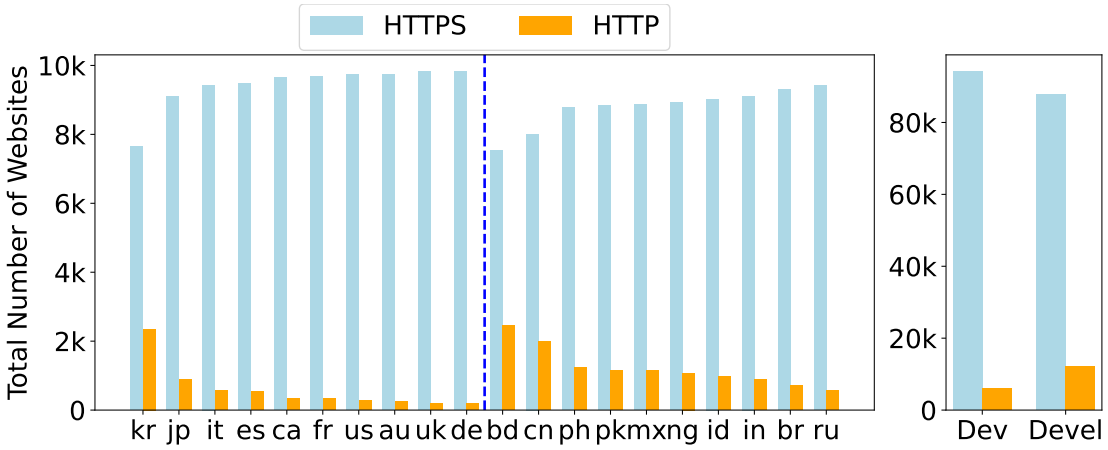


Figure 3.17: Country-wise HTTPS and HTTP adoption rates.

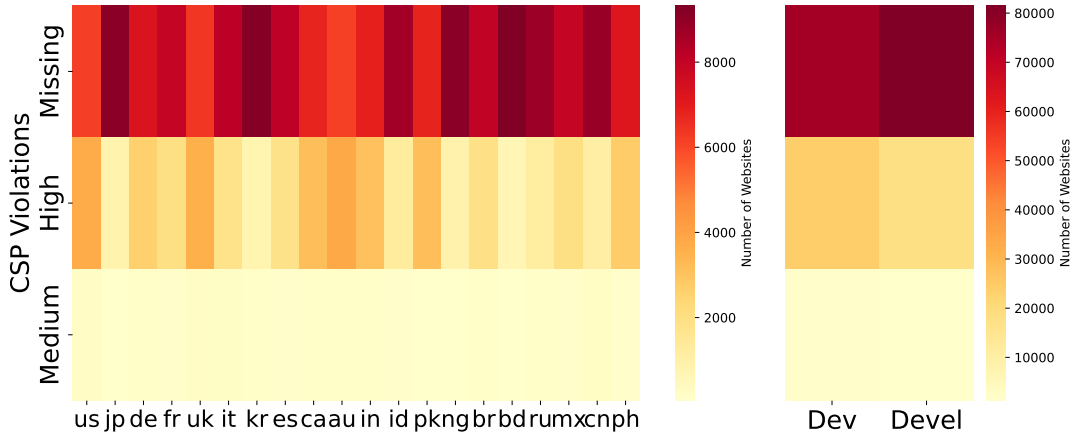


Figure 3.18: CSP usage rates across regions.

and developing nations. In developed countries, only 5% of websites still use unencrypted HTTP, a significant improvement from previously-reported measurements [157]. In contrast, 12% of websites in developing countries still rely on HTTP. This gap is even more concerning in certain countries, with Bangladesh and China having the most websites using HTTP: 25% and 20%, respectively. These results show a lag in adopting basic security standards in the developing world, potentially on purpose, leaving users in those regions vulnerable to (un)lawful interception.

Content Security Policy (CSP) Usage Content Security Policies (CSPs) act as a critical defense mechanism against cross-site scripting (XSS) and other injection attacks, offering a structured way to control which resources can be loaded and executed on a webpage. However, the correct usage of CSP is a known challenge [428]. Figure 3.18 illustrates CSP usage rates across countries, showing a significant lack of adoption in both developed and developing nations. In developed countries, 75% of websites fail to implement a CSP. The situation is even more concerning in developing countries,

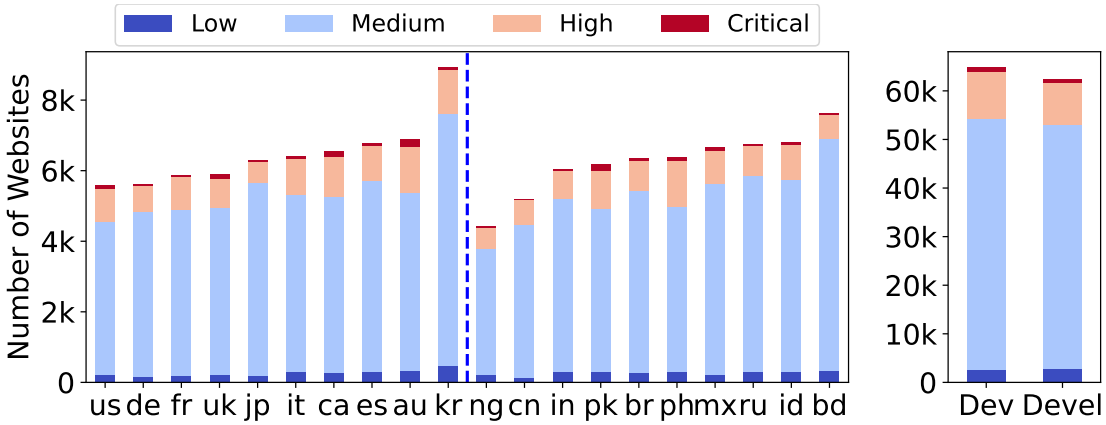


Figure 3.19: Number of vulnerabilities by country.

where 81.5% of websites do not implement CSP at all. This widespread absence of valid and robust CSPs is not confined to specific regions; over 99% of websites globally, in both developed and developing countries, either lack CSPs altogether or use highly ineffective deployments. These findings underscores a universal struggle with content security policies, as previously reported by developers in the recent work of Roth et al. [428].

Vulnerabilities in Web Ecosystems When analyzing vulnerabilities, an unexpected trend emerges: websites in developed countries are more prone to severe security issues than those in developing regions. Figure 3.19 provides a comparison of vulnerability rates across countries, highlighting the higher incidence of critical vulnerabilities in developed nations. For example, in developed countries such as Italy, Canada, Spain, Australia, and South Korea, around 10% of websites exhibit high or critical vulnerabilities. In contrast, the highest rate in the developing world is 7%, observed in Bangladesh. This trend is particularly interesting because, despite higher security standards, developed countries experience more critical security issues. One plausible explanation for this trend is the increased reliance on JS libraries and frameworks in developed regions (see Figure 3.4). While JS enhances website functionality, it also increases the attack surface. The vast ecosystem of third-party libraries and dependencies, which are not always well-maintained, can contribute to the large amount of observed vulnerabilities. These results are in line with prior measurements [300, 287] that report that 40% of the websites contain a client-side vulnerability.

Outdated and Duplicate Libraries The use of outdated and/or duplicate libraries can significantly increase a website's exposure to security risks, as these libraries may harbor vulnerabilities that attackers can exploit. Figure 3.20 illustrates the prevalence of outdated libraries across different countries. Developers across regions often hesitate to upgrade to newer versions of libraries. These results are in line with Raula et al. [284] who state that 81.5% of systems include at least one outdated library, and Lim et al. [300] who say that the majority of websites include JavaScript libraries that are more than seven years old. The issue is especially critical in Korean websites, where 94% of websites use at least one older library version, contributing to the higher number of

Figure 3.20: Outdated libraries usage across countries.

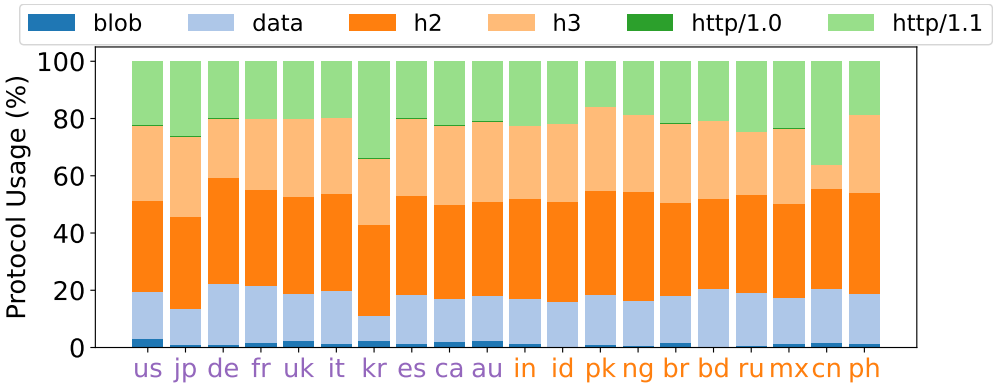


Figure 3.21: Comparison of the adoption of HTTP protocols across developed and developing countries.

vulnerabilities observed in Korean websites, as shown in Figure 3.19.

3.7 Technology & Affordability

Technology Adoption The adoption of new technologies reflects factors like economic development and infrastructure. While developed countries are assumed to adopt new technologies faster, data on HTTP protocols challenges this notion.

Figure 3.21 shows HTTP protocol usage across countries. Newer protocols like HTTP/2 (h2) and HTTP/3 (h3) improve performance and security. Surprisingly, websites in developing countries are not more reliant on older protocols than developed nations. Nigerian websites, for example, show high combined h2 and h3 usage at 65% (37.80% for h2, 27.09% for h3), exceeding the US at 58% (31.87% h2, 26.17% h3). Similarly, Pakistani websites report strong adoption with 36.03% for h2 and 29.39% for h3. In contrast, some developed countries like Germany and the UK report lower combined adoption of h2 and h3. Outdated protocols like HTTP/1.0 remain rare but slightly more common in developed countries (e.g., US: 0.31%, Nigeria: 0.02%). These findings challenge assumptions about technological leadership.

Figure 3.22 shows the adoption of various Web APIs; we first selected 74 Web API

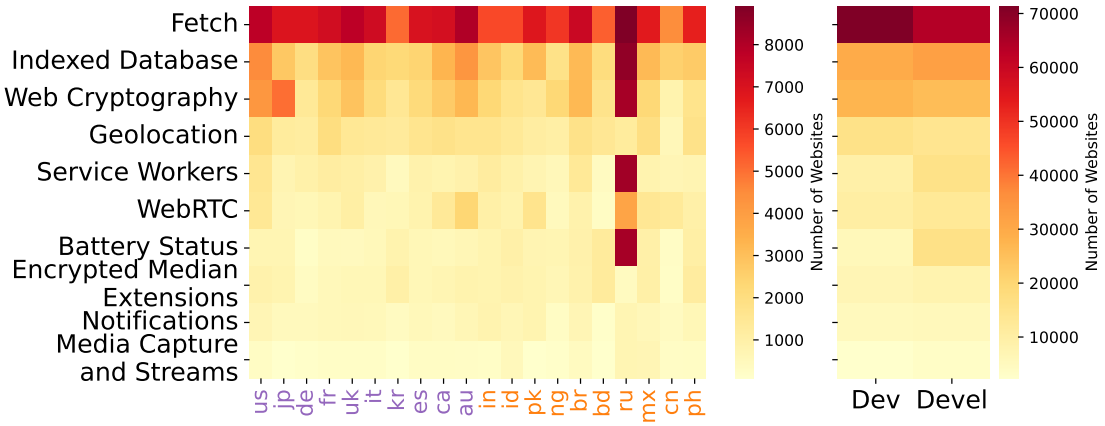


Figure 3.22: Comparison of the usage of web APIs across developed and developing countries.

standards using the same technique as [452], and then focused on 10 APIs that are security- and privacy-critical. The results show that the adoption of these security-critical APIs is not confined to developed countries. Russian websites show significant use of the Web Cryptography API (8,219 instances vs. 4,214 in the US). Similarly, Bangladeshi websites surpass US sites in Encrypted Media Extensions usage (1,307 vs. 887 instances). Nigerian websites also demonstrate active engagement with modern web technologies, with 2,318 instances of the Web Cryptography API, highlighting strong adoption in developing regions. These findings indicate that developing countries are embracing new technologies, sometimes at rates matching or surpassing developed nations. This may be due to the absence of legacy systems, enabling easier adoption of modern technologies, alongside widespread mobile internet use and a younger, tech-savvy population.

Website Affordability To assess websites' affordability, we use the PAW (Price Adjusted Web access) index [402]. This metric evaluates the affordability of accessing web content in a given region by comparing average broadband prices, local webpage sizes, and income levels to international affordability targets. The PAW Index is calculated using the following formula:

$$PAW_i = \frac{P_i}{P_T} \times \frac{W_{j,i}}{W_{global}}$$

where P_i represents the average broadband price in country i, and P_T is the target broadband price, set at 2% of the country's Gross National Income (GNI) per capita in accordance with the UN Broadband Commission's target [112]. Furthermore, $W_{j,i}$ denotes the size of webpage j in country i, and W_{global} represents the global average webpage size. A PAW index of 1 or less indicates affordable access, while values above 1 suggest that webpage sizes or broadband costs are too high relative to income, limiting access to essential online services [402].

Figure 3.23 shows the PAW indices for 20 countries, grouped into developed and developing. The index values were based on average webpage sizes and mobile broadband prices in each country. The median PAW index for both developed and developing

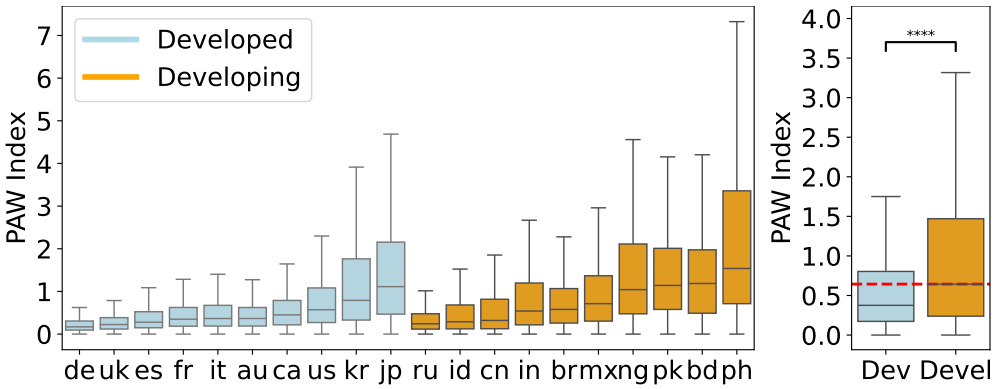


Figure 3.23: Website affordability (PAW Index) across countries.

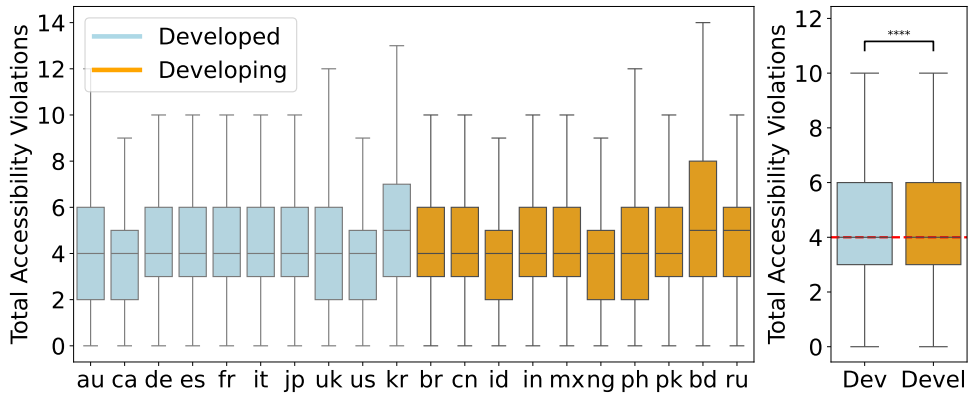


Figure 3.24: Total accessibility violations across countries. The similar median values suggest that accessibility challenges are a global issue.

countries was below 1, indicating overall affordability. Japan (1.11) was the only developed country with a PAW index above 1. In developing countries, Pakistan (1.14), Nigeria (1.04), Bangladesh (1.19), and the Philippines (1.54) exceeded the threshold, with the Philippines having the highest index. This points to substantial barriers to web access, due to a combination of high mobile data costs and large webpages. Developed countries showed less variation in PAW values, while developing countries had more inconsistency in affordability across regions, reflecting significant regional disparities.

3.8 Accessibility Violations Analysis

Figure 3.24 shows the total number of accessibility violations across countries, highlighting that accessibility challenges are a global issue affecting both developed and developing nations. Similar trends have been observed in prior studies [56], which indicate that accessibility issues persist worldwide due to a lack of prioritization and awareness, rather than economic or technological limitations. To assess each website's accessibility, we use Google Lighthouse, an automated tool that evaluates web content against the Web Content Accessibility Guidelines (WCAG) [255]. Lighthouse calculates

an Accessibility score as a weighted average of audits, informed by axe-core [136], an open-source engine that analyzes user impact and identifies compliance issues. Our evaluation specifically targets WCAG 2.2 [256], which is organized around four principles: Perceivable, Operable, Understandable, and Robust (POUR). These principles ensure accessibility for diverse user needs, including those with disabilities, across various devices.

The median number of accessibility violations remains consistent at 4 for most countries, with exceptions like Bangladesh, Korea, and Russia recording slightly higher medians of 5. However, this uniformity belies significant disparities in maximum violation counts: Bangladesh leads with 14 violations, followed by Korea (13), and Australia, the Philippines, and the United Kingdom (12). Most nations, including Brazil, China, Germany, France, India, and Pakistan, cluster around 10 violations, while Canada, Indonesia, Nigeria, and the United States report marginally lower counts at 9. The striking similarity in median values across diverse economic and technological contexts underscores that accessibility challenges are systemic rather than isolated to specific conditions. This pattern suggests violations stem not from national wealth or infrastructure gaps but from a pervasive lack of global awareness and prioritization of accessibility standards in development practices.

These findings emphasize the urgent need for coordinated action to foster inclusivity. Prioritizing developer education, enforcing compliance incentives, and harmonizing accessibility policies globally could mitigate barriers and create a more equitable digital landscape for all users.

3.9 Data Availability

To promote reproducibility and encourage further research into the digital divide, we have made all artifacts from this study publicly available. The list of websites analyzed, along with the scripts and other details, can be accessed at:

`https://github.com/kal-purush/digital-disparities-www25`

3.10 Summary

Despite two-thirds of the global population being online, a significant digital divide persists: 93% of people in developed regions are connected, compared to just 60% in developing ones. This gap, exacerbated by reliance on mobile internet and high data costs, affects access and user experience, with slower load times, and accessibility challenges in developing regions. This study examines these disparities by comparing the web in developed and developing regions. Using Google CrUX, we identified the 10,000 most popular websites in the top 10 countries from each group. Next, using Google Lighthouse and Puppeteer, we crawled and audited a total of 200,000 webpages. The resulting dataset, the largest of its kind, is publicly available to the research community. Our findings show websites in developing regions are 10% smaller and use fewer third-party resources and resource-heavy JavaScript, leading to simpler DOM structures and shorter browser blocking time. However, these sites exhibit weaker optimization,

wasting 10% more bandwidth on improperly sized images and redundant JS/CSS. Their smaller size seems to stem from simpler structures rather than intentional optimization. Security practices also lag, with HTTPS adoption at 88% (vs. 95%) and CSP adoption at 75% (vs. 81.5%). Whether these gaps reflect limited awareness or design choices for surveillance remains unclear.

4

Accessibility Challenges in the Global South

In the previous chapter, we examined how economic conditions shape the technical structure of the web, including differences in performance, security, and aggregate accessibility violations between developed and developing regions. In this chapter, we significantly deepen our investigation of accessibility, shifting the focus from broad technical metrics to the specific inclusion challenges that affect the lived experience of users with disabilities. In doing so, we address demographic diversity (challenge C_1) by asking a more fundamental question: is the web actually usable for its most vulnerable users?

As shown in §3.8, accessibility violations remain widespread across all economic contexts. However, approximately 80% of the world's disabled population lives in the Global South [500, 536]. In these regions, users face a distinct constellation of challenges, including heavy reliance on low-end mobile devices, expensive and unstable internet connectivity, limited availability of assistive technologies, and weaker regulatory enforcement. These factors compound technical accessibility violations, often making digital exclusion more severe than in developed economies. To quantify this landscape, we conduct the first large-scale measurement study of 100,000 websites across ten countries in the Global South, examining how local factors shape digital inclusion.

Our findings show that while laws and regulations can drive measurable improvements, accessibility outcomes in the Global South are shaped by additional societal and institutional factors. Accessibility violations are not merely technical errors but systemic barriers that disproportionately affect blind and low-vision users, often rendering essential digital services unreachable.

This chapter shares material with the corresponding publication [P2].

4.1 Motivation

The Global South refers to a broad set of countries primarily located in Africa, Asia, Latin America, and Oceania [117, 328, 247]. These regions are characterized by developing economies, diverse cultures, and a large proportion of the world's population. Collectively, they represent over 6 billion people or approximately 80% of the world's population [500, 536]. These countries are marked by their varied economic, social, and cultural landscapes, but many share common challenges in economic development, technological infrastructure, and access to essential resources. Although economic conditions often limit access to internet access, mobile technology has become a transformative force. For many in the Global South, mobile devices serve as the primary, and often only, means of accessing the internet, as mobile technology is generally more affordable and accessible than traditional desktop computing [498, 217].

With the increasing penetration of mobile internet in the Global South, web accessibility has emerged as a critical issue. Web accessibility refers to the practice of creating inclusive digital environments, enabling people of all abilities to interact with online content. Inaccessible websites limit users' access to essential information, digital services, and economic or social opportunities, deepening existing inequalities [512, 87]. Mobile platforms face unique accessibility challenges due to smaller screen sizes, touch-based interactions, and device variability. These challenges require dedicated solutions that go beyond traditional desktop accessibility practices [513, 81, 343].

Recognizing these differences, the World Wide Web Consortium (W3C) proposed updates to the Web Content Accessibility Guidelines (WCAG) in versions 2.1 and 2.2, focusing on mobile-specific issues such as small touch targets, adaptable layouts, and gestures [540]. However, the adoption of such guidelines remains questionable, as many developers are either unaware of mobile-specific requirements or do not prioritize them [513, 20]. Moreover, the desktop and mobile versions of the same website often present different accessibility issues [81, 336], as developers need to take additional steps to make mobile versions fully accessible. This gap in accessibility practices further underscores the need for targeted efforts to address the unique barriers mobile web users face.

Although there has been a growing body of research focused on WCAG, much of this work has concentrated on specific website categories or has been limited to single-country studies. Previous studies have examined accessibility in the public sector [260, 359, 317, 292] or government websites within individual countries [379, 3, 8, 259], providing insights into specific sectors but often overlooking a cross-national perspective. This narrow scope has limited the generalizability of findings and has created a knowledge gap regarding how accessibility practices vary across the Global South [1, 342]. This lack of regional and cross-national analysis restricts the insights needed to support broad-based accessibility initiatives in the Global South and hampers efforts to develop inclusive policies and practices that address the unique needs of these regions [39].

To address this gap, we conduct the first large-scale comparative analysis of mobile web accessibility across the Global South, evaluating 100,000 websites from 10 countries. Our study is automated using Lighthouse, an audit tool that enables accessibility assessments aligned with the WCAG standards [255]. This approach not only fills a critical gap in the literature but also lays a foundation for actionable improvements in web accessibility practices across the Global South. This chapter makes three primary contributions to the understanding of web accessibility in the Global South:

1. **Comprehensive Analysis**: This study examines mobile web accessibility issues across 10 countries in the Global South, providing a broad perspective on practices and challenges in diverse contexts.

2. **Identification of Common Challenges**: Our analysis highlights recurring issues like inadequate touch targets, low-contrast elements, and missing alt text, emphasizing systemic barriers requiring urgent attention.

3. **Impact on Diverse Disability Groups**: Our findings highlight how accessibility violations disproportionately affect different disability groups in the Global South, such as blind users (missing alt text, `meta-viewport` issues) and users with motor impairments (small touch targets).

4.2 Related Work

Accessibility research has grown recently but remains focused on the Global North, despite over 80% of people with disabilities living in the Global South [373, 238]. Studies on mobile phone use in these regions reveal a heavy reliance on social networks and

community support to address barriers like limited digital fluency and inaccessible designs [39]. Previous research also explores digitalization [373, 238], accessibility gaps [317, 359], and usability [342, 260] issues in mobile and web platforms. Sector-specific studies, such as those on university websites in Kyrgyzstan, Africa, Asia, and Latin America, reveal significant accessibility gaps hindering navigation for users with disabilities [260, 317].

Government websites are a major focus in accessibility research due to their role in public services. Studies from Tanzania, Turkey, Saudi Arabia, Oman, and Nigeria highlight widespread failures to meet WCAG standards, often due to limited awareness, resources, or policy enforcement [342, 8, 1, 3]. Similarly, healthcare websites frequently lack accessibility, creating barriers to vital medical information [161, 438]. Alajarmeh et al. [9] found low WCAG 2.0 compliance in healthcare websites across 25 countries. Italian tourism portals [121] also suffer from design issues affecting users with autism spectrum disorders. While valuable, these studies often focus on specific countries or sectors. Our research bridges this gap by analyzing accessibility practices across multiple sectors and countries in the Global South, offering a broader perspective for targeted interventions and policies.

4.3 Methodology

Our methodology involves identifying and analyzing 100,000 popular websites from 10 Global South countries, focusing on local top-level domain (TLD) or in-country hosting. Accessibility is evaluated using Google Lighthouse.

4.3.1 Country and Website Selection

We select 10 countries from the Global South based on population size and web data availability in the CrUX database [139]. These countries—China, India, Indonesia, Pakistan, Brazil, Nigeria, Bangladesh, Mexico, the Philippines, and Vietnam—cover approximately 4.19 billion people, or 51.7% of the global population. For each country, we analyze 10,000 websites, identified through a combination of country code top-level domains (ccTLDs) or registration data verified using `python-whois` [382] and the `who.is` API [530]. This method, applied to CrUX's top-ranked websites for each country, ensures a diverse sample while prioritizing locally relevant websites and minimizing the effect of global platforms, capturing the unique digital ecosystems of these nations.

4.3.2 Accessibility Evaluation

To assess each website's accessibility, we use Google Lighthouse, an automated tool that evaluates web content against the Web Content Accessibility Guidelines (WCAG). Lighthouse calculates an Accessibility score as a weighted average of audits, informed by axe-core [136], an open-source engine that analyzes user impact and identifies compliance issues. Our evaluation specifically targets WCAG 2.2, which is organized around four key principles: Perceivable, Operable, Understandable, and Robust (POUR). These principles ensure accessibility for diverse user needs, including those with disabilities,

across various devices. Lighthouse evaluates several critical aspects of accessible mobile design, encompassing:

- **Color Contrast:** Verifies adequate contrast for readability by users with visual impairments.
- **Touch Targets:** Ensures interactive elements are mobile-friendly, helping users with motor impairments.
- **Alternative Text:** Checks for descriptive alt text on images for screen reader users.
- **Form Labels:** Confirms proper labeling of form fields for easier navigation with assistive technologies.
- **ARIA Descriptions:** Ensures Accessible Rich Internet Applications (ARIA) descriptions are properly implemented for assistive technologies.

These evaluations are compiled into an aggregated *accessibility score* ranging from 0 to 100, providing a quantitative measure of each website's overall accessibility performance.

4.3.3 Alignment with Accessibility Standards

Our approach adheres to WCAG 2.2, with a focus on Levels A and AA, which set essential and enhanced accessibility standards for usability. Developed from WCAG 1.0, WCAG 2.0 introduced adaptable principles and technology-neutral recommendations, later expanded in WCAG 2.1 to support mobile accessibility. By aligning with these standards, our analysis reflects widely accepted expectations for accessible design. By applying Lighthouse's WCAG-aligned evaluations on popular websites in each country, this methodology offers a robust cross-country perspective on mobile web accessibility practices in the Global South. This approach highlights the strengths and areas for improvement within each country's digital landscape, identifying common accessibility challenges and opportunities for inclusive design across the region. Our analysis provides valuable insights into how accessibility is implemented on high-traffic mobile websites in these countries, supporting efforts to make the web more inclusive and user-friendly for all.

4.4 Results

We conduct the accessibility audit using Google Lighthouse [203], with a 30-second timeout per webpage. To ensure mobile rendering, audits emulate an iPhone X (375x812 resolution, scale factor of 3). Our software runs on university servers in Germany to reduce bias from commercial cloud IPs [265].

Table 4.1: Average number of accessibility violations per website by country

Country	All Issue	Critical	Country	All Issue	Critical
Bangladesh (bd)	5.48	1.29	India (in)	4.40	1.06
Vietnam (vn)	4.92	1.33	Mexico (mx)	4.21	1.01
China (cn)	4.63	1.54	Philippines (ph)	4.11	0.97
Pakistan (pk)	4.46	1.02	Nigeria (ng)	3.90	0.82
Brazil (br)	4.45	1.05	Indonesia (id)	3.88	1.01

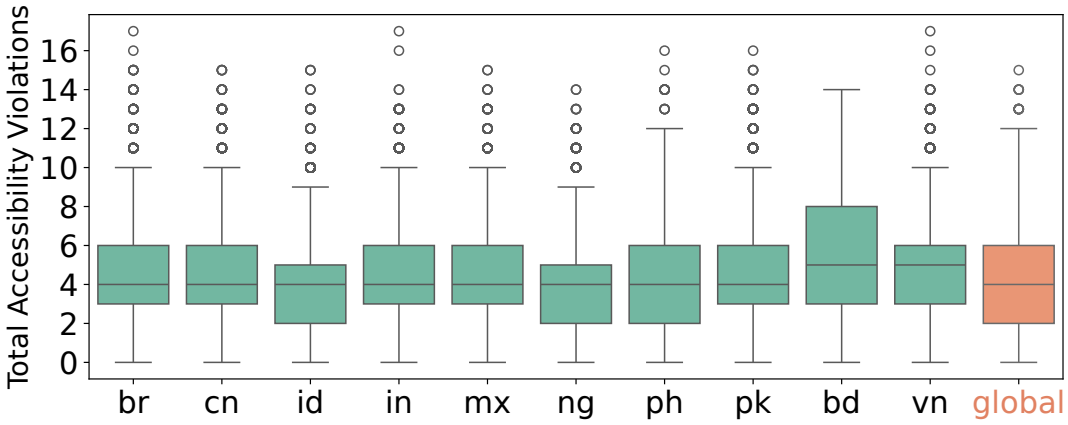


Figure 4.1: Distribution of accessibility violations across countries.

4.4.1 Accessibility Scores Across Regions

Table 4.1 provides an overview of the average number of accessibility and critical issues per website for each country. The table shows widespread accessibility violations across all the countries analyzed, with the average number of critical issues per website showing considerable variation. Bangladesh exhibits the highest averages, with 5.48 total issues and 1.29 critical issues per website, indicating substantial accessibility challenges. In contrast, countries like Nigeria (3.90 total issues, 0.82 critical issues) and Indonesia (3.88 total issues, 1.01 critical issues) report lower averages, reflecting relatively better compliance with WCAG guidelines.

Figure 4.1 further investigates the distribution of accessibility violations across countries, highlighting stark contrasts in WCAG compliance within the Global South, alongside data for the global top 10k websites as a benchmark. Globally (orange boxplot), 27% of the top 10k websites exhibit more than five accessibility violations, revealing room for improvement even among leading web platforms. Bangladesh stands out, with nearly 50% of websites exceeding five accessibility violations. This aligns with prior studies [6, 38] that identified significant issues in Bangladeshi websites, particularly in government and education sectors. In contrast, countries like India (30.2%), the Philippines (25.7%), and Nigeria (21.9%) report fewer websites exceeding this threshold, indicating better compliance. Disparities grow at higher thresholds: over 32% of Bangladeshi websites report more than seven violations, compared to 8% or less in the Philippines, Nigeria, Mexico, and globally among the top 10k. These discrepancies underscore substantial

Table 4.2: Top 10 most common accessibility issues

Issue	Occurrences	Issue	Occurrences
Color Contrast	76,040	Button Name	20,640
Link Name	62,440	Frame Title	13,537
Image Alt	43,915	Label	10,121
Meta-Viewport	26,230	List Structure	9,744
HTML Has Language	21,571	Select Name	7,955

accessibility gaps in Bangladesh, likely stemming from the lack of robust regulatory frameworks, insufficient incentives to enforce inclusive design practices, and insufficient knowledge or awareness of accessibility laws and guidelines among developers [38, 510].

Vietnam also exhibits a significant number of websites affected by accessibility issues, with 38.5% of its websites exceeding five violations, reflecting challenges in compliance. In contrast, countries with established accessibility laws, such as India and Brazil, show lower levels of violations. For example, India's *Rights of Persons with Disabilities Act, 2016* mandates digital accessibility, fostering greater compliance across public and private sectors. Likewise, Brazil's *eMAG - e-Government Accessibility Model*, enacted in 2014, supports accessibility guidelines that encourage inclusive design practices and help address gaps in developer knowledge.

Countries with higher accessibility violations, such as Bangladesh, Pakistan, and Vietnam, often grapple with deeply rooted social stigma surrounding disabilities, which exacerbates the problem. In these societies, individuals with disabilities are frequently marginalized, and their needs are overlooked in public discourse. For instance, in Pakistan, the national census officially reports that only 0.48% (3.3 million) [329] of its population is disabled, a figure starkly contradicted by the British Council's estimate of 27 million [75]. This significant disparity reflects how stigma surrounding disabilities leads to systemic underreporting and disregard for disability-related issues. This societal stigma not only impacts public perception but also influences government policies and priorities, leading to insufficient investment in accessibility initiatives and digital inclusivity. Similarly, in Bangladesh and Vietnam, societal attitudes often undermine efforts to promote awareness and education on accessibility, further hindering progress [85, 376].

These findings underscore the critical role of regulatory frameworks, education, and societal attitudes in promoting web accessibility. The stark contrast between countries with high proportions of violations, like Bangladesh and Vietnam, and those with lower proportions, such as Brazil and India, underscores the importance of government policies, national advocacy, and developer training in fostering inclusive digital environments. Addressing societal stigma is equally crucial, as it directly affects the prioritization of accessibility in governance and the creation of a more inclusive digital landscape for individuals with disabilities.

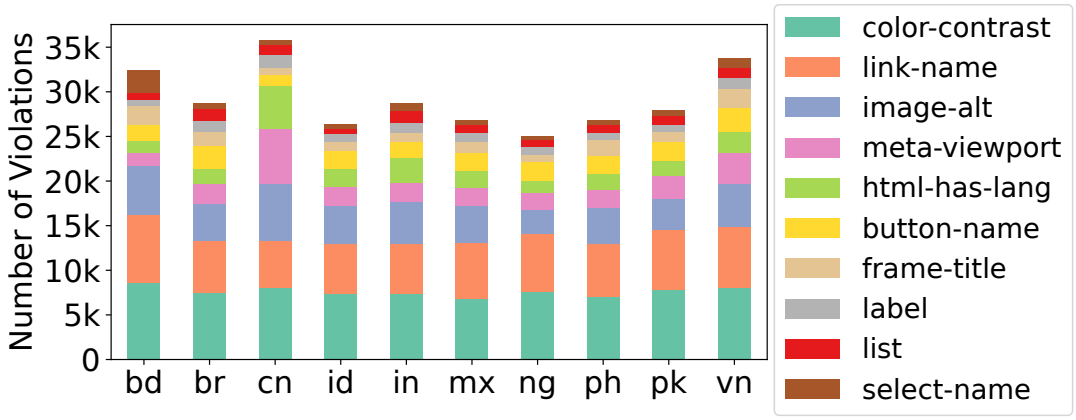


Figure 4.2: Number of different accessibility issues identified across countries.

4.4.2 Common Accessibility Violations and Their Impact

Next, we investigate which specific accessibility challenges, e.g., low contrast or lack of alt-text, are responsible for the identified violations. Table 4.2 shows the top 10 accessibility issues identified across countries, highlighting those that pose the most substantial barriers to web accessibility. Figure 4.2 shows the distributions of the top 10 accessibility issues across countries. Among these, *color contrast* violations are the most frequent. Individuals with low vision often struggle to distinguish elements with poor contrast, as areas of brightness and darkness may blend together, making it difficult to discern outlines, borders, and text details. Text with insufficient luminance contrast against its background is particularly challenging to read, especially for the many people who have low vision or color blindness. Without adequate color contrast, users may find it difficult to read and interact with content. Bangladesh exhibited the highest number of color contrast violations (85.6%), followed by China (80.3%) and Vietnam (80.03%), indicating a significant need to address text visibility and clarity on mobile screens in these regions.

Link name issue is another widespread problem. Missing or non-descriptive link names create substantial challenges for users relying on screen readers or keyboard navigation. Users with visual impairments and/or unable to use a mouse depend on meaningful link descriptions to understand the purpose and destination of each link. Without accessible link names, navigation becomes confusing and inaccessible, as users cannot determine where a link will lead. Again, Bangladesh shows the highest count of link name violations (76%), followed by Vietnam (68%) and Pakistan (67%). Image alt text is also missing in a large number of cases; China recorded the most missing alt text cases (63%), followed by Bangladesh (55%) and India (46%). Alt text is crucial for conveying an image's content and purpose, allowing screen readers to convert text into sound or braille. Without it, images become inaccessible, creating gaps in understanding and engagement.

Meta-viewport configuration issue is also prevalent, affecting how websites scale and display on mobile devices. Proper viewport settings are essential for mobile

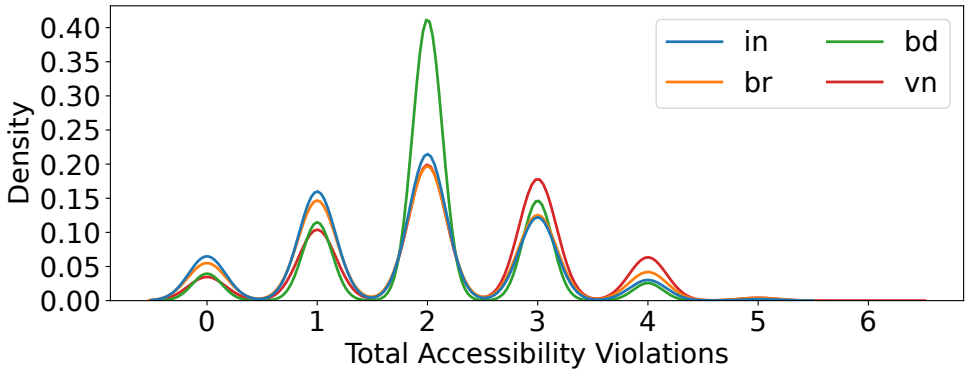


Figure 4.3: Distribution of mobile accessibility violations for Bangladesh, Brazil, India, and Vietnam.

accessibility, allowing users to zoom and adjust text size as needed. Parameters like `user-scalable="no"` or restrictive `maximum-scale` values limit the ability of users with low vision to enlarge content, creating usability challenges on smaller screens. Users with partial vision often rely on zooming functions or screen magnifiers to make text readable, and improper viewport settings can prevent these adjustments. China recorded the most viewport configuration issues (61%), followed by Vietnam (34%) and Pakistan (25%), highlighting the need for mobile optimization to enhance accessibility.

4.4.3 Mobile Accessibility Challenges

Although WCAG 2.0 is highly relevant to both web and non-web mobile content and applications [538], the additional guidelines introduced in WCAG 2.1 and WCAG 2.2 provide more targeted support for mobile accessibility [256]. These updates address critical elements such as color contrast, touch target sizing, and screen navigability, which are essential for improving the mobile web experience. For example, the *touch-size* guideline ensures that touch targets are sufficiently large and well-spaced, addressing a key challenge of small screens. This is particularly critical for users with fine motor impairments, such as arthritis or tremors, who may struggle with small or tightly packed touch elements.

We identified 10 accessibility tests from Lighthouse that closely align with WCAG 2.2 standards for mobile usability, as shown in Table 4.3. These tests offer a comprehensive framework for assessing and improving the accessibility of mobile websites, ensuring they meet the needs of diverse users and provide an inclusive experience.

Figure 4.3 shows the distribution of *mobile accessibility* violations for four representative countries: Bangladesh, Brazil, India, and Vietnam. The results for other countries fall between these distributions, highlighting varying levels of compliance. While Bangladesh exhibited a high overall count of accessibility violations (Figure 4.1), 76.5% of its websites have fewer than three mobile accessibility violations. This indicates that despite the higher overall violations, most websites adhere relatively well to mobile accessibility guidelines. Similarly, 73.9% of the Indian websites are affected by three violations. These findings align with 4.4.1, which highlights higher accessibility compli-

Table 4.3: Mobile accessibility checks categorized by WCAG principles

Principle	Checks
Perceivable	`color-contrast`, `meta-viewport`, `touch-size`, `input-image-alt`
Operable	`scrollable-region-focusable`
Understandable	`label`, `link-name`
Robust	`button-name`

Table 4.4: Percentage of websites with no critical accessibility violations for blind users by country.

Country	Percentage (%)	Country	Percentage (%)
`ng`	53.56	`br`	42.19
`pk`	46.29	`in`	40.74
`ph`	44.56	`vn`	33.53
`id`	43.58	`bd`	31.52
`mx`	43.02	`cn`	29.90

ance in these regions. In contrast, only 57% of the Vietnamese websites are affected by less than three violations, indicating a significant barrier for mobile users. Brazil's performance fell between these extremes, reflecting moderate levels of compliance and mobile accessibility challenges.

While this analysis suggests that mobile accessibility issues are less concerning than general issues, addressing severe issues such as `color-contrast`, `link-name`, `meta-viewport`, and `button-name` violations remains critical to improving accessibility and usability. By prioritizing these areas, developers can significantly enhance the mobile web experience, particularly in the Global South, where mobile-first access is essential.

4.4.4 Accessibility Challenges for Individuals with Disabilities

Accessibility challenges impact individuals with various disabilities, including `blindness`, `deafblindness`, `low vision`, `mobility impairments`, and `deafness`. Each type of disability requires specific accommodations to ensure equitable access to digital content. For instance, blind users depend on screen readers and well-structured, accessible content, while individuals with low vision rely on features like magnification, high contrast, and text resizing for better readability. Deafblind users may require advanced assistive technologies like braille displays. Individuals with mobility impairments may face challenges in using a mouse or touchscreen and often depend on alternative input methods such as keyboard navigation, voice commands, switch devices, or adaptive hardware to interact with digital content effectively. Similarly, deaf users depend on subtitles, transcripts, or visual alerts for content comprehension. Critical violations, as defined by Lighthouse and axe-core, are issues that directly prevent users with disabilities from accessing key features or content. These violations demand urgent remediation to ensure inclusivity. While axe-core provides an impact level based on

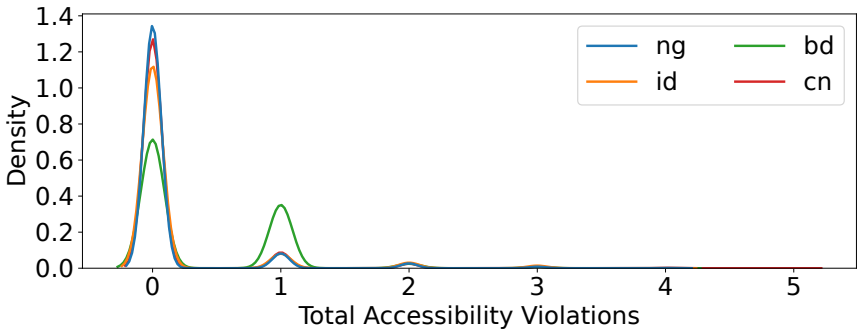


Figure 4.4: Distribution of critical accessibility violations for users with mobility impairments across countries.

expected severity, the actual impact may vary based on context, underscoring the need for tailored assessments.

Critical Accessibility Violations for Blind Users Table 4.4 summarizes the critical accessibility violations observed for blind users across the Global South. Globally, only 40.89% of websites meet critical accessibility standards, underscoring substantial barriers for blind users in accessing essential features and content. At the country level, notable variations are evident. Nigeria performs relatively well, with 53.56% of websites meeting the benchmark, followed by Pakistan (46.29%) and the Philippines (44.56%). In contrast, Bangladesh (31.52%) and China (29.90%) show the lowest proportions of websites passing the test, highlighting significant accessibility shortcomings. For example, **`https://fmcabuja.gov.ng`**, the official website of the Federal Medical Centre in Nigeria, is missing alt text for images, lacks proper use of ARIA attributes (e.g., `aria-allowed-attr` and `aria-required-children`), and does not have clear button names or labels, making it difficult for a blind person to access the website using a screen reader.

Critical Accessibility Violations for Mobility Impairments Users Figure 4.4 shows four representative distributions (China, Nigeria, Indonesia, and Bangladesh) of critical accessibility violations affecting users with mobility impairments. Compared to blindness, the figure shows much higher accessibility support, with 86.7% of the overall websites meeting the critical accessibility criteria. Nigeria leads in accessibility for mobility disabilities (91.91%), closely followed by China (91.15%) and Indonesia (89.88%). Bangladesh lags significantly, with only 65.01% of websites meeting the critical criteria, indicating a substantial gap. For instance, **`http://krishi.gov.bd`**, a website of the Ministry of Agriculture in Bangladesh, lacks proper keyboard navigation and does not implement accessible features such as well-structured ARIA attributes. Additional issues, including improper button size (`target-size`), inadequate logical tab order, and missing skip links, make navigation difficult for individuals with mobility impairments.

Critical Accessibility Violations for Low Vision Users Table 4.5 shows the accessibility assessment for users with low vision across the Global South; overall, 73.77% of the websites meet the criteria, thus outperforming blindness support while

Table 4.5: Percentage of websites with no critical accessibility violations for low vision users by country.

Country	Percentage (%)	Country	Percentage (%)
bd	85.74	id	78.26
ng	81.50	br	77.01
mx	79.14	pk	74.95
in	78.84	vn	65.01
ph	78.72	cn	38.52

under-performing support for mobility impairments. Notably, Bangladesh performs better in this category compared to its results for other disabilities, with 85.74% of websites meeting the benchmark. On the other hand, Chinese websites remain a concern, with only 38.52% meeting the criteria, highlighting persistent challenges for low vision accessibility. For instance, **https://www.mfa.gov.cn**, the official website of the Ministry of Foreign Affairs in China, lacks meta-viewport settings and sufficient color contrast, making it difficult for low vision users to navigate. Other countries, such as Nigeria (81.50%), Mexico (79.14%), and the Philippines (78.72%), perform close to the overall average, while Vietnam (65.01%) lags slightly behind.

4.5 Summary

This study provides the first large-scale analysis of mobile web accessibility across the Global South, evaluating 100,000 websites from 10 countries. Our findings reveal the uneven impact of accessibility issues on different user groups. While websites show better compliance for users with mobility impairments or low vision, blind users face critical violations like missing link names and alt text. This disparity highlights the urgent need for targeted interventions to address the unique challenges faced by blind users. Another notable finding is the correlation between national laws, the social stigma surrounding disabilities, and accessibility compliance. Countries with robust legislation, such as India's *Rights of Persons with Disabilities Act* and Brazil's *eMAG - e-Government Accessibility Model*, demonstrate higher adherence to accessibility standards. In contrast, countries like Bangladesh, Pakistan, and Vietnam, where stigma about disabilities is prevalent, show significantly lower compliance. This highlights how societal attitudes can hinder both policy development and enforcement. Further, this underscores the pivotal role of governance and policy in fostering inclusive digital environments. Despite these insights, there remains considerable room for improvement across all regions. Addressing widespread violations, especially those affecting the most vulnerable groups, will require stronger regulatory frameworks, developer education, and widespread adoption of accessibility-first design practices. These efforts are essential to bridge the accessibility gap and promote digital equity in the Global South.

Part II

Multilinguality in Web Accessibility and Development

5

Accessibility of the Multilingual Web

In the first part of this dissertation, we demonstrated how economic constraints and geographic location shape the structural quality and accessibility of the web. We showed that the "digital divide" is not merely a lack of connectivity, but is deeply embedded in modern software engineering practices related to security, privacy, optimization, and accessibility. As we transition to the second part of this work, we focus on a distinct but equally critical challenge: the friction between global linguistic diversity and the implicit monolingual assumptions embedded in software ecosystems (see challenge C_2 in the introduction). Specifically, we examine how the increasing presence of non-English and non-Latin languages challenges long-standing assumptions in web development and software tooling.

This challenge becomes especially visible in the context of web accessibility. Assistive technologies such as screen readers rely on both visible content and underlying accessibility metadata, including alternative text, ARIA labels, and language attributes, to interpret and present web interfaces. These mechanisms implicitly assume that the language of accessibility metadata aligns with the language of the visible interface. In multilingual settings, however, this assumption often breaks down. Websites frequently present content in regional or native languages while providing accessibility metadata in English, or omit language annotations altogether. As a result, screen readers often struggle with multilingual content and fail to convey meaning, creating practical accessibility barriers even on otherwise functional websites.

In this chapter, we examine the accessibility of the multilingual web by studying how these linguistic assumptions manifest in real-world web content. To enable a systematic analysis, we introduce `LangCrUX`, the first large-scale dataset of 120,000 popular websites across 12 languages that primarily use non-Latin scripts. Using this dataset, we find widespread neglect of accessibility hints and show that accessibility metadata frequently fails to align with the language of the visible content. Furthermore, we find that current automated accessibility testing tools do not check for this linguistic consistency; instead, they primarily verify the mere presence of accessibility text, and often do so imperfectly. To address this limitation, we propose `Kizuki`, a language-aware automated accessibility testing extension that accounts for the limited usefulness of language-inconsistent hints.

This chapter shares material with the corresponding publication [P3].

5.1 Motivation

An estimated 2.2 billion people live with some form of visual impairment, and 90% of them reside in low- and middle-income countries [537]. As internet adoption grows in these regions, an increasing share of web content is created in languages other than English. Many of these languages use non-Latin writing systems such as Devanagari, Bengali, Arabic, or Thai and are often presented alongside English in mixed-language interfaces [387, 389, 388]. Popular screen readers like JAWS [179] and NVDA [361] still exhibit limited support for non-Latin scripts and often perform poorly when confronted with mixed languages [28, 320, 504, 503]. These tools may misrender non-English words or produce unintelligible output, as has been documented with languages like Nepali [435].

The root of the problem lies in the inadequate usage of *text alternatives* [285, 255] metadata such as `lang` attributes or image `alt text`, which screen readers use to process content appropriately. When such metadata is absent, incorrect, or inconsistent with the visible text, it creates a mismatch between the displayed content and what the assistive tool offers. This issue is compounded for scripts that require complex shaping or language-specific pronunciation models. For example, Apple's VoiceOver [26], the default screen reader on macOS and iOS devices, does not provide any support for languages such as Urdu, Amharic, or Burmese [26]. This lack of linguistic inclusivity is at odds with the principles laid out by the W3C Web Accessibility Initiative, which advocates for equal digital access regardless of language or ability [539].

Web accessibility research only marginally studies the intersection of multilingualism and assistive technology [183, 546]. A key bottleneck in this area is the lack of data. Widely used datasets like Tranco [390] focus on popularity but do not provide insights into the linguistic composition of websites. This limits the researchers' ability to measure the prevalence of multilingual content and to assess the accessibility landscape for speakers of less commonly supported languages. To bridge this gap, we introduce `LangCrUX`, a dataset of 120,000 popular websites across 12 languages that primarily use non-Latin scripts. We construct LangCrUX by selecting high-traffic websites from the Chrome User Experience Report (CrUX) dataset, verifying language use via automated detection and manual sampling, and through Puppeteer-based crawls routed via country-specific VPNs to capture localized versions of each site.

We leverage `LangCrUX` to conduct the first large-scale analysis of multilingual web accessibility, focusing on how assistive technologies interact with real-world, multilingual web content. Our findings reveal that nearly 40% of websites in Bangladesh and India lack any accessibility text in the native language, despite having predominantly native-language visible content. More broadly, language-aware accessibility remains a widespread and under-addressed issue: many websites use non-descriptive, placeholder, or untranslated text (e.g., "file1," "button," or generic English terms) in critical accessibility elements like image `alt text`. These mismatches significantly reduce the utility of screen readers, which rely on metadata to convey meaningful information to users. We observe that current automated accessibility testing tools fail to detect these language inconsistencies, as they typically only evaluate the presence of accessibility hints. To address this gap, we propose and develop `Kizuki`[1], a testing extension that identifies such mismatches and evaluates metadata based on alignment with the surrounding linguistic context, offering a more inclusive measure of accessibility.

5.2 Related Work

Several studies have explored the challenges of multilingual web accessibility. Vázquez et al. [419, 509, 420, 422] conducted user studies highlighting the limitations of screen readers in handling multilingual interfaces. Casalegno [83] reported similar findings, emphasizing the cognitive strain users face when navigating mixed-language content. García-Garcinuño et al. [183] confirmed that language mismatches in screen reader

[1]Named after the Japanese word for "awareness"

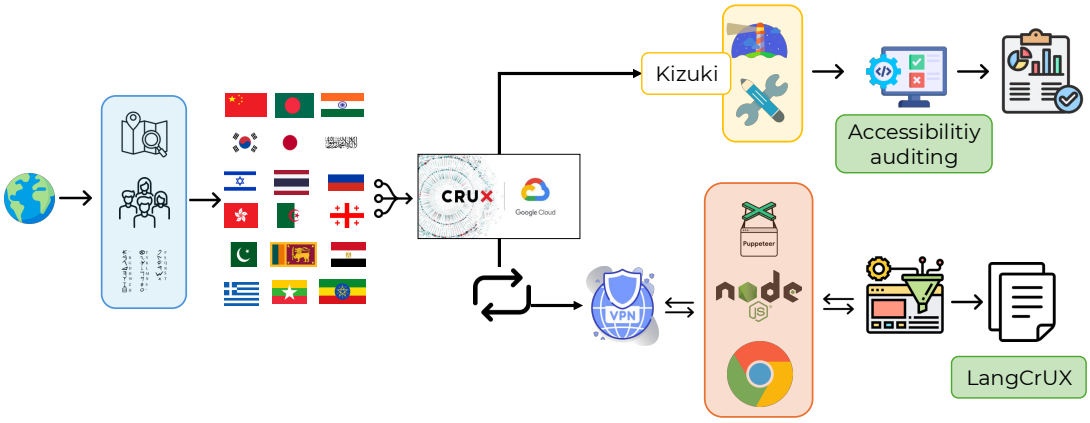


Figure 5.1: Overview of our methodology for constructing and analyzing the `LangCrUX` dataset.

output persist even when markup is correctly annotated. Vázquez et al. [423, 421] also showed that accessibility is often excluded from localization workflows, leading to untranslated alt text and inconsistent metadata. Several works highlight problems such as mispronunciation, broken accents, or robotic voices in the context of multilingual content for screen readers [320, 504, 503]. Sankhi et al. [435] documented similar barriers in Nepal, while Raghavendra et al. [407] pointed to the lack of robust multilingual speech synthesis systems in Indian languages, emphasizing infrastructural challenges for regional screen reader development.

Researchers also explored automated solutions to tackle accessibility issues. Several approaches are proposed for alt text generation, including human-curated [554, 191], image-search-based [219, 374], and AI-based methods [542, 10, 120]. While human and search-based approaches emphasize contextual accuracy and reusability, AI-driven techniques offer scalable alternatives, including generating captions [446, 334, 227] or even producing the image itself with embedded descriptions [10]. However, these methods still rely on high-quality training data and often require human oversight to ensure contextual relevance and inclusivity.

5.3 Methodology

We investigate the accessibility of websites in non-Latin-script languages, focusing on linguistic diversity in underrepresented language communities. We collect `LangCrUX`, a dataset comprising websites with a high proportion of non-Latin-script content, combining data from the Chrome User Experience Report (CrUX) [139], custom web crawls using Puppeteer, and language metadata verification. Figure 5.1 visualizes our methodology, including dataset construction, language selection, and automated accessibility analysis. `LangCrUX` is released as an open-source dataset on GitHub[2].

[2] `https://github.com/kal-purush/LangCrux`

5.3.1 Language and Country Selection Criteria

We begin with the top 30 widely spoken non-Latin-script languages, including Hindi, Bangla, Modern Standard Arabic, Tamil, Telugu, Mandarin Chinese, Urdu, Amharic, Russian, Marathi, Hebrew, Sinhala, Greek, Burmese, and others [152]. Our goal is to identify a diverse, representative set of languages underrepresented in web accessibility research, particularly those using non-Latin writing systems. Language selection is guided by three main factors: script type (non-Latin), size of the global speaker base, geographic and linguistic diversity. To ensure representativeness and sufficient data, we apply two strict inclusion criteria: 1) at least 10,000 websites with 50% or more visible textual content in the target language, and 2) inclusion in the CrUX dataset which provides user experience metrics from Chrome users with sufficient traffic and performance data. For languages spoken in multiple countries, such as Modern Standard Arabic, used in Algeria, Saudi Arabia, and Morocco, we select the country with the highest population of native speakers (Algeria, in this case). In multilingual countries like India, we include all major non-Latin-script languages with substantial speaker populations. However, only Hindi meets our data threshold; other widely spoken languages (e.g., Tamil and Telugu), do not meet the 10,000-website requirement. Similar exclusions apply to Sinhala (Sri Lanka) and Burmese (Myanmar), where websites with sufficient native-language content fell below the threshold despite initial inclusion.

The selected countries, their corresponding languages, and approximate global speaker populations are China (Mandarin Chinese, 1.2 billion), India (Hindi, 609 million), Algeria (Modern Standard Arabic, 335 million), Bangladesh (Bangla, 284 million), Russia (Russian, 253 million), Japan (Japanese, 126 million), Egypt (Egyptian Arabic, 119 million), Hong Kong (Cantonese, 85.5 million), South Korea (Korean, 82 million), Thailand (Thai, 71 million), Greece (Greek, 13.5 million), and Israel (Hebrew, 9 million). Collectively, these 12 languages are spoken by over 3.19 billion people, representing about 39.5% of the global population.

5.3.2 Website Selection

We use Google CrUX to identify and rank websites by real-world usage metrics. For each selected language-country pair, we extract the top 10,000 websites based on CrUX rankings, which reflect user engagement, load performance, and interaction quality. To validate language presence, we use a Unicode-based heuristic that matches visible text content against script-specific character ranges. Concretely, each language is defined by a set of Unicode blocks (e.g., Devanagari `U+0900–U+097F` for Hindi, Hangul `U+AC00–U+D7AF` for Korean, and Cyrillic `U+0400–U+04FF` for Russian). If multiple languages share the same base script, we extend the definition with characters unique to a specific language. For example, Arabic and Urdu both use the Arabic script, but Urdu includes additional characters such as ٹ (U+0679), ڈ (U+0688), ڑ (U+0691), which we explicitly include to disambiguate the two.

A website is included in our dataset only if at least 50% of its visible text is in the target language. To measure this, we classify all visible text into three groups: 1) target language, 2) English, and 3) other or unidentified scripts. The 50% cutoff was selected

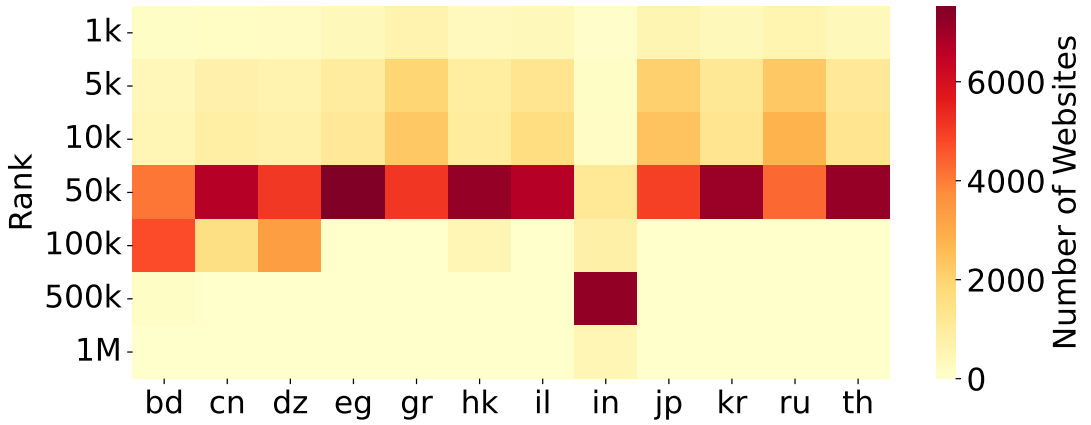


Figure 5.2: Distribution of website rankings across countries in `LangCrUX`. Each cell indicates the number of websites within a specific global rank range for a given country. Most countries have websites concentrated in the top 50,000 ranks, while India shows a broader distribution extending toward the 1 million rank range.

as a practical balance. A lower threshold (e.g., 30%) would capture websites where the target language appears only in marginal elements such as menus or advertisements. Conversely, a higher threshold (e.g., 90–95%) would exclude many widely used sites that combine native content with English navigation or technical terms.

In practice, we find that most websites are well above this cutoff: the median site contains 88% of its visible text in the target language, and fewer than 1% of sites fall close to the 50% threshold. Sites that do not meet the requirement are excluded and replaced with the next-ranked candidate from the Chrome User Experience (CrUX) list. When the top-ranked entries did not yield 10,000 qualifying websites, we extended the selection to lower-ranked CrUX entries until the quota was reached. Figure 5.2 shows the resulting distribution of global website rankings across countries in `LangCrUX`.

5.3.3 Data Collection

To extract accessibility-related features from the selected websites, we develop a web crawler using Puppeteer [399], which simulates web browsing conditions in a Chromium environment. Each website is visited programmatically, allowing us to capture network-level metadata, page structure, and accessibility indicators such as alternative text, ARIA (Accessible Rich Internet Applications) attributes (which enhance the semantics of web elements for assistive technologies [339]), and declared language tags.

To capture the localized experience of users in each country, we route all browser traffic through VPN servers physically hosted in the corresponding country. This step is critical for collecting region-specific versions of websites, as many sites dynamically serve content—including language settings, layout, or accessibility features—based on the user's IP location. Without VPN-based localization, web crawlers risk accessing global or English-dominant versions of websites that do not accurately reflect the intended user experience of native speakers. We use a combination of commercial VPN services, including ProtonVPN [398] and Hotspot Shield [249], to achieve broad geographic

Table 5.1: Web elements requiring natural language.

button-name	document-title	image-alt
frame-title	summary-name	label
input-image-alt	select-name	link-name
input-button-name	svg-img-alt	object-alt

coverage. Since not all VPN providers have servers in every target country, we select the provider on a per-country basis to ensure reliable and consistent access from within national borders. Compared to crawling from generic cloud-hosted IPs, this approach offers a significantly more realistic vantage point, reducing the risk of content variation, redirection, or censorship artifacts that may otherwise skew accessibility analysis.

5.3.4 Accessibility Element Selection Criteria

To identify accessibility elements where natural language plays a critical role, we follow a structured process based on the Lighthouse accessibility auditing framework [203]. Lighthouse evaluates a range of accessibility checks, each associated with specific HTML elements and best practices. Our goal is to select those elements for which the presence, clarity, and appropriateness of natural language directly influence accessibility outcomes.

We begin by examining the set of Lighthouse accessibility tests, which internally relies on the Axe-core accessibility engine [137]. For each test, we identify the corresponding HTML element it targets (e.g., `<img>`, `<button>`, `<input>`). We then analyze the test rationale by referencing the associated rule definitions from Axe-core, which provide detailed explanations and criteria for each audit. From these specifications, we assess whether natural language content is integral to the test. Specifically, we ask whether the accessibility of the element depends on human-readable text, for example, whether a screen reader user would rely on the clarity and relevance of that text to understand the element's purpose. If natural language is central to the function or evaluation of the element, we include it in our set. Following this process, we identify the twelve elements listed in Table 5.1 as language-sensitive accessibility features.

These elements span a range of interface components, including images, forms, buttons, and navigation elements, all of which rely on meaningful textual descriptions to be accessible to users with visual impairments. We explicitly exclude certain tests, such as `video-caption`. Although captions are inherently language-dependent and critical for accessibility, accurately evaluating them at scale poses challenges. In many cases, captions are not embedded in the HTML but are provided through separate files (e.g., VTT or SRT) or dynamically loaded via JavaScript. These may be inaccessible to crawlers unless the video is played or fully rendered in the browser. Furthermore, identifying whether a video has accurate, synchronized, and complete captions often requires manual inspection or audio-visual comparison—steps outside the scope of automated large-scale analysis. Due to these limitations, we omit video-caption checks to maintain consistency and reproducibility in our methodology.

Table 5.2: Summary statistics for accessibility elements. Metrics include median (Med), standard deviation (σ), and mean (μ) for missing and empty percentages, as well as descriptive richness (text length and word count).

Element	Missing (%)			Empty (%)			Text Length			Word Count		
	Med	σ	μ	Med	σ	μ	Med	σ	μ	Med	σ	μ
button-name	71.43	37.25	61.92	0	4.18	0.36	14	25.01	21.35	3	4.72	3.83
frame-title	87.50	30.09	75.81	0	3.33	0.21	13	11.57	17.45	1	2.39	2.54
image-alt	1.89	28.86	17.12	7.46	32.40	25.39	14	1332.15	22.97	2	8.27	3.67
input-button-name	100	22.62	93.90	0	4.03	0.19	12	13.12	14.26	2	2.70	2.83
input-image-alt	0	47.17	35.07	0	21.27	4.85	3	6.92	5.66	1	1.26	1.41
label	100	10.01	98.55	0	1.27	0.02	8	6.49	9.28	1	1.04	1.67
link-name	100	11.98	95.96	0	0.98	0.04	22	27.01	26.64	3	4.39	4.67
object-alt	100	23.30	94.19	0	5.09	0.26	9	17.64	14.26	1	3.48	2.49
select-name	100	28.78	89.84	0	2.00	0.05	10	23.03	12.94	2	2.26	2.30
summary-name	100	25.84	90.47	0	2.98	0.17	5	3.68	5.69	1	0.59	1.18
svg-img-alt	100	15.15	96.66	0	2.90	0.15	13	5.38	11.98	2	0.74	1.88

5.3.5 Limitations

Our methodology has several limitations. First, reliance on CrUX limits our scope to websites with measurable Chrome traffic, potentially excluding low-traffic or highly localized websites. Second, while using a VPN allows accessing region-specific versions of websites, some websites may detect VPN use and return generic or restricted versions. In such cases, we replace the affected websites with the next eligible candidate. Third, Puppeteer's simulated browsing environment may not fully reflect user experiences. Fourth, language detection (both automated and manual) can be challenged by embedded content or non-standard scripts, though our 50% content threshold and manual verification aim to reduce this risk. A further limitation comes from our script-based heuristic for language detection: although overlapping scripts (e.g., Arabic vs. Urdu) could lead to misclassification, this risk is low because we build country-specific website lists and crawl them through VPNs, which minimizes the likelihood of large-scale cross-language contamination. Finally, our evaluation of Kizuki is limited to 20,000 websites from two countries (Bangladesh and Thailand). While the results demonstrate its usability, broader evaluations across more languages and regions are needed to fully establish its effectiveness.

5.4 Is Multilingual Web Accessible?

Table 5.2 provides statistics on the quality and presence of accessibility text across twelve HTML elements. For each element, we report the median, standard deviation, and average percentage of websites where the accessibility attribute is missing or empty. We also include metrics like text length (in characters) and word count to assess richness

Table 5.3: Lighthouse test results for individual accessibility elements under different conditions. A ✓ indicates the test passed, while a ✗ indicates the test failed.

Accessibility Rule	Missing Element	Empty Value	Incorrect Language
button-name	✗	✓	✓
document-title	✓	✗	✓
frame-title	✗	✗	✓
image-alt	✗	✓	✓
input-button-name	✓	✗	✓
input-image-alt	✗	✗	✓
label	✓	✓	✓
link-name	✗	✗	✓
object-alt	✗	✗	✓
select-name	✗	✗	✓
summary-name	✓	✓	✓
svg-img-alt	✓	✓	✓

and verbosity. These measurements allow us to compare how frequently accessibility features are implemented and how informative they are when present. In the following, we analyze these patterns by examining missing and empty values, evaluating text length and word count, filtering uninformative content, and characterizing the language distribution of informative accessibility text.

Prevalence of Missing and Empty Accessibility Texts: Missing accessibility text is a widespread issue across HTML elements. Several elements exhibit extremely high average missing rates, including `label` (98.5%), `svg-img-alt` (96.6%), `link-name` (95.9%), `input-button-name` (93.9%), and `summary-name` (90.47%).
The `image-alt` attribute stands out for its relatively lower average missing rate (17.12%) but exhibits the highest percentage of empty values (25.39%). While an empty `alt` attribute is sufficient for passing automated accessibility checks, such text does not convey meaningful information to users and provides no semantic value for assistive technologies.

To better understand how automated tools handle these cases, Table 5.3 summarizes the results of our Lighthouse tests for the individual accessibility elements discussed in §5.3.4. We construct isolated test pages, each containing a single target element, and evaluate three scenarios: the element being completely missing, present but with an empty value, and present with content written in a different language. The table reports whether Lighthouse flags each scenario as passing or failing. The results show that Lighthouse primarily checks for the presence of accessibility attributes and often treats empty or language-inconsistent values as valid, helping explain the high prevalence of empty accessibility texts observed in practice.

To obtain comparison values for non-multilingual websites, we crawled the top 10,000 websites for each country and applied the same 50% threshold rule, selecting only those where the native (target) language share was below 50%. In practice, most of the resulting sites are global services and predominantly English. Using this baseline, we find that the missing percentage is slightly higher in multilingual websites (17.12% vs.

15.19%) and the empty percentage is notably higher (25.39% vs. 16.36%), indicating a greater tendency to include but not meaningfully populate the attribute in multilingual contexts.

To obtain comparison values for non-multilingual websites, we crawled the top 10,000 websites for each country and applied the same 50% threshold rule, selecting only those where the native (target) language share was below 50%. In practice, most of the resulting sites are global services and predominantly English. Using this baseline, we find that the missing percentage is slightly higher in multilingual websites (15.19% vs. 17.12%) and the empty percentage is notably higher (16.36% vs. 25.39%), indicating a greater tendency to include but not meaningfully populate the attribute in multilingual contexts.

Attributes like `button-name` and `link-name`, which are essential for identifying interactive components, also show high missing rates (61.92% and 95.96%, respectively). Similarly, attributes associated with form controls, such as `input-button-name` and `select-name`, are frequently absent or left empty, compromising the clarity of form interactions. A likely reason for these high rates is that, in many cases, screen readers fall back to reading visible HTML text (such as the inner text of a button or link) when accessibility attributes like `aria-label`, `label`, or `alt` are missing. This fallback behavior reduces the perceived need for developers to explicitly include accessibility metadata, especially when the element already includes visible text.

Text Length and Word Count Analysis: Table 5.2 also captures the descriptive quality of accessibility text through text length and word count metrics. Among the elements, `link-name` has a relatively high average text length ($\sim$27 characters) and word count ($\sim$5), compared to `summary-name`, `select-name`, `label`, and `button-name`, which show lower average word counts (1.17, 2.31, 1.67, and 3.86, respectively). This suggests that link descriptions tend to be more detailed and contextually informative. However, for some other elements, shorter texts are often acceptable; for example, buttons labeled "Login," "Send," or "Submit" typically provide sufficient clarity with just one or two words.

For elements like `image-alt`, which require contextual descriptions to convey the meaning of an image, we find an average word count of only $\sim$4. This shortness, especially when combined with the high empty rate noted earlier, suggests a broader trend of developers including minimal or superficial alt text, possibly to satisfy automated checks rather than to genuinely enhance accessibility. Finally, the table reveals substantial outliers, e.g., `image-alt` has a maximum of 261,864 characters and 12,306 words, while `link-name` reaches 5,228 characters and 518 words. These extreme but rare cases likely indicate instances where extraneous content, such as metadata, boilerplate text, or full paragraphs, has been mistakenly inserted into accessibility attributes, potentially overwhelming assistive technologies and undermining user experience. Table 5.4 presents representative examples of `image-alt` values exceeding 1,000 characters, extracted from webpages in Bangladesh, India, Japan, Greece, and Thailand, along with their source URLs.

Filtering Uninformative Accessibility Text: To assess the quality of accessibility text, we apply a filtering step to discard uninformative or placeholder texts. This is

Table 5.4: Examples of image alt texts exceeding 1000 characters in length, collected from websites in Bangladesh, India, Thailand, Greece, and Japan. Each row shows the corresponding alt text and the source URL where the alt text was extracted.

Image Alt Text	Source URL
সাউদার্ন ইউনিভার্সিটি বাংলাদেশ এর ৩৮তম অর্থ কমিটির সভা আজ রোববার সকাল সাড়ে ১১টায় স্থায়ী ক্যাম্পাস বায়েজিদ আরেফিন নগরের সম্মেলন কক্ষে অনুষ্ঠিত হয়েছে। অর্থ কমিটি ও বোর্ড অব ট্রাস্টিজের চেয়ারম্যান শিল্পপতি খলিলুর রহমানের সভাপতিত্বে আয়োজিত সভায় ট্রাস্ট সেক্রেটারি অধ্যাপক সরওয়ার জাহান, ট্রাস্ট ভাইস—চেয়ারম্যান শফিক উদ্দিন ও লিয়াকত আলী চৌধুরী, উপাচার্য(ভারপ্রাপ্ত) ড. শরীফ আশরাফউজ্জামান, ট্রাস্টি বোর্ডের সদস্য আব্দুস সালাম, অধ্যাপক ড. ইসরাত জাহান, ফেরদৌস ওয়াহিদ বাপ্পী, অধ্যাপক ড. সালেহ জহুর, অধ্যাপক প্রকৌশলী ড. মোজাম্মেল হক, জমির উদ্দিন ও শাহাদাত হোসেন সহ হিসাব শাখার কর্মকর্তারা উপস্থিত ছিলেন। সভার সার্বিক কার্যবিবরণী উপস্থিত সদস্যদের সামনে তুলে ধরেন কোষাধ্যক্ষ। আলোচনায় ৩৫, ৩৬ ও ৩৭তম অর্থ কমিটির সভার গৃহিত পদক্ষেপ ও প্রস্তাবনা নিশ্চিতকরণ, জুলাই ২০২৪ থেকে ডিসেম্বর ২০২৪ পর্যন্ত আয় ব্যয় বিবরণী উপস্থাপন, ভবন সম্প্রসারণ কাজের বাজেট প্রণয়ন, জমিক্রয়, প্রস্তাবিত বিভিন্ন খরচের অনুমোদন সহ বিবিধ বিষয় নিয়ে আলোচনা করা হয়। ভবিষ্যতে ইউনিভার্সিটিকে আরও সাফল্যের পথে এগিয়ে নিতে ঐক্যবদ্ধভাবে কাজ করার আশাবাদ ব্যক্ত করে সভার সমাপ্তি ঘোষণা করেন সভাপতি খলিলুর রহমান।	https://www.shiksharalo.net
भारत समाचार यूपी और उत्तराखंड का सर्वश्रेष्ठ हिंदी न्यूज चैनल है। भारत समाचार न्यूज चैनल राजनीति, मनोरंजन, बॉलीवुड, व्यापार और खेल में नवीनतम समाचारों को शामिल करता है। Bharat Samachar LIVE I UP News I Breaking News I CM Yogi I Akhilesh Yadav I UttarPradesh IUttrakhnad #BharatSamachar is India's best Hindi News Channel. #BSTV news channel covers latest news in politics, entertainment Bollywood, business and sports. #bharatsamacharlive #UttarPradesh #Uttrakhand #yogiadityanath #akhileshyadav #oprajbhar #pushkarsinghdhami #bjp #samajwadiparty #brajeshmisra #virendrasingh #bstv #bharatsamachar #breakingnews #viralnews Subscribe our channel for latest news updates :- https://goo.gl/4j1Nh1 Follow Us On: Instagram: https://www.instagram.com/bharatsamac... Facebook: https://www.facebook.com/BharatSamach... Koo : https://www.kooapp.com/profile/bharat... Twitter : https://twitter.com/bstvlive YouTube : https://www.youtube.com/channel/UC8Xb... Website : https://bharatsamachartv.in/	https://bharatsamachartv.in
เผยความเท่แบบ NEWTRO กับ G-SHOCK DW-6900TR เมเดลพิเศษฉลองครบรอบ 30 ปี กับดีไซน์อันเป็นเอกลักษณ์ที่ลงตัวกับ Street Lifestyle ครบรอบ 30 ปี DW-6900TR เคารพรุ่นต้นฉบับด้วยการใช้ดีไซน์แบบดั้งเดิมที่มากับจุดเด่นอย่างหน้าปัด Triple Graph แบบ 3 ตาอันเป็นเอกลักษณ์ ซ่อนความพิเศษด้วยคำว่า 'SINCE 1995' ที่จะปรากฏบนหน้าจอ LCD เมื่อไฟพื้นหลังสว่างขึ้น เป็นการอิงถึงปีของนาฬิกาตระกูล 6900 อีกหนึ่งความพิเศษ คือส่วนประกอบเรซินหลักที่ใช้ในขอบตัวเรือน สาย และส่วนอื่น ๆ ผลิตขึ้นมาจากเรซิ่นชีวภาพ ซึ่งเป็นวัสดุอินทรีย์ที่สามารถนำกลับมาใช้ซ้ำได้และช่วยลดผลกระทบต่อสิ่งแวดล้อม นอกจากนี้ DW-6900TR ทุกเรือนจะมากับกล่องดีไซน์พิเศษที่มีสัญลักษณ์ 'G' และมีสี 3 สีที่เป็นตัวเลือกของนาฬิการุ่นนี้มาเป็นส่วนหนึ่งในงานดีไซน์ เพื่อเป็นเกียรติแก่การออกแบบกล่องของนาฬิการุ่น DW-6900H ในอดีต อีกหนึ่งตัวเลือกของคนหัวใจสตรีทที่ชอบความเท่แบบเรียบง่าย สวมใส่ได้หลายโอกาส แมทช์กับการแต่งตัวได้หลายลุค เป็นนาฬิกาที่เราอยากใช้คำจำกัดความว่า "NEWTRO" จากการผสมผสานความเก่าและความใหม่เข้าด้วยกันได้อย่างลงตัวในเรือนเดียว พร้อมด้วยเอกลักษณ์ที่บ่งบอกถึงความเป็น G-SHOCK ได้อย่างแท้จริง มาพร้อมตัวเลือก 3 สี ได้แก่ สีแดง สีเหลือง และสีดำ พร้อมให้เป็นเจ้าของแล้ววันนี้ในราคา 4,900 บาท ที่ G-SHOCK CASIO ทุกสาขา หรือดูรายละเอียดเพิ่มเติมที่ casio.com/th/ . #CASIO #GSHOCK #NEWTRO #dw6900tr #MDsSTYLE #MenDetails_TH	https://www.mendetails.com
Η κα Αγγελική Καμβύση, είναι Πρόεδρος του Συλλόγου ΑΜΕΑ Νάξου. Ένας Σύλλογος που βασίζεται στη θέληση και το πείσμα των μελών του Δ.Σ, ένας Σύλλογος που παίρνει δύναμη από το χαμόγελο των παιδιών με αναπηρία. Η δράση του Συλλόγου είναι πολύ σημαντική μέσα από τη λειτουργία κέντρου ειδικών θεραπειών, την πραγματοποίηση εκδηλώσεων, την ενημέρωση γονέων σε θέματα αναπηρίας, την οργάνωση εκπαιδευτικών εργαστηρίων για παιδιά, την οικονομική ακόμα στήριξη οικογενειών. Το όραμα του Συλλόγου ακόμα μεγαλύτερο για το μέλλον, με προτάσεις προς τους αρμόδιους φορείς για τη δημιουργία υποδομών που να είναι φιλικές στα άτομα με αναπηρία, με την πραγματοποίηση περισσότερων εκπαιδευτικών εργαστηρίων, με τη δημιουργία κέντρου διημέρευσης και ημερήσιας φροντίδας κ.ο.κ.. Το ζητούμενο είναι η στήριξη όλων όσων μπορούν να προσφέρουν ο καθένας από το δικό του μετερίζι. Είτε μιλάμε για την τοπική αυτοδιοίκηση ή γενικότερα τους φορείς του τόπου, είτε μιλάμε για τον καθέναν από εμάς που μπορεί να αφιερώσει χρόνο και εθελοντικά να προσφέρει στο έργο του Συλλόγου. Το μόνο σίγουρο είναι ότι από τους συνανθρώπους μας με αναπηρία μπορούμε να διδαχτούμε και να λάβουμε πολύ περισσότερα από αυτά που θα δώσουμε. Η δύναμη της ψυχής τους είναι δίδαγμα για όλους μας για τον αγώνα της ζωής. Ο Σύλλογος θα πραγματοποιήσει την εκλογική διαδικασία αυτή την Κυριακή 16 Μαρτίου στις 17:00 στο υπόγειο του ναού του Αγίου Νικοδήμου στη χώρα της Νάξου. *Η κα Αγγελική Καμβύση φιλοξενήθηκε στη συχνότητα της Ραδιοφωνίας Κυκλάδων 101,3 και στην εκπομπή "Cyclades On Air"	https://naxospress.gr
最近のおうちの幸せあれこれ。 今のおうちに引っ越して早くも半年が過ぎました🪴 友人が遊びにくるタイミングでちょうど片付けをしたので、いまだ！と思ってお部屋の写真を撮ってみたものの。 いざ客観的にみてみると、あれ、物が少ない？なんだか寂しい…？ と、新たな視点を得ることになりました。 ダイニングテーブルの上って、やっぱりお花や高さのあるものがあった方が素敵なのかしら、とか。 いやー、インテリアって奥が深いです。 足りているようで、足りていないようにも思えて、でも足りている気もする・・・。 自分にとって「足る」とはなんなのかを思わず考えてしまいます。 でも、これが"今"のわたしのおうちってことで、記録しておくことにする！ それにしても、今のおうち、大好きです。 真っ白な明るい空間でもなく、モノトーンなわけでもなく、ダークで深いトーンにまとめたくて試行錯誤した家具や雑貨たちはどれも丁寧に選んだものなので大満足。 家で仕事をすることが多いので、1人にしては大きめのダイニングテーブルを選んだのは大正解でした。 目黒のACME Furnitureで購入したのですが、足元のデザインがちょっとロックでかっこよくないですか？少しスパイスがあった方が好きなタイプなので、一目惚れでした。 椅子も、あえて揃えずに2種類全く違ったデザインをチョイス。 手前にある椅子、すごくふかふかで安定感もある大きさで、気持ちいいんです。 ベッドは急いでいたので、楽天のネルコンシェルジュでポチッと。 色味は不安だったけど、テーブルとぴったりの落ち着いた木目調で安っぽさもなく、いい感じにまとまりました。 ただ、組み立ては結構大変w早朝から父親を駆り出してしまい、老体に鞭を打たせてしまった…（老体とか言ったらまた怒られそうw） そしてなんといっても鏡台！！ スキンケアやメイク、美容が大好きな私にとって鏡台は欠かせなかったので、すこし場所をとったとしても良いものを、と思いLOWYAで購入。 これ、本当に素敵なので心からお勧めしたい。 色味がとってもあたたかいし、作りもしっかりしているし、脚のデザインもかわいいのなんのって。 収納力はあまりないけれど、そんなことどうでもいいくらい可愛くてお気に入りです。 ここでお風呂上がりにパックしながら髪を乾かしたり、朝メイクしたり、大切なスイッチの切り替え場所。 …こだわりを語り出したらキリがなさそうなのでこの辺で。。。 家電もかなりこだわったので、今度ご紹介します🫶 (絶対インスタに書く長さじゃないよね) #インテリア #ひとりぐらしインテリア #一人暮らしインテリア #1kインテリア #10畳インテリア #木目調 #interiordesign #お部屋作り #ひとりぐらし部屋 #ひとり暮らしインテリア	https://www.low-ya.com

essential because the presence of an `alt` or `aria-label` attribute does not guarantee usefulness. Labels such as `button`, `file1`, or `image1` may satisfy automated checks but provide no semantic value to screen reader users. We define eleven categories of uninformative accessibility text that are excluded from our analysis:

1. **Emoji:** Symbols skipped or mispronounced by screen readers.

2. **Too Short:** Strings below a language-specific character threshold (1 character for CJK scripts and 3 for others) rarely convey meaning (e.g., “go”, 图).

3. **File Names:** Asset references with extensions such as .jpg, .png, or .svg (e.g., “banner_img123.jpg”).

4. **URLs/Paths:** Raw links or file system paths provide no semantic context (e.g., “/assets/logo.svg”, “https://ex.com/a.png”).

5. **Generic Actions:** Standalone UI verbs without context add little value (e.g., “search”, 닫기).

6. **Placeholders:** Generic media or UI terms give no detail (e.g., “icon”, 图像).

7. **Developer Labels:** Internal IDs or class names not meaningful for users (e.g., “btn-submit”, “nav_menu”).

8. **Label Numbers:** Numbered labels typically lack descriptive content (e.g., “image 1”, “slide 3”).

9. **Single Words:** Generic words that are ambiguous in isolation, e.g., “photo”, “submit”.

10. **Mixed Alphanumerics:** Tokens reflecting program logic (e.g., “img123”, “icon2”).

11. **Ordinal Phrases:** Pagination or slider counts provide structure but not meaning (e.g., “2 of 10”, “1 of 3”).

Although in some cases a single word can be sufficiently descriptive, for example in labels on `button` elements (“Search”, “Close”) or `select` menus (“Language”), our dataset shows that these cases are rare. Only 7.6% of `button` labels and 15% of `select` labels consisted of single words. In contrast, the overwhelming majority of single-word entries appeared in `img-alt` attributes, where a single word (e.g., “photo”, “image”) typically fails to convey meaningful information about the image content. As shown in Figure 5.3, most single-word cases occur in `img-alt` rather than in interactive elements. To avoid overestimating accessibility quality, we conservatively filter all single-word labels across elements. Any accessibility text not matching the above patterns is retained and considered `useful` for analysis. This filtering process distinguishes meaningful metadata from boilerplate or autogenerated labels, enabling more accurate assessments of multilingual accessibility practices.

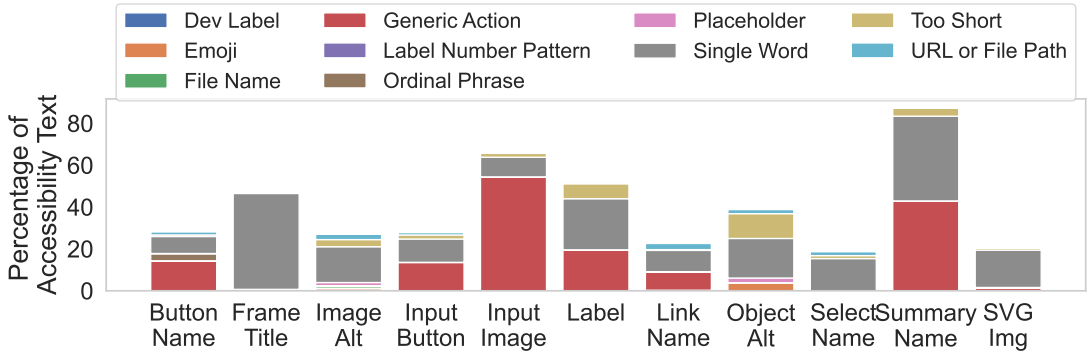


Figure 5.3: Discarded accessibility texts by HTML element and category. General action labels and single word entries are the most common issues across different elements.

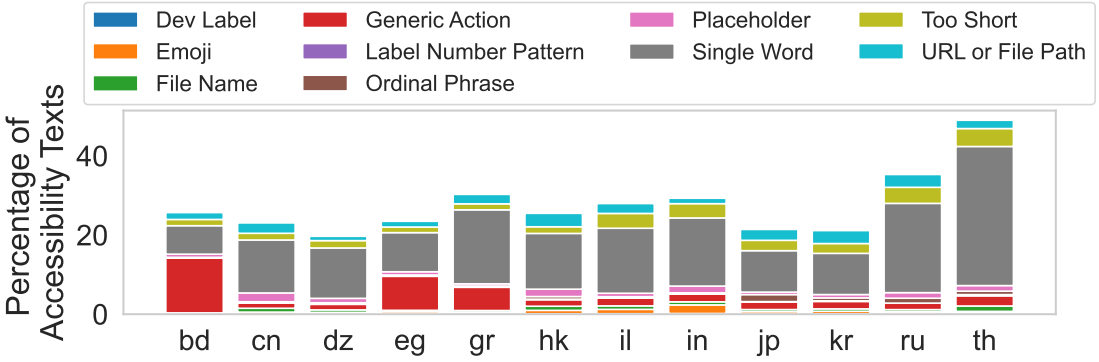


Figure 5.4: Distribution of discarded accessibility texts by category across countries. Bars show the percentage of texts filtered out as uninformative (e.g., single words, placeholders, file names, URLs).

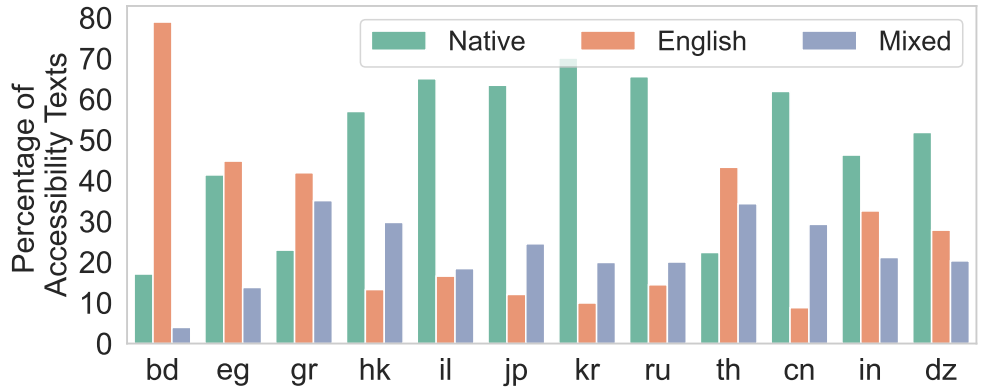


Figure 5.5: Language distribution of filtered accessibility texts across countries. Only texts classified as potentially useful are included, grouped into native (target) language, English, and mixed. The figure shows that English dominates accessibility metadata even in regions where visible content is overwhelmingly in the native script.

Figure 5.4 shows the percentage distribution of filtered accessibility texts across countries and categories. One of the most common issues is the use of generic single words. For example, in Thailand, over 33% of accessibility texts are single-word labels. High rates are also observed in Russia (22.2%), Greece (18.03%), and India (17.1%). In contrast, countries like Bangladesh (6.9%) and Egypt (10.5%) show lower proportions. A small but non-negligible portion of texts are too short to convey meaning. In Russia, 4.26% of labels fall into this category, followed by Thailand (4.24%), Israel (4.03%), and India (3.6%). Some websites also use raw URLs or file paths as labels. This affects 3.8% of labels in Hong Kong, 3.5% in South Korea, and 3.17% in Russia. These findings show that even when accessibility text is present, it often lacks descriptive content. This reinforces the need to go beyond presence-based metrics and assess the actual semantic value of accessibility labels.

Language Distribution of Informative Accessibility Text: After removing uninformative and placeholder accessibility text, we reanalyze the remaining content to understand the language distribution of texts that are potentially useful. This filtered set reflects more intentional and meaningful uses of `alt`, `label`, and other accessibility attributes. Figure 5.5 shows the proportion of accessibility texts written in native languages, English, or a mix of both, across the 12 analyzed countries. A prominent pattern is the heavy reliance on English, even in countries where it is not the primary language. In Bangladesh, 79% of informative accessibility texts are in English, the highest among all countries analyzed. Egypt, Thailand, and Greece also show a strong tendency to default to English. This suggests that developers may rely on English for accessibility metadata due to limited localization tools or being unaware of screen reader needs in native languages.

Another important pattern is the use of mixed-language accessibility hints, where a single `alt` attribute contains both the native language and English. This occurs frequently in Greece (35%), Thailand (34%), and Hong Kong (30%), and in over 20% of websites in China, Russia, Japan, and India. While sometimes intended to aid

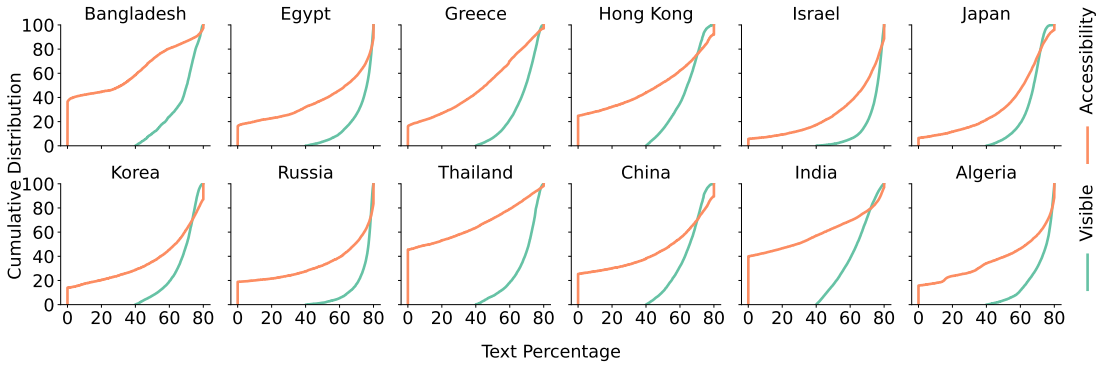


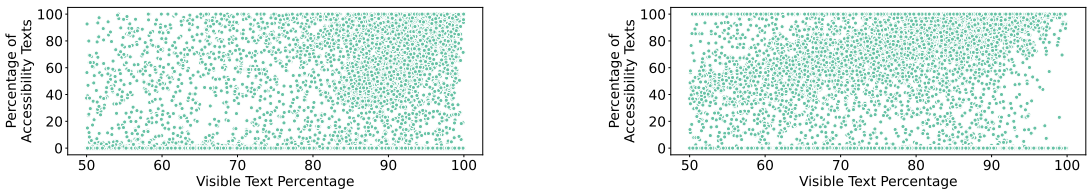
Figure 5.6: CDFs of native language usage in visible vs. accessibility text across countries. Most websites with visible content in native languages still rely on English in accessibility metadata.

multilingual users, such mixing often confuses screen readers, which typically do not handle language switching within a single label, resulting in mispronunciations or reduced clarity.

5.5 Language-Aware Accessibility

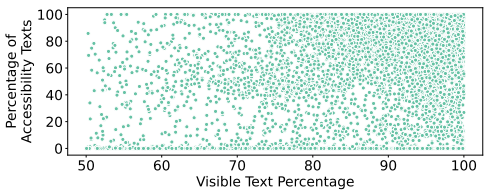
Mismatch Between Visible and Accessibility Text: Figure 5.6 compares native language usage in visible versus accessibility text across the 12 analyzed countries. While the visible content of many websites is multilingual or predominantly in the native language, the associated accessibility metadata, such as `alt` text, `aria-labels`, and form labels, is typically written in English. This mismatch is especially noticeable in countries like India and Bangladesh, where over 40% of websites have less than 10% of their accessibility text in the native language. Thailand, China, and Hong Kong also show similar trends, with more than a quarter of their websites falling into this category. In contrast, countries like Japan and Israel have significantly lower rates of mismatch, with fewer than 10% of websites showing such disparities. For blind users who rely on screen readers, this language discrepancy introduces an additional barrier, forcing them to navigate a bilingual interface where visible content and assistive text do not align.

To better understand how these aggregate patterns manifest at the level of individual websites, we further analyze country-level scatter plots (Figure 5.7) that visualize the relationship between native-language usage in visible content and accessibility metadata. In these plots, each point represents a website, with the x-axis indicating the percentage of visible content in the native language and the y-axis indicating the percentage of accessibility text in the same language. These plots provide a more granular view of the mismatches observed at the aggregate level. For instance, the scatter plot for Thai websites (Figure 5.7k) reveals a dense cluster in the bottom-right region, corresponding to websites whose visible content is almost entirely in Thai but whose accessibility metadata contains little or no Thai. In contrast, points in the top-right region represent websites with consistent native-language usage across both visible and accessibility content.

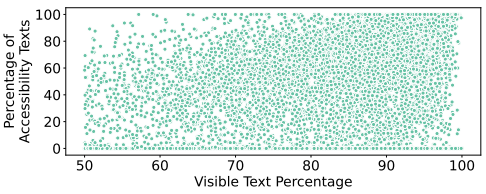


(a) Bangladesh

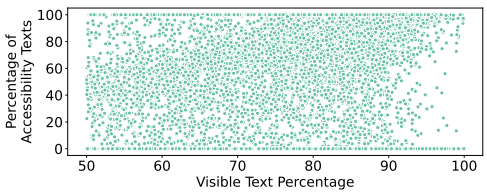
(b) China



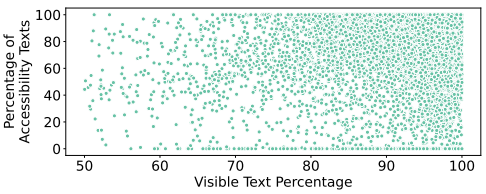
(c) Egypt



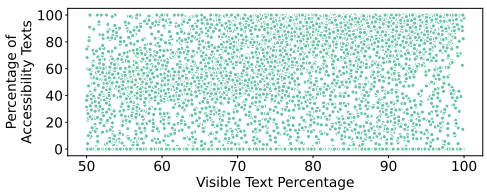
(d) Greece



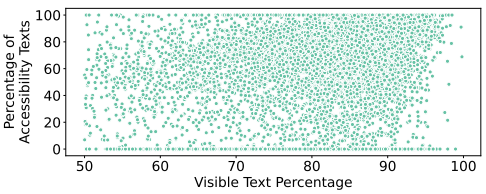
(e) Hong Kong



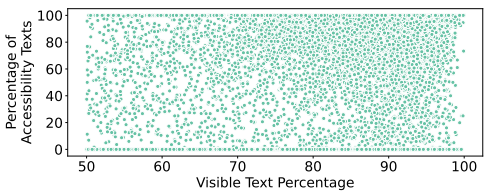
(f) Israel



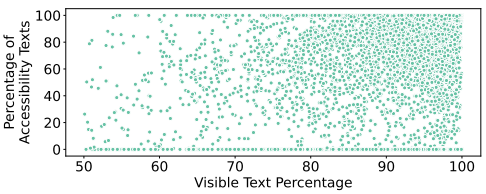
(g) India



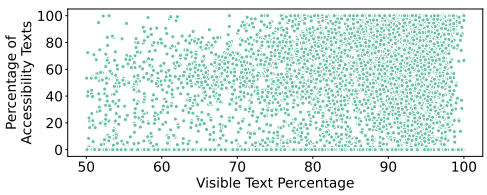
(h) Japan



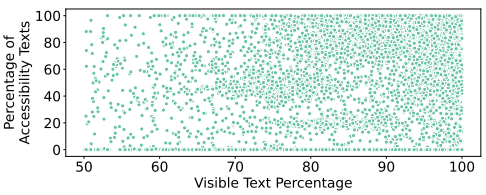
(i) South Korea



(j) Russia



(k) Thailand



(l) Algeria

Figure 5.7: Scatter plots showing the percentage of native-language usage in visible text versus accessibility text for websites across twelve countries. Each point represents a single website. The plots illustrate the degree of alignment or mismatch between the language of visible content and the corresponding accessibility metadata.

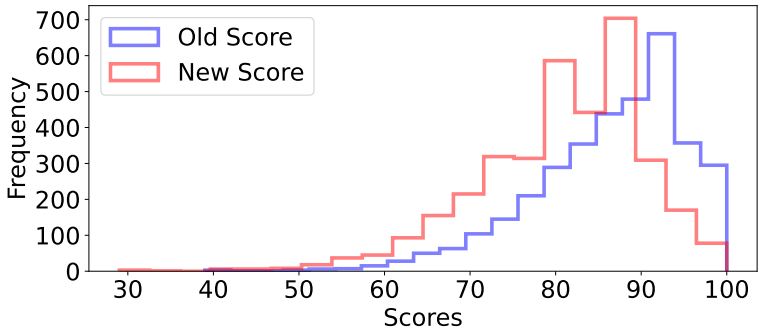


Figure 5.8: Accessibility score distribution before and after applying Kizuki's language-aware alt text evaluation on websites from Bangladesh and Thailand.

These patterns are also evident in concrete, real-world examples. In Bangladesh, `https://teachers.gov.bd` is a widely used government education portal. Although more than 98% of its visible content is in Bangla, only one of the 79 images with `alt` attributes uses Bangla. Comparable patterns are also seen in India, Thailand, and China. The Indian website `https://cmhelpline.mp.gov.in` has a Hindi version where the interface is almost entirely in Hindi, but all accessibility text is in English. The Thai news website `https://www.khaosod.co.th` contains over 92% of its visible content in Thai, but its accessibility labels are mostly in English. The Chinese provincial government site `https://kjt.shaanxi.gov.cn` is almost fully in Chinese, yet its accessibility texts are entirely in English. Table 5.5 illustrates examples of accessibility mismatches on six websites across Bangladesh, India, Thailand, Egypt, China, and Hong Kong.

Adding Language Awareness to Lighthouse: Automated testing tools such as Lighthouse do not consider the language of accessibility text when evaluating compliance. As a result, alt attributes are marked as present regardless of whether their content matches the language of the surrounding interface. To address this limitation, we introduce **`Kizuki`**, a Lighthouse extension that incorporates language awareness in accessibility evaluation. Specifically, we extend the audit for image alt text to verify whether the description is written in the same language as the page's visible content.

We evaluate `Kizuki` on 10,000 websites from Bangladesh and Thailand, two countries where language mismatch between visible content and accessibility metadata is particularly common. For fairness, we exclude websites that fail the original Lighthouse test due to missing alt attributes. Figure 5.8 shows the resulting shift in accessibility scores. Without considering language, 43% of websites received a Lighthouse score above 90 (considered "good" [140, 147]), and 5.6% achieved a perfect score. After applying Kizuki's language-aware check, these numbers dropped significantly: only 15.8% of websites scored above 90, and just 1.8% retained a perfect score.

Table 5.5: Illustrative examples of accessibility mismatches on six websites across Bangladesh, India, Thailand, Egypt, China, and Hong Kong.

Link: https://developer.nvidia.cn

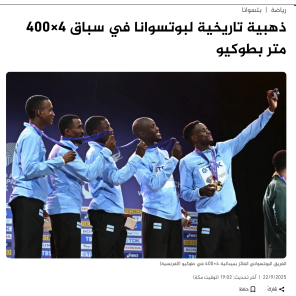
Alt text: "The image depicts a stack diagram highlighting NVIDIA NIM Operator, a Kubernetes Operator that is designed to facilitate the deployment, management, and scaling of NVIDIA NIM microservices on Kubernetes clusters."

Link: https://goodmoneybygsb.com

Alt text:"A hand is holding a smartphone displaying the "GOOD MONEY" app by GSB. The screen shows the app's logo, featuring a white "G" on a gradient background transitioning from pink to orange."

Link: https://www.jagonews24.com

Alt text:"Three people were killed in a head-on collision between an ambulance and an engine-powered van in Jhikargachha, Jessore. Another person was injured in the incident. Photo: Milan Rahman"



Link: https://www.ajnet.me

Alt text: "(From L) Gold medallists Botswana's athlete Busang Collen Kebinatshipi, Botswana's Leungo Scotch, Botswana's athlete Bayapo Ndori, Botswana's athlete Letsile Tebogo and Botswana's athlete Lee Bhekempilo Eppie pose on the podium after the men's 4x400m relay final during the World Athletics Championships in Tokyo on September 21, 2025."

Link: https://hindi.cricketaddictor.com

Alt text: "Preity Zinta's team Saint Lucia Kings won the 1st title of CPL 2024 under the captaincy of Faf du Plessis"

Link: https://www.aia.com.hk

Alt text: "Lovely Asian girl and mother celebrating birthday with friends / family by having a virtual birthday party at home. Enjoying birthday celebration in front of the laptop during pandemic. Practicing social distancing. Birthday lifestyle theme."

5.6 Data Availability

To support transparency, reproducibility, and further research, we have released the artifacts from this chapter as open-source software.
1. Kizuki Extension
The source code for the Kizuki extension and usage instructions are available at:

https://github.com/kal-purush/accessibility_study

2. LangCrUX
The complete dataset comprising accessibility reports for 120,000 websites is available at:

https://github.com/kal-purush/LangCrux

3. Interactive Visualization
An interactive platform for exploring the dataset and language distributions is hosted at:

https://kal-purush.github.io/LangCrux

5.7 Summary

Multilingual web content is becoming increasingly prevalent, yet accessibility support continues to lag behind, particularly for non-Latin scripts. In this chapter, we introduced LangCrUX, the first large-scale dataset of 120,000 popular websites across 12 languages that primarily use non-Latin scripts. Using this dataset, we conducted a systematic analysis of multilingual accessibility practices and identified widespread language mismatches, missing or inconsistent accessibility annotations, and gaps in current automated evaluation tools. To address these limitations, we presented Kizuki, a Google Lighthouse extension that augments existing accessibility audits with language-consistency checks across user-facing content. By explicitly accounting for language diversity, Kizuki enables more accurate detection of accessibility issues that are otherwise overlooked in multilingual settings. Together, these contributions provide empirical evidence of accessibility challenges on the multilingual web and offer practical tools to support more language-aware accessibility evaluation. We hope these resources will enable the community to better understand and address the accessibility challenges of an increasingly multilingual web.

6

The Role of Multilingualism in Open-Source Development: A Longitudinal Study of Collaboration and Friction

In the previous chapter, we examined how linguistic assumptions shape end users' experience on the web, showing how language influences usability and inclusion in non-English and non-Latin contexts. In this chapter, we shift the focus from the *consumption* of software to its *production*, and examine how the same linguistic forces shape participation, coordination, and visibility in open-source software (OSS) communities. By doing so, this chapter addresses the intersection of demographic diversity (challenge C_1) and Unicode adoption (challenge C_2), and asks a central question: how does language influence who can effectively contribute to modern software ecosystems? More broadly, this transition reflects the core theme of the second part of this dissertation: software systems are socio-technical, and language operates as a structural factor that shapes both participation and collaboration.

Open-source software (OSS) underpins much of modern software engineering, yet its collaborative practices have historically centered on English as a de facto lingua franca. While this convention facilitates coordination at scale, it also privileges Anglophone developers and embeds implicit linguistic norms into contribution practices. As global participation expands, these monolingual assumptions are increasingly challenged. Improved support for non-Latin scripts through Unicode has enabled developers to use their native languages across a wider range of artifacts, from issue discussions to code comments, raising new questions about how linguistic diversity interacts with collaboration, visibility, and governance.

In this chapter, we investigate the role of multilingualism in open-source development through a longitudinal study of collaboration and friction. We analyze 9.14 Billion GitHub issues, pull requests, and discussions, alongside 62,500 repositories across 5 programming languages and 30 natural languages, covering the period from 2015 to 2025. We track changes in language use across communication, code, and documentation to assess whether open-source development is becoming more linguistically inclusive as participation broadens.

Our findings reveal a complex trajectory. We observe a steady increase in multilingual participation, particularly in languages such as Korean, Chinese, and Russian, not only in social interactions but also in technical artifacts such as code comments and string literals. At the same time, increased linguistic diversity introduces new forms of friction. We find that the ability to communicate in a native language can conflict with established norms of English standardization, leading to a form of *linguistic siloing*: non-English or multilingual projects tend to receive less external visibility and attract fewer contributors. This suggests that while OSS communities are becoming more demographically diverse, language remains a structural barrier that can fragment collaboration along linguistic lines.

This chapter shares material with the corresponding publication [P4].

6.1 Motivation

The open-source software (OSS) ecosystem has been a cornerstone of modern software development, fostering collaboration among developers worldwide. Historically, English has served as the lingua franca for OSS, dominating codebases, documentation, and community interactions. Platforms like Stack Overflow require all questions and answers

to be in English [456], and most discussions on GitHub also happen in English. This dominance is not accidental. It comes from the early days of computing, when research and development were led by the United States and United Kingdom, and the first widely used programming languages were created in English [313]. As a result, developers everywhere learned to use English to write code and work with software tools, no matter their native language.

Over the years, software development tools and systems have improved to support many writing systems. The release of Unicode 3.0 in 1999 made it possible to encode and display non-Latin scripts more reliably. As a result, people today often use scripts such as Chinese, Arabic, Cyrillic, and Bengali across digital writing, websites [P3], and software projects. Web technologies have followed this trend, and most modern browsers and development tools now support a wide range of scripts. Ebbertz et al. showed English made up 80% of web content in 1998, but by 2002 this had dropped to 56% [149]. More recent estimates place the figure between 20% and 50% [387, 389]. At the same time, the global landscape of software development has changed. Nearly a quarter of the open source community reports limited English proficiency[1]. GitHub's own growth data also shows that many of the fastest-growing contributor communities are in countries where English is not the main language, including China, Brazil, India, and Russia [188, 189, 118]. These changes raise important questions about the role of English in online collaboration.

Prior research provides limited insight into the linguistic dynamics of open-source software development. Pawelka et al. [380] analyzed natural language use in comments and identifiers across 23 software projects. They found that while industry projects contain some non-English text, open-source projects contain almost none. Their findings point to a strong English-language bias in public code repositories. To the best of our knowledge, this remains the only study that has examined the presence of non-English languages in code repositories. However, their study does not explore whether this trend has evolved over time.

Beyond the single study on non-English content in code, others have extensively studied how language affects collaboration and participation, especially for newcomers. Yoseph et al. identified language-related challenges as the third most commonly reported bias faced by contributors in open-source communities [11]. Steinmacher et al. reported that limited English proficiency is one of the major barriers preventing newcomers from communicating or participating in discussions in open-source projects [466]. Similarly, Balali et al. found that differences in language fluency can negatively affect mentoring relationships, making onboarding more difficult both socially and technically [35]. However, these findings are based on limited samples and also do not track longitudinal changes or large-scale usage across repositories. GitHub also publishes annual Octoverse statistics [118, 189] that track demographics, programming language trends, and regional growth. While useful, these reports do not analyze the natural languages used in issues, pull requests, documentation, or code. Demographics can overlap with linguistic patterns, but the relationship is not linear: multiple regions share the same language (e.g., Arabic, Spanish), and many countries have several official languages (e.g., India). As a result, demographic counts alone do not indicate the languages developers actually use on the

[1] https://opensourcesurvey.org/2017/

platform.

Despite these insights, there is still a gap in the literature on how language use is evolving in open-source projects over time. A key question remains: How is it changing across code, discussion, and documentation? In this chapter, we present the first large-scale empirical study of multilingualism in open source software development. We analyze a dataset of 9.14 Billion GitHub messages, including issues, comments, pull requests, reviews, and discussions, collected from 2015 to May 2025. To examine language use in code, we also study 62,500 repositories over the same period. These repositories span five widely used programming languages: JavaScript, Python, TypeScript, Java, and C#. Using language detection and classification tools, we identify the presence of 30 natural languages across different parts of open source collaboration, including code elements such as identifiers and string literals, documentation, and user-generated messages. In particular, our study answers the following six research questions:

- *RQ1: Is developer discussion in open-source becoming more multilingual?* Our results show that non-English content in issues and pull requests grew by 164 percent between 2015 and 2025.
- *RQ2: Which natural languages are most prevalent in open source interactions?* Korean usage increased by 1706 percent, and Chinese by 120 percent. Along with Japanese, Russian, and Vietnamese, they lead non-English content on GitHub.
- *RQ3: Is the code itself, such as string literals and identifiers, becoming more multilingual?* Yes, mostly in comments and literals. Non-English comments rose from 3.6 to 11.9 percent, and literals rose from 3.2 to 9.6 percent. Identifiers stayed mostly English.
- *RQ4: Are there significant differences across programming languages?* Yes. Java, Python, and JavaScript show more non-English use than C# and TypeScript. Java has the highest share of multilingual comments and literals.
- *RQ5: Are there repositories where developers regularly use multiple languages? Are there signs of language-based friction or coordination issues?* We found 193,000 repositories using multiple languages. But language often creates tension for non-English participants.
- *RQ6: Does non-English content negatively affect software development practices?* Yes. English-dominant projects tend to receive more visibility, collaboration, and participation.

6.2 Related Work

Prior work on multilingualism and open-source development has examined natural language in code, the impact of language barriers on collaboration [467, 395], communication practices on platforms like GitHub [135, 414], and how language shapes inclusivity [327, 471, 440]. Researchers have also studied user behavior in forums [240], social and collaborative structures [29], and diversity in developer communities [507, 482].

These works highlight barriers and individual practices, but few analyze multilingualism at scale or over time. Most are qualitative or focus on isolated aspects. In contrast, our study offers a longitudinal, large-scale analysis of multilingual trends across code, documentation, and discussion.

Language Barriers in Collaboration: English proficiency is a recurrent barrier for contributors, especially newcomers [159, 156, 556, 356, 468, 467, 395]. Steinmacher et al. [468] identified language as a consistent challenge for new contributors. Balali et al. [35] found that differences in English fluency hinder mentoring and communication. In global software teams, Noll et al. [356] showed that non-native speakers face difficulties in meetings, documentation, and cross-cultural understanding, affecting coordination and productivity. These studies focus on onboarding or team settings, whereas our work examines multilingual trends in open source over time.

Language in Developer Discussions: A growing body of research explores how language affects communication dynamics on developer platforms such as GitHub and Stack Overflow [270, 222, 46, 47, 72, 371, 135, 414]. Kavaler et al. [270] linked higher linguistic complexity in issues to slower resolution. Guo et al. [222] showed that non-native speakers struggle with documentation, API names, and informal expressions. Bregolin et al. [72] found that native-language Stack Overflow sites increase communication effort and acceptance rates without harming the English site. Becker et al. [47] showed that technical or unnatural English in error messages challenges both non-native and native speakers. These works target specific interactions, not long-term multilingual trends.

Language and Toxicity in Open Source: Language use is also deeply tied to communication dynamics in OSS. Miller et al. [327] studied toxicity in GitHub discussions and found that non-responsiveness and ambiguity in language can escalate into negative exchanges. Sultana et al. [471] linked toxic and discriminatory communication to decreased diversity in OSS participation. Sarker et al. [439, 440] developed automated methods for detecting toxicity in code reviews, illustrating how linguistic cues can signal exclusionary behavior.

Platform Reports on Developer Activity: GitHub publishes the Octoverse report every year to summarize platform-wide activity, highlighting trends in programming languages, repository growth, and the geographic distribution of developers [118, 189]. Stack Overflow also releases similar annual Developer Survey [457], focusing on developer demographics, technologies, and work environments. These reports provide valuable aggregate statistics about participation and technology adoption, but they emphasize technical and demographic dimensions rather than the linguistic composition of developer activity. They do not examine how language is used within discussions, documentation, or source code, and therefore overlook the multilingual characteristics that shape collaboration and accessibility in open-source projects. In particular, while Octoverse tracks regional growth in countries such as India, Indonesia, and Brazil, it does not assess whether this expansion is accompanied by greater linguistic diversity within repositories or discussions. Our study addresses this methodological gap by introducing large-scale, language-aware measurements of communication and code that reveal trends beyond the scope of aggregate platform reports.

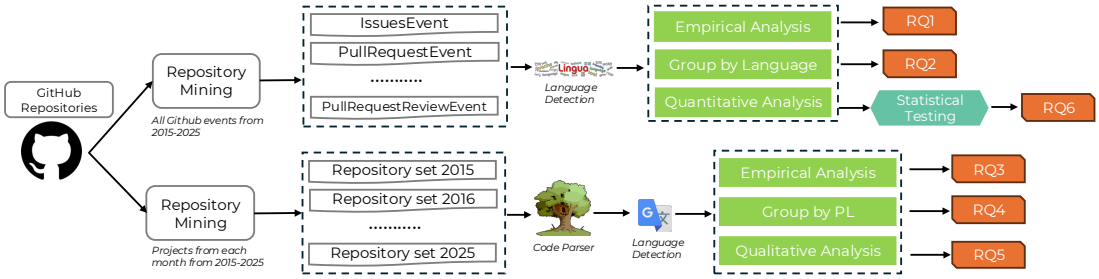
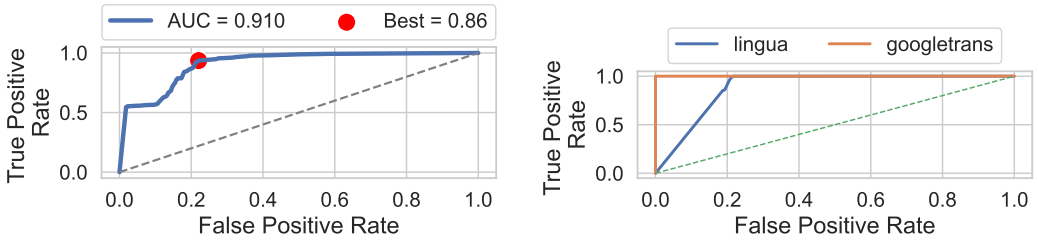


Figure 6.1: Overview of our methodology: data collection, language detection, and analysis across code and discussions.

6.3 Study Design and Methodology

To examine the role of multilingualism in open-source software development, we analyzed GitHub activity from January 2015 to May 2025. Open-source platforms contain multiple forms of natural language content. Some elements facilitate communication among developers, such as issue descriptions, pull request discussions, and review comments, while others, like commit messages, are transactional and not central to this study. Because the available data sources differ in scope and scalability, our approach combines two complementary analysis pipelines, each aligned with specific research questions. The first pipeline (used for **RQ1**, **RQ2**, and **RQ6**) focuses on communication-related activity drawn from GHArchive. This dataset provides large-scale event metadata, including issues, pull requests, and associated comments. The second pipeline (used for **RQ3**, **RQ4**, and **RQ5**) targets source code and documentation, analyzing identifiers, literals, and code comments. Since GHArchive does not include source code content, we use the GitHub REST API to retrieve code patches. API access is rate-limited and less scalable, so this dataset covers a representative, temporally balanced sample of repositories rather than the full population. Figure 6.1 summarizes both pipelines and the overall data flow. For RQ6, we also performed statistical testing to assess whether the observed differences across language categories are significant. Given the skewed nature of repository activity data, we used non-parametric methods suitable for comparing multiple groups. We report overall group differences and pairwise comparisons where relevant.

All analyses were performed on a dedicated Linux server equipped with an AMD EPYC 7H12 processor (128 physical cores, 256 threads), 2 TB of RAM, and 60 TB of storage. Processing the entire dataset required several weeks and included the classification of over 9.14 Billion GitHub messages and the analysis of 62,500 sampled code patches. The pipeline covered metadata extraction, text preprocessing, and natural language identification across both communication and code elements. In the following sections, we present the detailed methodology and results for each research question. Each section explains the specific subset of data used, the metrics derived from it, and the statistical analyses applied, followed by quantitative findings and interpretation.



(a) Lingua ROC curve.

(b) Lingua vs Google Translator.

Figure 6.2: Evaluation of language detection models: (a) ROC curve and threshold selection for `Lingua`; (b) performance comparison with Google Translator on short code texts.

6.4 RQ1: Is open source development becoming more multilingual?

This research question examines whether open-source development has become more multilingual over time and how the linguistic composition of GitHub communication has changed.

Data Collection We analyzed all available GitHub discussions from January 2015 to May 2025, including issues, pull requests, and their associated comments and reviews. This resulted in a dataset of approximately 9.14 Billion messages, drawn from over 476 million repositories and spanning contributions from 78.01 million developers. The data was collected from GHArchive, a public record of GitHub activity that tracks user interactions and event metadata over time. We used the May 2025 release of GHArchive and processed a total of 4.3 TB of compressed JSON data. To extract full message content, we parsed it directly from GHArchive event records, which include both the message text and relevant metadata for each interaction. We focused on five types of user interactions where natural language is commonly used: creating and commenting on issues, opening and reviewing pull requests, and posting comments during code reviews. These represent the main communication channels used by developers on GitHub.

Language Detection We used the `Lingua`[2] language identification library (v2.1.1) to classify the natural language of each message. After filtering for relevant message types, we retained 1.7 Billion data points containing natural language text. Language detection on GitHub data is challenging because many messages are short, contain typos, or include unformatted code, which can confuse classifiers. To reduce these errors, we used Lingua's strictest detection mode and excluded all messages with low-confidence results. We evaluated the classifier's performance using a manually annotated sample of messages and computed the receiver operating characteristic (ROC) curve shown in Figure 6.2a. The model achieved an area under the curve (AUC) of 0.91, with the optimal confidence threshold at 0.86. To remain conservative and minimize false

[2] `https://github.com/pemistahl/lingua-py`

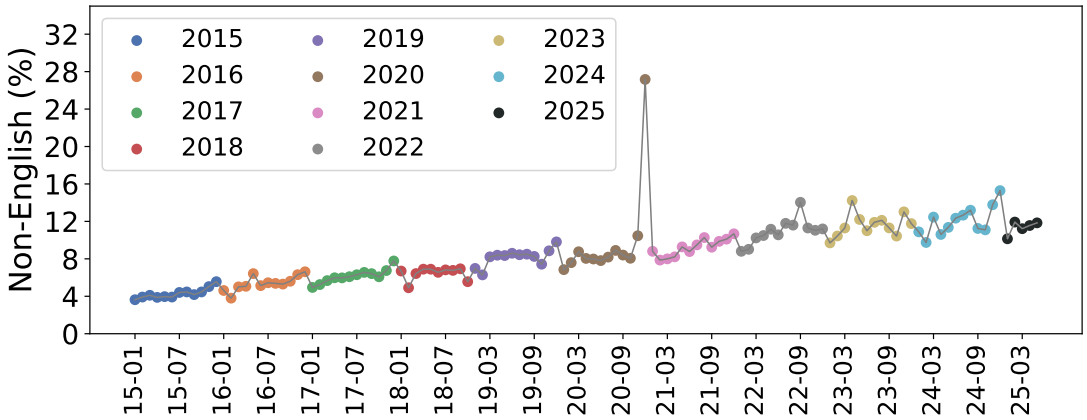


Figure 6.3: Monthly percentage of non-English content in GitHub discussions (issues, pull requests) from 2015 to 2025, showing a consistent upward trend and increased multilingual participation in open-source development.

positives, we set our cutoff slightly higher, at 0.9. Messages below this confidence level were excluded from the analysis and from the percentage calculations. Lingua supports 75 languages, and we accepted any message with a high-confidence match, while focusing our analysis on non-Latin scripts and non-English text.

Results As shown in Figure 6.3, the proportion of non-English content increased consistently over this period, rising from an average of 4.2% in 2015 to 11.3% in 2025, representing a 164% relative increase over the decade. Monthly data show a steady upward trajectory, with notable acceleration after 2019. In 2019, the average proportion of non-English content was 8.2%, increasing to 9.9% in 2020, marking a structural shift in participation dynamics. From 2022 onward, the percentage of non-English messages remained above 10% annually, signaling a lasting change in the linguistic makeup of open-source communities.

Observations A few notable patterns emerge. First, there is a sharp spike in December 2020, where non-English content briefly reaches 27.1%. Upon manual inspection of the underlying data, we found that this spike corresponded to a sudden surge in issues containing Chinese text, posted across multiple repositories. These issues accounted for a large volume of content in that month. However, none of the issue links are currently accessible, suggesting that the repositories or issues may have been deleted, made private, or removed from GitHub. The content of these issues points to a coordinated spam attack related to promoting WeChat groups. Many messages contained phrases such as “澳洲幸运10信誉公众号微信群” (“Australia Lucky 10 trusted public WeChat group”), “极速赛车微信群游戏” (“Speed racing WeChat group game”), and “比较信誉极速赛车微信公众号平台” (“Compare trusted speed racing WeChat public account platforms”). The repeated references to WeChat, group promotions, and gambling-like keywords suggest the posts were meant to advertise or exploit access to Chinese-language WeChat groups.

We observed a similar anomaly on May 1, 2016, when a large number of issues

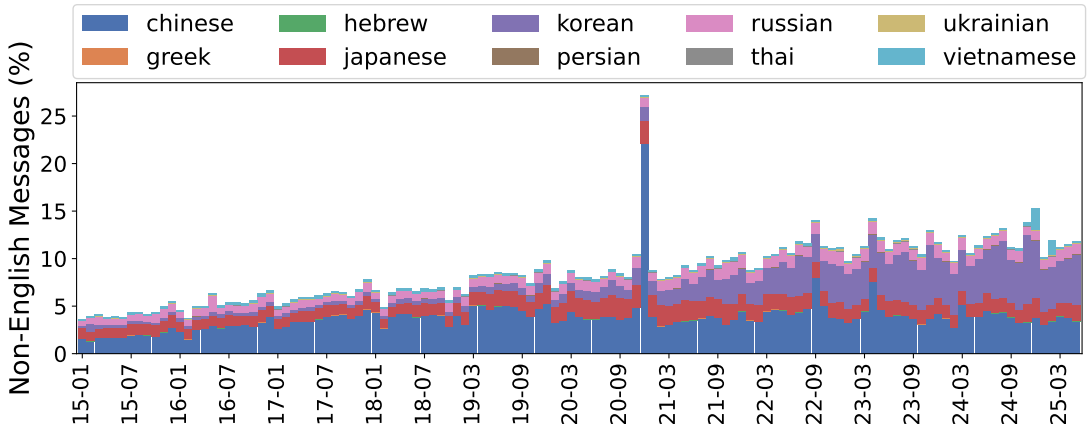


Figure 6.4: Monthly distribution of non-English natural languages in GitHub discussions from 2015 to 2025, highlighting the sustained presence of Chinese, Japanese, Russian, Korean, and Vietnamese, and the growing participation in Korean, Persian, and Vietnamese over time.

containing Russian text appeared across several repositories. These entries likewise drove a temporary increase in non-English content. As with the 2020 spike, the original links to these issues are no longer available. While both events appear to be isolated, they reveal that even attackers assume some level of language diversity and inclusivity. They adapt their attacks to the target audience's language, leveraging localized communication to increase effectiveness. Second, the persistence of double-digit non-English percentages from 2022 onward suggests a real decline in English dominance. For example, in January 2015, 3.6% of issue comments were non-English. By May 2025, that number rose to 11.8%. It is important to note that these values are percentages, not absolute counts. GitHub activity has grown significantly over time, but the use of percentages accounts for that underlying growth.

6.5 RQ2: Which natural languages are most prevalent in open-source interactions?

This research question identifies the most commonly used non-English natural languages in open-source discussions on GitHub and how their prevalence has evolved over time. We use the same dataset and methodology described in §6.4, based on communication data from GHArchive between January 2015 and May 2025.

Results Figure 6.4 shows the distribution of natural languages across all GitHub messages over the ten-year period. The top five languages by average share across the 10-year period are Chinese (3.8%), Korean (1.9%), Japanese (1.5%), Russian (1.0%), and Vietnamese (0.18%). Other languages such as Ukrainian (0.06%), Thai (0.02%), Persian (0.02%), Hebrew (0.01%), and Greek (0.01%) contributed minimally on average. Chinese consistently appears as the most dominant non-English language throughout

the decade, growing from 1.6% in January 2015 to 3.4% in May 2025. Its influence peaked in December 2020, when it spiked to 22.1% of all messages, likely contributing to the multilingual surge observed in §6.4. Korean shows the most significant growth, rising from 0.3% in early 2015 to 5.2% in May 2025, with a peak of 7.3% in November 2024. Japanese usage remains steady, fluctuating around 1.1% to 1.7%, while Russian gradually increases from 0.5% to 1.1%. Vietnamese has a low average but shows occasional spikes, reaching 2.2% in December 2024.

ObservationsThese results highlight three key observations. First, the top five non-English languages, Chinese, Korean, Japanese, Russian, and Vietnamese, account for the vast majority of multilingual content on GitHub. Their consistent presence over the 10-year period suggests that these languages correspond to stable and active developer communities, rather than sporadic or event-driven participation.

Second, the rapid growth of Korean messages, particularly after 2021, signals a rising open-source contributor base in South Korea. Korean-language content increased from 0.3% in 2015 to 5.2% in 2025, despite South Korea representing less than 1% of the global population. This contrast highlights an unusually high level of open-source engagement relative to country size. National initiatives have likely played a role: for example, the Korean government's *Open Source Software Invigoration Plan* (2014) and the 2020 amendment to the *Software Promotion Act* mandated that publicly funded software projects release their code under open-source licenses [258]. There are also active Korean open-source groups, such as the Linux Foundation's OpenChain Korea Work Group[3], and prominent projects like SeoulTech's `open-data-seoul`[4], which further illustrate a growing open-source ecosystem.

Third, although the average percentages for Thai, Vietnamese, and Persian remain below 0.2%, all three show clear upward trends. Vietnamese usage increases steadily, while Persian reaches 0.13% by early 2025. The rise in Persian-language content is especially notable given the context of international sanctions and access restrictions facing Iranian developers. These growing contributions suggest persistent effort from developers in regions often underrepresented in global software ecosystems.

6.6 RQ3: Is source code becoming more multilingual?

We now turn to the source code to examine whether developer-written elements, such as comments, literals, and identifiers, show increasing use of non-English text over time.

Temporal Sampling To capture temporal changes in the use of natural language within code, we sampled 385 patches per month from January 2015 to May 2025. This number was derived from an estimated 1.12 billion public GitHub contributions [458], providing a 95% confidence level with a 5% margin of error. We first filtered for patches that introduced at least one new file within each push event to ensure sufficient syntactic context, as new files typically contain complete function definitions, class declarations, comments, and string literals—the main elements where natural language appears. Parsing partial patches, in contrast, is difficult and often yields incomplete or

[3] `https://github.com/OpenChain-Project/OpenChain-KWG`
[4] `https://github.com/SeoulTech/open-data-seoul`

noisy syntax. To remove trivial or auto-generated changes, we applied a 500-character minimum threshold. Such filtering helps retain meaningful development activity and sufficient semantic context; prior mining studies likewise exclude very small commits or diffs for this reason [280, 245]. We then randomly selected 385 patches per month (one per repository), resulting in a total of 48,125 patches. We chose this approach because partial patches are hard to parse and rarely provide enough syntactic context. New files, in contrast, usually contain full function definitions, class declarations, comments, and string literals, the main places where natural language appears. The same strategy was applied to documentation files to analyze multilingual README content.

Code Parsing We selected the top five programming languages on GitHub [189]: JavaScript, Python, Java, TypeScript, and C#, to ensure broad coverage across dominant ecosystems. Each source file was parsed using `tree-sitter` [489] to extract and tokenize the following elements based on each language's grammar: The parser extracted and tokenized the following language elements based on each language's grammar:

- **Identifiers:** user-defined variable and symbol names, excluding function and class declarations.
- **Function and Class Names:** identifiers used to declare functions, methods, and classes.
- **Literals:** string literals excluding minified or template-embedded content.
- **Comments:** inline, block, and documentation comments (e.g., Python docstrings [400] or JavaDoc [368]).

Language Detection For code-level elements, we used the Google Cloud Translation API[5] instead of `Lingua`. As noted in Lingua's documentation, the model struggles with short or mixed-language inputs, a common characteristic of code comments and identifiers. We empirically confirmed this limitation using a manually annotated sample of 1,000 short texts extracted from code. As shown in Figure 6.2b, Google Translator achieved higher precision and recall for detecting non-English text under the same 0.9 confidence threshold. We therefore adopted the Google API for code-related content, using 0.9 as the cutoff to remain consistent with our earlier setup. All predictions below this threshold were discarded. To focus on human-written content, we also removed language-specific keywords such as `int`, `float`, and `bool`, which are always English and not relevant to multilingual analysis.

Results Our findings show a steady rise in multilingual usage across comments and string literals, as illustrated in Figure 6.5. These parts of the code are used for documentation or user-facing text and give developers more freedom to use their native language. In contrast, structural elements such as identifiers, class names, and function names remain overwhelmingly English. This is likely due to two main technical constraints. First, many programming languages have restrictions on how identifiers can be named. For example, JavaScript only added support for Unicode 5.1 identifiers in ES6, which was released in 2015 [409]. Unicode 5.1 itself was released in early 2008 [494]. Older versions of C++ also lacked full Unicode support for identifiers [531]. Some compilers and linters still warn against using non-ASCII characters. Second, typing in

[5] `https://cloud.google.com/translate/docs/reference/rest/v3/projects/detectLanguage`

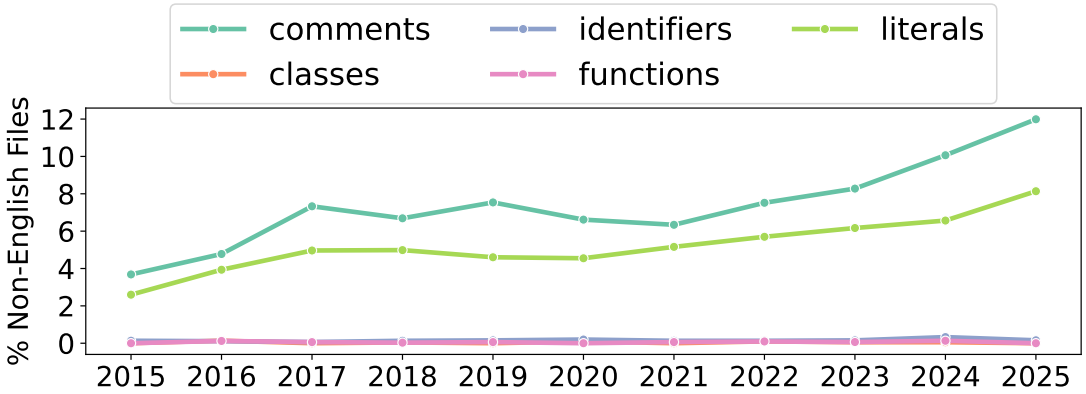


Figure 6.5: Share of non-English content in code elements from 2015–2025, with comments and string literals showing the most growth.

non-Latin scripts like Arabic or Chinese often requires switching keyboard layouts or input methods. This interrupts coding and slows down development. Some tools also do not fully support multilingual code. For example, older versions of Eclipse [367, 150] and IntelliJ IDEA [262] may show broken characters when displaying mojibake text in variable names or logs, especially if encoding is not set correctly.

These limits make it harder for developers to use non-English text in core parts of code, even if they use their native language in comments or documentation. The results reflects this pattern clearly. Comments exhibit the strongest trend, with non-English usage rising from 3.6% of files in 2015 to 11.9% in 2025, a 225% relative increase. Korean comments, for instance, were present in only 0.27% of files in 2015 but rose to 1.7% by 2025, reflecting a steady increase in participation from Korean-speaking developers. String literals also show a significant change, increasing from 3.2% to 9.6%. These elements are commonly used for documentation or user-facing text, making them more susceptible to natural language variation. One likely factor driving this recent rise is the growing use of large language models (LLMs) such as GitHub Copilot and ChatGPT. These tools often produce code with more inline comments than human-written code [375] and support multilingual input, enabling developers to write comments in their native language.

Identifiers, variables, and functions show smaller but consistent increases. The share of files with non-English identifiers grew from 0.14% to 0.28%, and for variables, from 0.09% to 0.28%. Though still marginal in absolute terms, these increases reflect a gradual broadening in naming practices. Function names remained almost entirely in English, with only negligible changes observed. Class names showed no measurable use of non-English terms throughout the period. For example, in a Japanese TypeScript project, developers used Japanese characters for both a variable and a class name in a VS Code extension plugin[6]. In a Java testing project, one function is defined using Korean characters[7]. Another Python script includes a Chinese function name for a

[6]https://github.com/sazae657/VS-Shimonizer
[7]https://github.com/arahansa/CodeCoast30Min

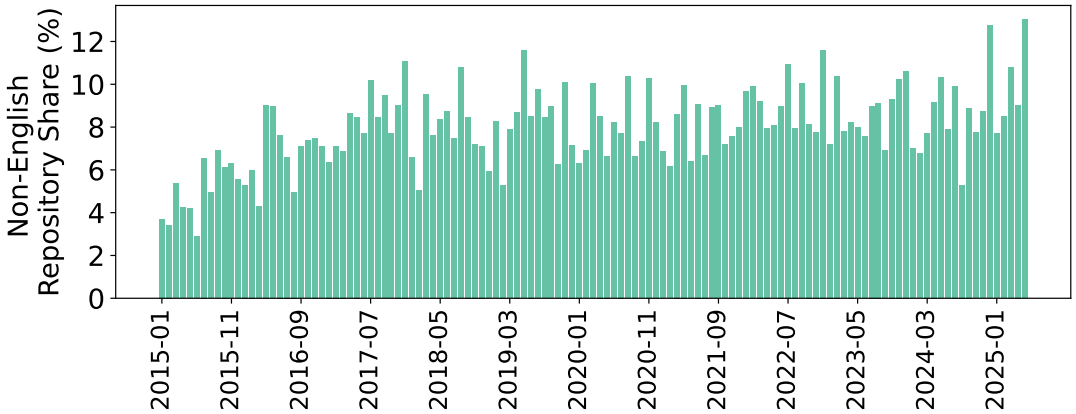


Figure 6.6: Percentage of repositories containing non-English documentation from 2015 to 2025.

math operation[8]. These examples are rare and do not yet reflect a large-scale shift, but they illustrate that non-English naming is both technically feasible and sometimes adopted.

In addition to code-level elements, we analyzed documentation files, i.e., READMEs, to assess multilingual trends in project-level communication. Based on monthly samples from 2015 to 2025, the proportion of repositories with non-English documentation increased from 3.7% in January 2015 to 13.0% in May 2025, as shown in Figure 6.6. The overall average across the dataset is 8.0%. Among non-English documentation, Chinese is the most prevalent, accounting for 3.3% of all repositories. Vietnamese documentation shows one of the most consistent upward trends, rising from 1.06% in 2015 to 3.20% in 2025. Russian (1.3%), Korean (0.7%), and Japanese (0.5%) are also commonly observed, while Ukrainian, Thai, Greek, and Arabic collectively account for less than 0.2%. These findings reflect a substantial shift toward multilingual documentation, reinforcing the broader trend of increasing language diversity in open-source projects. This shift can improve inclusivity by making projects more accessible to local contributors. For instance, Vietnamese README files help first-time contributors from Vietnam engage without needing English fluency.

These results suggest that the increase in multilingualism is concentrated in human-facing components of code, particularly comments and literals. In contrast, structural identifiers such as class and function names remain almost exclusively English, indicating a persistent linguistic norm in core elements of program logic and syntax.

[8]https://github.com/DavidLXu/MathY

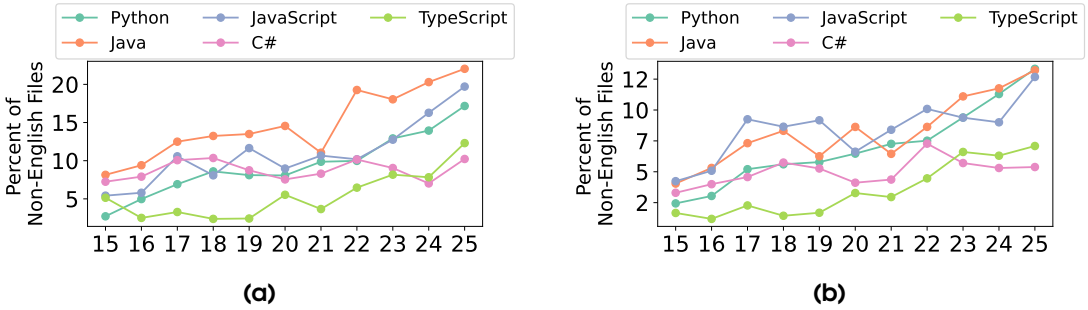


Figure 6.7: Non-English content in source code by programming language from 2015 to 2025. (a) Percentage of files with non-English comments. (b) Percentage of files with non-English string literals. Java, JavaScript, and Python exhibit the most significant multilingual growth over time.

6.7 RQ4: Does multilingualism vary across programming languages?

This research question investigates whether multilingual practices differ across programming languages and which ecosystems exhibit the most non-English content in source code and documentation.

Data Sampling For this analysis, we examined the inclusion of natural language across programming languages using the same code parsing and language detection approach described in §6.6. To control for language-specific effects, we analyzed the same five major programming languages introduced earlier: JavaScript, Python, Java, TypeScript, and C#. Because random sampling in GitHub data tends to overrepresent JavaScript and Python, we applied stratified sampling to maintain balanced representation. Specifically, for each month from January 2015 to May 2025, we randomly selected 100 repositories per programming language, resulting in 500 repositories per month and a total of 62,500 repositories over the 125-month period, applying the same exclusion criteria used in **RQ3**.

Results Figure Figure 6.7a and Figure 6.7b show how non-English text in string literals and comments changed from 2015 to 2025. The trends differ across languages. Java has the most steady rise in multilingual content in all parts of the code that hold natural language. In 2025, 22.03% of Java files have non-English comments and 13.2% have non-English literals. In 2015, these numbers were 8.1% and 4.04%. Java's growth likely comes from its long role in global education and enterprise work. It is still used worldwide, including in many regions where English is not the main language [346, 263]. Java also stays common in mobile and GUI software, such as Android apps, where developers often put localized strings directly in code [23]. These use cases and the strong developer communities outside English-speaking countries help explain Java's high multilingual share.

Python also shows steady growth in multilingual content across most elements. In 2025, 17.1% of Python files contain non-English comments, and 13.3% include non-English string literals. This rise matches Python's wide global adoption, especially

in education and data science [221, 18, 345]. Python is now the most used language worldwide, including in places where English is not the first language [189]. Many schools teach Python as the first programming language, so students often write comments or literals in their own language. This learning environment, along with Python's simplicity, likely increases its share of non-English text.

JavaScript shows a pattern similar to Java. Non-English string literals rise from 4.2% in 2015 to 12.7% in 2025. Comments grow from 5.4% to 19.7%. C# and TypeScript show the lowest multilingual levels. In 2025, only 10.2% of C# files have non-English comments. TypeScript shows 12.3% for comments, even though its non-English literals grow from 1.6% to 7.08%. These languages often have more stable or corporate-focused communities and slower adoption. C# use remains tied to Microsoft and enterprise settings. For example, the official C# Dev Kit for Visual Studio Code requires a paid license for teams with more than five developers [325], which may limit use among small groups. TypeScript is still mostly developed and maintained by Microsoft [532], which may slow broader adoption.

Multilingual use in identifiers, class names, and function names stays rare in all languages. These elements almost never exceed 0.3%, which aligns with common naming practices and tools that default to English [82]. We also see rising multilingual use in Python, JavaScript, and Java after 2022, which matches the spread of large language models like GitHub Copilot [186] and ChatGPT [366]. Around the same time, GitHub noted that most new contributors are from non-English-speaking regions [189]. LLMs can understand and produce many languages [269], so they make it easier for developers to work in their native language. This likely supports the rise in multilingual content. Recent studies also show that LLMs tend to prefer Python, choosing it in 90–97% of benchmark tasks and 58% of project setups [493]. This preference likely strengthens Python's multilingual growth after 2022.

6.8 RQ5: Do multilingual repositories show signs of language friction or coordination issues?

While RQ1 examined broad trends in the growth of non-English content across GitHub discussions, this question shifts the focus to the repository level. By aggregating language usage by repository, we examine whether projects operate primarily in a single dominant language or support communication in multiple languages. This allows us to assess whether open-source development is becoming more linguistically inclusive at the project level and whether multilingual collaboration introduces coordination challenges or fosters inclusive behavior.

Repository Classification We use the same dataset as in §6.4, grouping messages by repository ID and aggregating the language of comments in issues, pull requests, reviews, and discussions. Language detection is performed using `Lingua`, as in RQ1. Each repository is then classified based on the proportion of messages in each language: **English-dominant** if at least 90% of all comments and issues are in English, **non-English-dominant** if at least 90% are in a single non-English language, and **mixed-language** otherwise. Setting a higher threshold can potentially inflate the number of

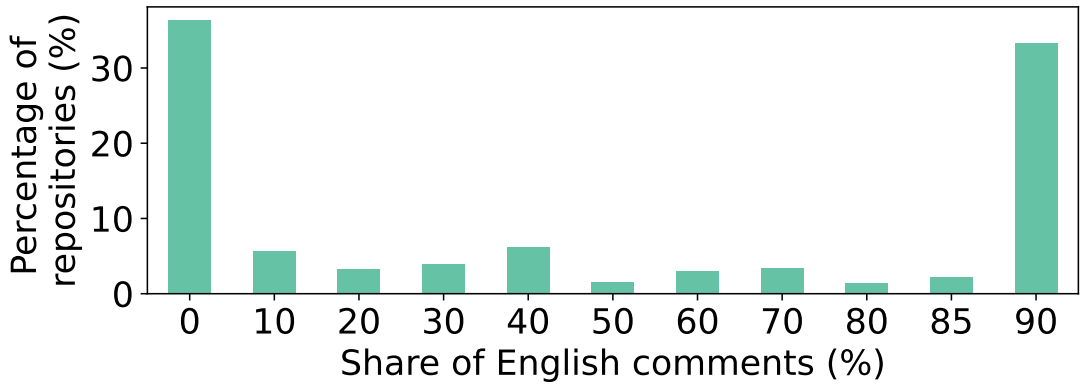


Figure 6.8: Distribution of English content across GitHub repositories in our dataset.

Figure 6.9: Number of repositories by dominant discussion language. Most multilingual activity is concentrated in English-dominant and mixed-language projects, with significant representation from Chinese, Japanese, and Korean repositories.

repositories labeled as multilingual; however, in our data, the distribution of English content is skewed toward the two extremes, as shown in Figure 6.8. As a result, choosing a lower threshold such as 80% or 70% would produce nearly similar groupings, as the difference is negligible.

Findings As shown in Figure 6.9, the majority of repositories fall into the English-dominant category, reflecting the continued central role that English plays in global open-source communication. However, we also observe a substantial number of non-English-dominant repositories, particularly in Chinese, Japanese, and Korean. These projects suggest the presence of regionally focused development communities where contributors primarily interact in their native languages.

To understand coding practices in non-English repositories, we examined 30 randomly selected Chinese and Russian projects. In the case of Chinese repositories, we found that 70% included documentation written in Chinese. The use of Chinese within source code was somewhat lower: 40% of the projects contained Chinese comments, and 60%

used Chinese in code elements, particularly in string literals. Similarly, among Russian repositories, 68% had Russian documentation. Russian comments showed up in 40%, and 28% used Russian in code elements, especially in literals. These observations suggest that even in non-English-dominant projects, developers often retain English conventions in code while localizing documentation, comments, and interface-level strings. This finding aligns with our earlier results in §6.6, where comments and literals showed the highest multilingual growth among code elements.

While these projects demonstrate a degree of linguistic flexibility, such diversity can also introduce both opportunities and frictions. In repositories where a single language dominates, whether English or a non-English language, that language can shape participation dynamics. It sets expectations for communication, influences who feels comfortable contributing, and can either include or exclude developers based on their language proficiency. To understand how language is negotiated in practice, we searched for signs of language enforcement in our dataset by identifying phrases such as "write in English," "напишите на русском" (write in Russian), "请写中文" (write in Chinese), "日本語で書いてください" (write in Japanese), and "한국어로 써 주세요" (write in Korean). We found 3801, 182, 1334, 717, and 149 occurrences of these phrases, respectively. To understand the context behind such interactions, we conducted a manual inspection of a sample of 500 issues containing these phrases. Our analysis found that language-related comments appeared in both English-dominant and non-English-dominant repositories, though the tone and intent varied. In English-dominant repositories, such remarks were often framed as reminders or expectations to communicate in English, especially in response to issues written in other languages. In non-English-dominant repositories, similar phrases appeared more sporadically and were often expressed as suggestions aimed at improving clarity or maintaining consistency. We also observed several cases where contributors in non-English-dominant projects explicitly enforce the use of their native language, such as Russian or Chinese. These findings suggest that language enforcement is not exclusive to English speakers and can occur in any community where shared linguistic norms guide collaboration. Table 6.1 lists ten representative examples of such linguistic tension, along with their corresponding GitHub issue URLs.

We also observed several cases where contributors and maintainers navigated multilingual discussions fluidly, often switching languages mid-thread to accommodate each other's preferences or limitations. For instance, in the `Buildroot` project,[9] a contributor used Russian, and the maintainer explicitly welcomed it, stating they would rely on translation tools. The conversation continued in Russian, prioritizing clarity over formality. In another case, `mod_execdir`,[10] the issue began in English, but the maintainer noted they were more comfortable in Korean and encouraged the contributor to switch, which they did. Similarly, in `Far-NetBox`,[11] a thread that started in English naturally transitioned into Russian, likely because both parties shared the language.

[9] https://github.com/furkantokac/buildroot/issues/10#issuecomment-920325626

[10] https://github.com/OOPS-ORG-PHP/mod_execdir/issues/18#issuecomment-333383366

[11] https://github.com/michaellukashov/Far-NetBox/issues/197

Table 6.1: Examples of language-related comments and enforcement in GitHub issues.

Comment (Paraphrased)	GitHub URL
用中文是 Xray 社区的传统，以区分于其它喜欢用英文的项目，有些还用出优越感了	https://github.com/XTLS/Xray-core/issues/4348#issuecomment-2634739760
@BekzodUzb All good, but we don't understand Turkish and even machine translate doesn't help. Write in English or Russian please.	https://github.com/FWGS/xash3d-fwgs/issues/1944#issuecomment-2569944991
Please write in English on our GitHub. Otherwise nobody really can read and understand what you are saying.	https://github.com/flybywiresim/simbridge/issues/137#issuecomment-2578127993
Folks, writing in French here is not polite. We communicate in English.	https://github.com/DurgNomis-drol/ha_toyota/issues/319#issuecomment-2578233882
这里再用中文写一遍（可能需要先生你去翻译一下）	https://github.com/ozntel/file-tree-alternative/issues/250#issuecomment-2599870921
а еще в этом трекере можно писать по-русски)	https://github.com/Sectoid/nemerle/issues/1#issuecomment-4574175
... 习惯先写中文了，毕竟我是人	https://github.com/X1a0He/Adobe-Downloader/issues/89#issuecomment-2782004919
建议如果写英文可以先用中文描述好后机翻一下。或者还是在 github 里直接写中文吧 。	https://github.com/RT-Thread/rt-thread/pull/10166#issuecomment-2785061178
PR 제목을 한국어 사용자가 읽기 편하도록 한국어로 작성하시면 어떨까요?	https://github.com/DaleStudy/daleui/pull/170#issuecomment-2815349228
이 PR은 한국어로 작성되었으니 리뷰도 한국어로 부탁드립니다.	https://github.com/sparta-java-team-22/tweaty/pull/49#issuecomment-2831811605

Finally, in MPLUS_FONTS,[12] the maintainer encouraged the contributor to write in Japanese due to language difficulty. This led to a smooth continuation of the discussion in a bilingual Japanese–English setting.

6.9 RQ6: Is Multilinguality Detrimental to Repository Engagement?

While multilingualism can support inclusivity, it may also negatively influence cooperation, i.e., how repositories are discovered, contributed to, and maintained. To investigate this, we analyzed four key engagement metrics across repositories categorized as English-dominant, non-English-dominant, and mixed-language: total number of comments (as a proxy for participation), number of contributors (as a measure of collaboration), number of stars (as a proxy for popularity and visibility), and issue resolution time (as a proxy for responsiveness).

[12] https://github.com/coz-m/MPLUS_FONTS/issues/10#issuecomment-840220932

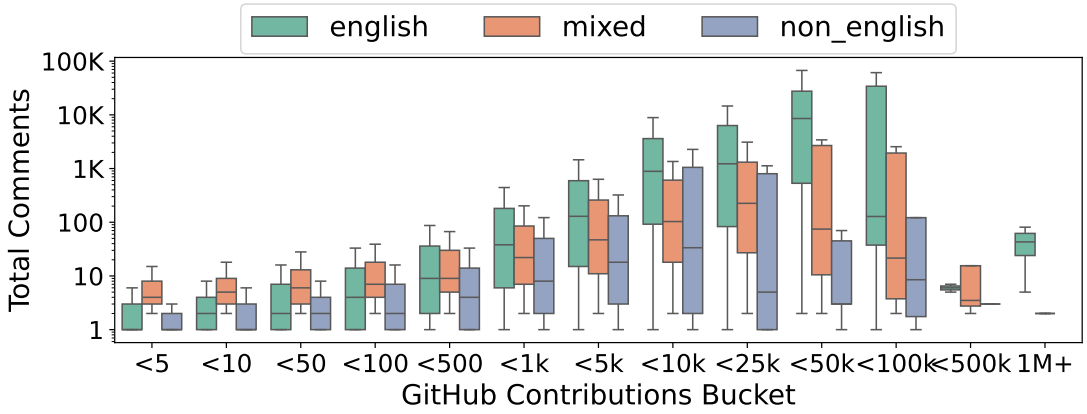


Figure 6.10: Total comments per repository by language category across contribution buckets (repository groups based on total commits). English-dominant projects lead in large buckets, while mixed-language projects show higher participation in smaller ones.

Methodology We use the same repository classification as in RQ5, grouping all issues, pull requests, reviews, and discussion comments by repository, and labeling each as English-dominant, non-English-dominant, or mixed-language.

We define a *contributor* as any non-bot GitHub user who has actively participated in a project through actions such as pushing code, opening pull requests, commenting on issues or code, creating new files, or publishing a release. This broad definition includes both code contributors and those involved in discussions, reviews, or project maintenance. Bot accounts were filtered based on known bot patterns in usernames and activity metadata. We restrict our analysis to activity occurring between January 2015 and May 2025, aligning with the overall timeline of our dataset.

Each metric is visualized on a logarithmic scale and across buckets of GitHub contribution activity, comparing distributions across language categories. We use this bucketed approach to control for repository size, since aggregating metrics across all projects regardless of scale could mask important differences. Larger repositories naturally attract more comments, stars, and contributors, so comparing projects within similar activity levels provides a clearer picture of how language influences engagement.

Statistical Testing To ensure that the observed engagement differences are not due to random variation, we conducted statistical tests across all three engagement metrics: number of comments, number of contributors, and number of stars. Because these metrics are discrete and heavily right-skewed, we used non-parametric tests that do not assume normality. Specifically, we applied the Kruskal–Wallis H-test to compare distributions across the three language categories (English-dominant, mixed-language, and non-English-dominant). When the Kruskal–Wallis test showed significant differences, we performed pairwise Mann–Whitney U tests with Bonferroni correction to identify which groups differed. We report the corresponding H statistic, adjusted p-values, and effect sizes (η^2) for each metric.

Participation: Number of Comments Figure 6.10 shows the distribution of total

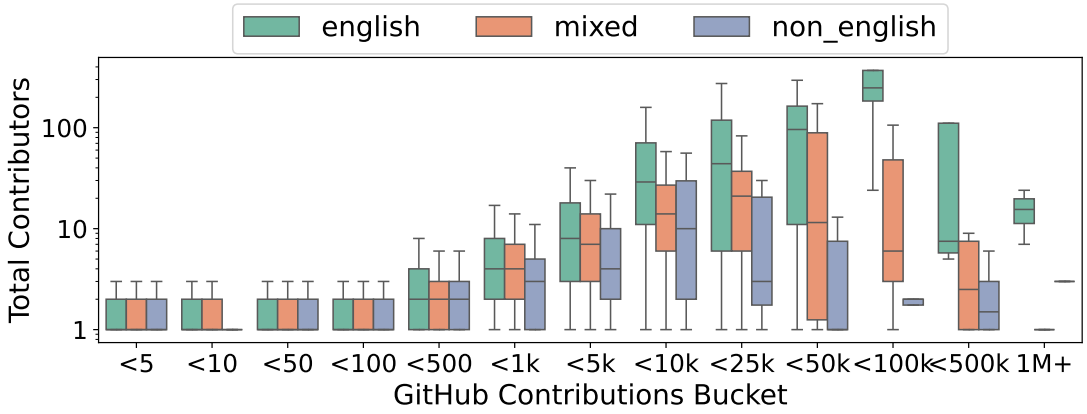


Figure 6.11: Distribution of unique contributors per repository across language categories. English repositories have more contributor growth as project size increases, while non-English repositories consistently have fewer contributors.

comments per repository across contribution buckets. In smaller repositories, mixed-language projects have a slightly higher median number of comments than English-dominant ones, suggesting more active localized discussion. However, as repository size increases, English-dominant projects surpass mixed-language repositories, both in comment volume and growth rate. The gap between English and mixed repositories widens steadily in larger buckets, with English-dominant repositories showing the highest levels of discussion beyond 1k contributions. Non-English-dominant repositories consistently show the lowest median comment counts across all size categories. This pattern suggests that non-English repositories may experience reduced participation in collaborative discussions. Since GitHub comments are central to issue triage, bug reporting, feature discussion, and community feedback, lower comment volume may hinder transparent collaboration or discourage new contributors from engaging.

The Kruskal–Wallis test confirmed that the observed differences are statistically significant ($H = 101{,}897.31$, $p < 0.001$, $\eta^2 = 0.1859$). Pairwise Mann–Whitney U tests with Bonferroni correction further showed that English-dominant repositories receive significantly more comments than both mixed and non-English repositories ($p < 0.001$). These results indicate a large practical effect, with language category explaining a substantial share of the variation in participation.

Collaboration: Number of Contributors To assess collaboration, we count the number of unique contributors per repository. Figure 6.11 shows the contributor distribution across language categories. In smaller repositories, contributor counts are generally low and similar across all groups. This is expected, as many small projects are maintained by a single developer or a very small team [53, 194]. Contributions often come from a few users who report bugs, request features, or submit patches. As a result, the median contributor count stays low regardless of the language used. As repositories grow, differences between language groups become clearer. English-dominant projects show steady contributor growth, likely due to greater visibility and broader accessibility.

They may attract a more geographically diverse contributor base. Mixed-language repositories show slower growth, even with multilingual participation. This may result from communication challenges (§6.8), inconsistent norms, or unclear language policies that discourage contributors. While causality is hard to confirm, factors like popularity, responsiveness, repository age, and community size may influence these patterns. Further study is needed to examine these. Non-English-dominant repositories consistently show the fewest contributors across all sizes. These projects often serve regional or linguistic communities and may be less visible on GitHub. While this focused scope doesn't imply lower quality, it can limit collaborative growth. For example, the French project `ratpstatus`[13] tracks transport disruptions in Île-de-France and is active, yet has only two contributors. The Chinese `API` project[14] offers reverse data interfaces with frequent commits, also by just two contributors. These examples show how language barriers and low discoverability can discourage external contributors. Using a local language can help serve a specific community, but comes with the cost of reduced visibility. Developers should consider this tradeoff. Language choice affects who participates. Prior work shows linguistic divides hinder global collaboration [522], while inclusive communication helps attract and retain diverse contributors [250].

The Kruskal–Wallis test confirmed a statistically significant difference in contributor counts across language categories ($H = 6{,}509.79$, $p < 0.001$, $\eta^2 = 0.0194$). Pairwise Mann–Whitney U tests with Bonferroni correction showed that English-dominant repositories have significantly more contributors than both mixed and non-English repositories ($p < 0.001$). Although the effect size is smaller than for participation, the difference remains systematic, indicating that language orientation influences collaboration at scale.

Visibility: Number of Stars GitHub stars are often used as a proxy for a project's popularity and reach. Prior work shows that stars signal user interest and are widely used in empirical studies to rank and select repositories [66, 411, 67, 277, 65]. Stars also have practical value: users bookmark projects they find useful [553], and maintainers highlight star counts to attract contributors or funders [116]. A survey by Borges et al. [67] reports that most developers look at stars before using or contributing to a project. Figure 6.12 shows the distribution of GitHub stars across repositories grouped by language category and contribution size. In small repositories (fewer than 100 contributions), differences in star counts are modest. But as activity increases, English-dominant repositories attract substantially more stars, dominating the upper visibility tiers. Mixed-language repositories follow, showing moderate growth, while non-English-dominant repositories consistently receive the fewest stars across all sizes. This highlights a clear visibility gap tied to language, with English projects more likely to gain recognition on the platform.

Unlike contributor count, which reflects actual participation, stars reflect perceived value or utility and are shaped by visibility. Mixed-language repositories gain more traction than non-English ones but still fall behind English projects. Non-English repositories remain concentrated in lower star ranges. Projects that do not use English may struggle to attract attention even when active and well-maintained. Because highly

[13] `https://github.com/wincelau/ratpstatus`
[14] `https://github.com/yuncaiji/API`

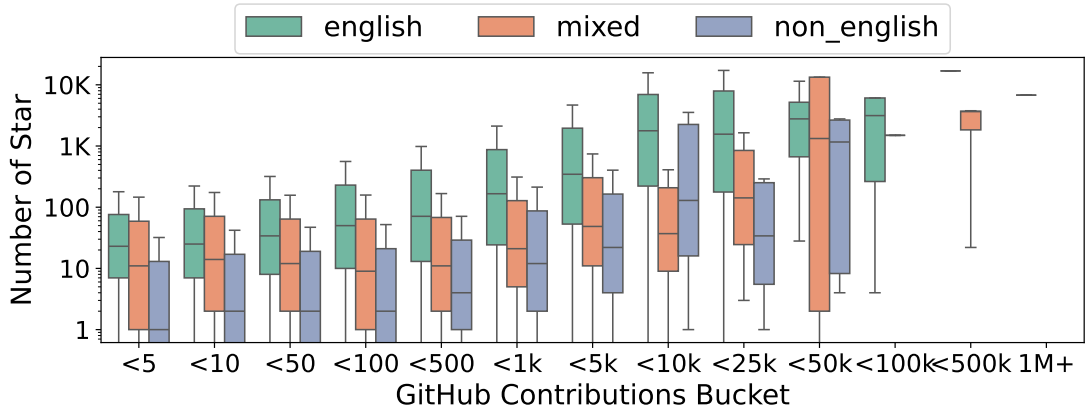


Figure 6.12: Distribution of GitHub stars per repository across language categories and contribution levels. English-dominant repositories show higher visibility, especially as project size increases.

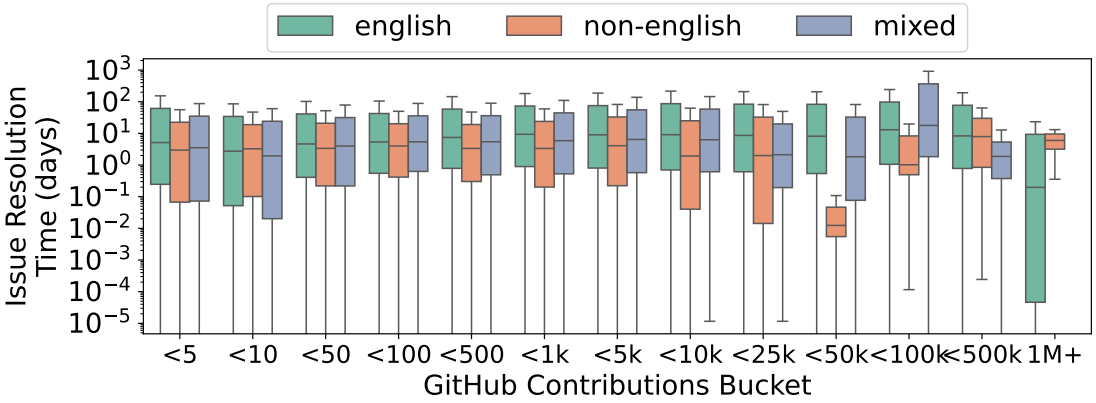


Figure 6.13: Distribution of issue resolution time across language categories.

starred repositories are more likely to be recommended or referenced, this gap reinforces platform inequality.

The Kruskal–Wallis test confirmed a significant difference in star counts across language categories ($H = 9{,}279.26$, $p < 0.001$, $\eta^2 = 0.1441$). Pairwise Mann–Whitney U tests with Bonferroni correction showed that English-dominant repositories have significantly more stars than mixed and non-English repositories, and mixed repositories more than non-English ones ($p < 0.001$; medians: 36, 12, and 2, respectively). The large effect size indicates that language category strongly influences repository visibility and popularity.

Responsiveness: Issue Resolution Time Issue resolution time is a key indicator of project responsiveness and maintenance quality. It reflects how quickly developers react to user feedback, bug reports, or feature requests [550, 207]. Previous studies show that long delays can slow progress and reduce overall project performance [229]. Figure 6.13 shows the distribution of issue resolution times across language categories.

English-dominant repositories take the longest to resolve issues, followed by mixed-language, and then non-English repositories. A Kruskal–Wallis test confirmed that these differences are statistically significant ($H = 417{,}873.16$, $p < 0.001$, $\eta^2 = 0.0065$). Pairwise Mann–Whitney U tests show significant differences across all pairs ($p < 0.001$), with median times of 7.17 days for English projects, 4.70 days for mixed projects, and 3.39 days for non-English projects.

While the effect size is small, the trend is systematic and somewhat counterintuitive: one might expect English-dominant projects, with larger contributor pools, to resolve issues faster. Instead, they often receive more issues and face coordination overhead due to scale and global participation [508, 514]. Non-English projects are often smaller and community-focused, which can lead to quicker responses within narrower scopes. Another possible explanation is that some countries, such as Korea, actively support or fund OSS in local languages, which could influence responsiveness. This remains a hypothesis, and future work should examine it further. These findings suggest that linguistic orientation not only shapes collaboration but also affects how efficiently projects address reported problems.

6.10 Threats to Validity

Language Detection Accuracy: Detecting the natural language of short and informal GitHub text is challenging, especially for code-mixed messages or comments with few tokens [52, 41, 283]. Models may misclassify languages with similar scripts or struggle when inputs are short or noisy. To reduce these errors, we use Lingua and Google Translate API in strict confidence mode and accept predictions only when confidence exceeds 0.9; otherwise, the message is excluded. We restrict analysis to non-Latin languages because Latin-based languages such as English, German, Spanish, and French share many common words, which often confuses detectors on very short text like identifiers. Our estimates are therefore conservative and likely represent a lower bound.

Filtering Criteria: For code-level analysis, we included only patches that introduced at least one new file and contained more than 500 characters. As our goal is to study longitudinal change, separating small edits or mixed-language content within the same file is difficult for reliable code parsing, which further motivates focusing on newly added files. Although this filter may exclude some meaningful edits, such as documentation updates or refactorings, applying it uniformly across all time periods and languages preserves comparability.

Partial Contribution History: All participation metrics such as the number of comments, stars, or contributors, are calculated based on activity observed between January 2015 and May 2025. Any comments, stars, or contributions made before 2015 or after May 2025 are not considered. This temporal constraint may undercount activity, particularly for older repositories or projects with long histories. However, we apply the same time window across all language groups, so we believe that comparisons remain valid within the scope of the dataset. Additionally, we believe this dataset remains representative due to the statistically significant sample size of contributions collected

over the decade-long period, which captures a substantial portion of activity across diverse repositories.

Repository Sampling and Classification: For code-level multilingual analysis (RQ3, RQ4, RQ5), we sample repositories and code patches from monthly GitHub activity. This ensures temporal and language balance but may miss less active or less visible repositories. Additionally, we classify repositories as English-dominant, non-English-dominant, or mixed based on issue and comment data, using a 90% threshold for language dominance. In practice, this threshold has minimal impact on group composition because the distribution of English content is strongly skewed toward the extremes, meaning lower thresholds (e.g., 80% or 70%) would produce similar groupings.

Generalizability: Our findings are based on GitHub, which has its own cultural and technical norms. Results may not generalize to other platforms such as GitLab or Bitbucket, or to open-source projects hosted independently. In addition, GitHub may not be equally popular or accessible in all regions. It may be less used in some countries or even subject to access restrictions or bans.[15] For example, GitHub has been temporarily blocked in countries like China, India, and Russia due to government censorship. GitHub has also restricted access to developers in certain regions to comply with U.S. sanctions, including countries such as Iran and Syria, as well as the Crimea region.[16] These limitations may affect the visibility and participation of developers from those regions. Moreover, our analysis focuses on public repositories; multilingual behavior in private or enterprise repositories may differ.

6.11 Discussion

The rise of multilinguality in open-source repositories marks a shift in global software development. While it increases accessibility for non-English-speaking developers, it also introduces challenges for maintainers, researchers, and developers who must adapt to support broader participation.

Beyond English in Developer Research: Much of the current research on developer communication, whether focused on toxicity [327, 439, 440], sexism [471], politeness [138], or emotional tone [344], has been conducted exclusively on English-language data. This creates a blind spot in the community's understanding of global developer behavior. As our results show, a significant portion of open source interaction now happens in other languages. Tools and models developed on English-only corpora risk missing or misclassifying key signals in multilingual settings. Future work should explore language-specific forms of toxicity, inclusion, and engagement. This may require building labeled datasets in non-English languages, adapting sentiment or toxicity classifiers for regional communication norms, and evaluating how moderation practices vary across linguistic communities.

Multilingual Tooling and Infrastructure: Automation is central to modern workflows, from issue triage to documentation bots [527, 436, 197, 196, 195]. Many tools assume English input, limiting their usefulness in multilingual repositories. New

[15] https://en.wikipedia.org/wiki/Censorship_of_GitHub
[16] https://techcrunch.com/2019/07/29/github-ban-sanctioned-countries/

tooling should support translation-aware linting, multilingual bots, localized search, and documentation generation. IDEs could offer localized interfaces and clearer error messages. Compilers and programming languages could provide better diagnostic support for multilingual users.

Community Tensions Around Language Use: As multilingual participation grows, contributors negotiate which language to use and when. Our RQ5 findings show that language enforcement is a recurring issue in GitHub repositories. Phrases like "write in English" or "请写中文" reveal ongoing frictions around communication norms. This enforcement is not limited to English-dominant projects; contributors in non-English repositories also assert language preferences. Multilingualism does not remove gatekeeping but shifts its focus, as language choice can include or exclude depending on how norms are set. Whether multilingual collaboration becomes more inclusive or more fragmented depends on project culture and governance. Some maintainers use translation tools or switch languages to accommodate contributors, while others reject messages or ask users to follow a preferred language. Future work could explore strategies such as language labels, integrated translation support, or multilingual contribution guidelines to manage these tensions more effectively.

The Visibility and Contribution Tradeoff: Despite the advantages of multilingualism, our findings show a drawback: repositories using non-English languages tend to receive fewer stars, attract fewer contributors, and generate fewer comments. This pattern is strongest in larger, active projects. English remains the lingua franca of open source, and moving away from it can reduce a project's discoverability. These metrics matter because they signal popularity and activity to funders, contributors, and users [67, 116, 237]. Projects outside the English-dominated space may struggle to gain traction even when serving important regional needs. Developers managing multilingual projects should be aware of this visibility gap. When possible, English summaries, dual-language READMEs, or tagging issues for translation can help bridge linguistic divides. There is no single solution: some projects may prefer regional accessibility over global reach. What matters is making intentional language choices rather than defaulting to one approach.

Demographics and Linguistic Dynamics in OSS: Prior work on inclusion in OSS often focuses on demographics such as ethnicity, gender, or geography [69, 437]. These factors matter, but language and economics also significantly shape participation, and they are largely overlooked in the literature. Demographic trends do not directly correspond to linguistic behavior. Countries like India and Pakistan have multiple official languages, yet English is widely used as the working lingua franca. As a result, demographic counts alone cannot portray actual inclusion patterns. These linguistic choices reflect cultural, educational, and economic forces that demographic indicators cannot capture. Future research on inclusivity should consider these nuances rather than relying solely on demographic measures.

AI and the Future of Multilingual Onboarding: Language barriers are known to hinder onboarding in open-source communities. Prior work shows that newcomers struggle to participate when project communication is English-centric [466], and mentoring relationships often break down when contributors lack shared language

proficiency [35]. As multilingual participation grows, these challenges may intensify. AI tools offer a possible mitigation path: automatic translation, summarization, and code-aware rewriting could help newcomers read and write project artifacts in their preferred language, lowering the entry barrier. At the same time, AI may also amplify linguistic tensions. Developers using AI assistants may generate more non-English issues, comments, and documentation, increasing the burden on maintainers and reinforcing the language negotiations we observe in mixed-language repositories. How communities integrate AI-mediated translation will shape whether multilingual collaboration becomes smoother or more fragmented.

6.12 Data Availability

The dataset for this study comes from public GitHub activity between January 2015 and May 2025, collected via GHArchive and the GitHub REST API. Due to the data volume, we do not redistribute raw messages. We release aggregated statistics, language annotations, and repository-level classifications at:

`https://github.com/kal-purush/bhasha-nirikhhon`

The repository includes preprocessing scripts, language detection outputs, and selected metadata to support replication.

6.13 Summary

This chapter presents the first large-scale study of multilingualism in open source software development across both communication and code. By analyzing 9.14 Billion GitHub discussions and over 62,500 code patches from 2015 to May 2025, we show that multilingual participation is steadily increasing, especially in Korean, Chinese, and Russian. This shift appears not only in issues and comments but also in code elements such as literals, comments, and documentation. Our results suggest that open source is gradually becoming more inclusive of non-English speakers. At the same time, the trend introduces new challenges. We observe language-based tensions around communication norms, and multilingual or non-English repositories tend to receive fewer contributions, comments, and stars. We also find a counterintuitive pattern in issue resolution: English-dominant projects resolve issues the slowest, while non-English projects are the fastest. Taken together, these findings highlight a tradeoff between linguistic expressivity and community inclusivity on one side, and visibility, collaboration, and coordination on the other. They point to the need for more language-aware tools, analyses, and future research on how multilingualism shapes developer experience, project success, and community dynamics.

Part III

Advancing Program Analysis for Modern Vulnerabilities and Complex Systems

7

Systematizing ReDoS: Understanding Concepts, Risks, Defenses, and Research Gaps

In the first two parts of this dissertation, we examined the socio-economic dimensions of modern software engineering, showing how economic conditions, geography, and language shape the challenges faced by contemporary software systems. These factors influence who can participate, who is excluded, and how software is designed and used in practice. In this third part, we shift our lens from *the user* to *the system*, focusing on redefining vulnerability analysis for the compositional age.

This chapter addresses the challenge of evolving threat models and systemic risks (challenge C_3; see Introduction). As discussed in the introduction, software engineering has moved away from monolithic designs toward compositional architectures built from large ecosystems of third-party components and algorithmic abstractions. In this compositional paradigm, severe vulnerabilities often emerge not from isolated implementation errors, but from the unanticipated interactions between components and the worst-case behaviors hidden behind trusted abstractions.

Regular Expression Denial of Service (ReDoS) serves as a paradigmatic example of this shift. Regular expressions are ubiquitous tools for input validation and routing, yet they are often used by developers without awareness of their worst-case performance risks [127, 459]. As a result, significant algorithmic complexity remains hidden behind a trusted abstraction. When expressive patterns interact with adversarial inputs, the underlying regex engine can enter catastrophic backtracking loops, causing resource exhaustion that threatens the availability of the entire service. Thus, ReDoS is not merely a bug; it is a manifestation of the tension between expressive power and predictable performance in modern systems.

At the same time, the growing prevalence of ReDoS highlights a core gap identified in C_3: the lack of consistent threat models and evaluation methodologies for emerging algorithmic vulnerability classes. In this chapter, we address this gap by presenting a Systematization of Knowledge (SoK). We systematically review existing research, analyze how modern regex engines handle ReDoS in practice, and discuss how these engineering realities must shape future detection and mitigation strategies.

This chapter shares material with the corresponding publication [P5].

7.1 Motivation

Regular Expression Denial of Service (ReDoS) [113, 209, 426] is a type of denial of service (DoS) attack [192, 360] that exploits inefficiencies in regular expression (regex) engines. Many regex engines have worst-case exponential backtracking behavior. By crafting input strings that trigger this behavior, attackers can cause systems to consume disproportionate CPU resources, leading to service disruptions. ReDoS threatens real systems: slow regex matches caused outages at Stack Overflow [465] and Cloudflare [211]. ReDoS is the fourth most common server-side vulnerability class in JavaScript's npm ecosystem—only path traversal, prototype pollution, and command injection appear more often [54]. It is also among the fastest-growing vulnerabilities in the same ecosystem [301]. Consequently, there has been much research on ReDoS, with dozens of papers since 2015 (§7.3). However, as yet there has been no systematization of knowledge about ReDoS, making it challenging to consolidate existing knowledge, identify gaps, and guide future efforts to address this complex and evolving threat.

To address this gap, we conducted a comprehensive literature review of recent papers on ReDoS (§7.3). Our goals were to provide a tutorial on regexes and ReDoS, summarize previous work, assess progress in the field, and identify future work opportunities. We analyze 35 papers on ReDoS vulnerabilities from top security conferences, focusing on detection, prevention, and mitigation strategies, to provide a comprehensive overview of the current state of the field. One primary observation from our review was that almost all prior works rely on at least one of two main models of regex engine behavior: the first introduced in 1968 by Thompson [484], and the second in 1994 by Spencer [455]. While these models have been instrumental in shaping foundational research, they do not fully reflect modern regex engines, many of which now incorporate defenses against ReDoS attacks. ReDoS research must be situated within real regex engine implementations, motivating us to perform an updated engineering review.

Thus, we conducted an engineering review of modern regex engines to address this disconnect between academic research and practical applications and update the understanding of the capabilities of regex engines (§7.4). This review examines the implementations in the major programming languages to evaluate how they handle ReDoS vulnerabilities, either through new algorithms, resource caps, or other mitigations. By analyzing these engines' designs, defenses, and real-world adoption, we provide a detailed account of the current state of regex processing and its implications for ReDoS research. This engineering perspective complements the findings of our literature review and highlights the differences between traditional regex engines and contemporary regex engine behavior, offering up-to-date insights for future studies.

As a result of our synthesis of academic knowledge and engineering analysis, we discuss five main findings (§7.5). In our review of the academic literature, we identified areas where threat models, ReDoS definitions, and attack evaluations could benefit from further refinement and alignment. In our engineering review of regex engines, we note that four major engines have used three distinct algorithms to mitigate ReDoS, but these defenses do not always reduce the time complexity to linear. We encourage the research community to explore opportunities to improve conceptual clarity in future ReDoS research and to focus efforts on evaluating and advancing state-of-the-art ReDoS defenses.

Our primary contributions are:

- We analyze 35 papers on ReDoS vulnerabilities from top security conferences, critiquing their definitions, methodologies, and assumptions. By identifying gaps in the literature, we provide a foundation for more practical and impactful research.

- We examine the current posture of the major regex engines, highlighting recent improvements, persistent vulnerabilities, and the trade-offs involved in mitigating ReDoS.

- Based on our findings, we propose guidelines for researchers to align their work with real-world needs and for practitioners to better evaluate and mitigate ReDoS risks.

7.2 ReDoS: Threat Model and Prevalence

Building on the regular expression and regex engine foundations described in §2.3, this section defines the threat model used throughout our literature review (§7.3). We focus on the conditions that give rise to ReDoS vulnerabilities and quantify their prevalence across modern software ecosystems to characterize the impact of the threat.

7.2.1 ReDoS in Principle

Regex engines often favor expressive power over predictable worst-case efficiency, and that engineering trade-off leaves them open to ReDoS [113]. The Common Weakness Enumeration catalog labels the problem CWE-1333: *Inefficient Regular Expression Complexity* [332]. ReDoS is an example of an algorithmic complexity attack [114], characterized by the small cost to create a malicious input and the large cost to process it. Algorithmic complexity attacks are one kind of asymmetric DoS attacks [316], where the asymmetry comes from the worst-case performance of the data structures and algorithms used to implement the system.

Following Davis *et al.*'s [129] definition, the threat model for a ReDoS attack has three ingredients:

1. *A Backtracking Regex Engine:* The regex engine employed for pattern matching in the victim system utilizes a backtracking approach (§2.3.2). Many regex engines implemented in programming languages (e.g., JavaScript, Java, Python) have at some point met this criterion (§7.4).

2. *A Vulnerable Regex:* The victim system uses a *super-linear*, *problematically ambiguous regex*; for which the attacker crafts *attack strings* that take polynomial or exponential time to match in the string length [543, 525].

3. *An Adversary-Controlled Input:* The attacker supplies inputs that result in attack strings reaching the vulnerable regex and triggering its worst-case time complexity.

Given these three ingredients, if the system has insufficient mitigations (safeguards) in place, then the attacker can cause the victim system to consume excessive computational resources (e.g., CPU and memory). The attacker may need to tailor the number of pumps based on the length of typical input (to avoid anomaly detection) or the maximum length allowed (to avoid truncation).

7.2.2 ReDoS in Practice

In recent years, ReDoS has become a common security concern. Two major web services had outages that resulted from slow regex behavior: Stack Overflow in 2016 [465] and Cloudflare in 2019 [211]. While these outages did not involve an adversary (*i.e.,* they were reliability issues, not security issues), they made service providers aware of the risk of ReDoS. Since 2020, the security company Snyk has observed a jump in reported ReDoS vulnerabilities, especially in the JavaScript package ecosystem npm [301]. In

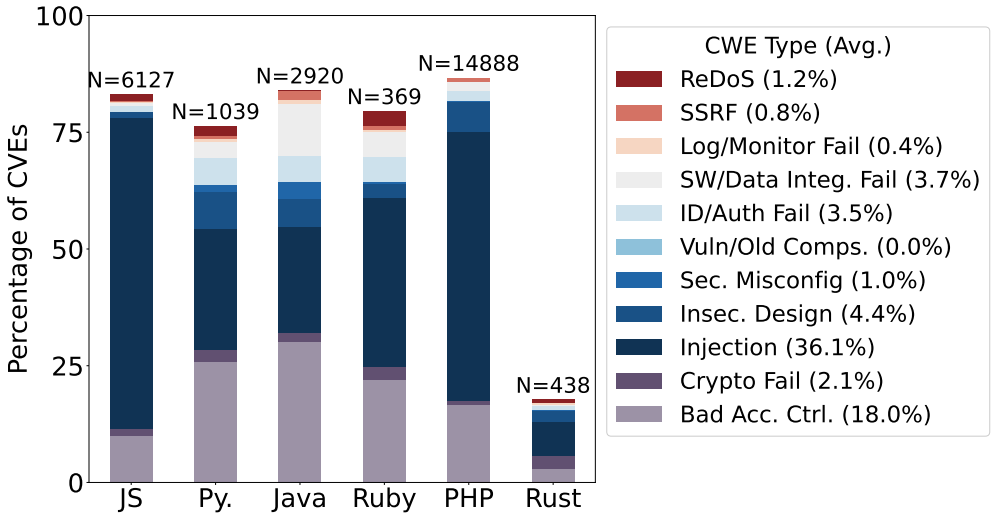


Figure 7.1: Distribution of OWASP Top 10 and ReDoS CVEs per ecosystem, showing average prevalence per category. N indicates the total number of CVEs recorded for each ecosystem. The data cover 2014-2023.

parallel with this surge, a rich body of grey literature has emerged to offer practical guidance on repairing ReDoS vulnerabilities [303, 33, 122, 208].

To quantify the formal disclosure of ReDoS vulnerabilities in practice, we conducted a comprehensive analysis of National Vulnerability Database (NVD) CVE records from 2014 to 2023. Following the keyword-based methodology of Hassan et al. [234], we examined CVEs across six major programming language ecosystems, focusing on OWASP Top 10 vulnerability categories. For each category, we relied on the CWE field in CVE reports and counted CVEs that referenced CWE entries associated with the corresponding OWASP category. To accurately identify ReDoS CVEs, we employed a systematic two-stage keyword search:

- **Stage one keywords:** `regex`, `regular expression`, `regexp`
- **Stage two keywords:** `algorithm`, `backtrack`, `uncontrolled`, `repetition`, `repeat`, `infinite`, `denial of service`, `dos`, `infinite loop`, `algorithmic complexity`

This method effectively identified ReDoS-related CVEs, and manual verification of a sample confirmed no false positives. We then categorized the identified CVEs by ecosystem using the following keyword sets:

- **npm:** npm, node.js, nodejs, javascript, js
- **PyPI:** pypi, python, pip, django, flask
- **Gems:** rubygems, ruby, gems, rails
- **Maven:** maven, java, spring

- **Packagist:** packagist, composer, php, laravel
- **Cargo:** cargo, rust, crates.io, rust-lang

We again manually verified samples from each ecosystem and found no false positives, confirming the accuracy of our categorization. We identified $\tilde{4}00$ ReDoS-related CVEs disclosed since 2014. As a reference point, we compared these results against CVEs associated with OWASP Top 10 vulnerability classes [172]. Averaged over the 2014–2023 period, Figure 7.1 shows that the disclosure rate of ReDoS vulnerabilities exceeds that of four OWASP Top 10 categories.

7.3 Academic Research on ReDoS

Contemporaneous with the increasing industry recognition of ReDoS, ReDoS has become a common subject of cybersecurity research. However, there has been no systematization of the resulting knowledge. We therefore conducted a systematic literature review of ReDoS papers published between 2015 and 2024 at leading conferences in computer security (S&P, USENIX, CCS, NDSS), software engineering (ICSE, FSE, ASE), and programming languages (PLDI, POPL, OOPSLA). Following the structured methodology outlined by Schloegel et al. [441], we identified relevant papers in conference proceedings via an initial keyword search conducted in June 2024, using the terms `ReDoS`, `denial-of-service`, `regex`, `regular expression`, `regular expression analysis`, and `regular expression execution`. This yielded 93 papers, which were further filtered via the following criteria:

- **Inclusion Criteria:** Papers that primarily address ReDoS, propose new attacks or defenses against ReDoS, or extensively study existing ReDoS attacks or defenses.
- **Exclusion Criteria:** Papers that focus on general DoS or those that discuss regexes, without specifically addressing ReDoS.

Some closely related papers were excluded because they do not focus on ReDoS-specific vulnerabilities or because their evaluations lack ReDoS-specific aspects. These exclusions were necessary to ensure that the review remains precise and focused on ReDoS. For example, [111, 502, 443] were excluded because they primarily address broader DoS vulnerabilities or general algorithmic complexity attacks. Other papers were excluded because they evaluate performance characteristics of regex engines without demonstrating their relevance to ReDoS vulnerabilities.

Table 7.1 presents representative examples of these excluded works along with the reasons for their exclusion, illustrating how we deliberately refined the scope of the review to focus on ReDoS-specific contributions.

To ensure informed inclusion and exclusion decisions, we reviewed the abstract, introduction, and evaluation sections of each paper. We identified 35 papers that met our inclusion criteria and systematically analyzed them along four key dimensions:

1. **Research Directions**: Captures the primary focus of each study, which typically falls into one of three categories:

Table 7.1: Examples of excluded works with explanations.

Work	Reason for Exclusion
Cox [111]	Evaluation does not address ReDoS-specific scenarios.
Varatalu et al. [502]	Evaluation does not address ReDoS-specific scenarios.
Chattopadhyay et al. [90]	Evaluation does not address ReDoS-specific scenarios.
Mamouras et al. [314]	Evaluation does not address ReDoS-specific scenarios.
Shan et al. [443]	Addresses general DoS attacks.
Wei et al. [323]	Addresses general DoS attacks.

- *Detection*: Identifying and analyzing vulnerabilities.
- *Prevention*: Addressing root causes by improving regex engines or limiting unsafe constructs.
- *Mitigation*: Reducing attack impact through runtime defenses or repair mechanisms.

2. **Threat Models**: Examines the assumptions about attacker capabilities (e.g., control over input) and system behavior (e.g., rate-limiting or regex engine design), highlighting the practical applicability of each study.

3. **Vulnerability Definitions**: Focuses on how papers describe what constitutes a ReDoS vulnerability or a vulnerable regex (e.g., having exponential matching time).

4. **Evaluation Methods**: Assesses the languages and regex engines used for testing, providing insights into the platforms where these approaches are validated.

Additionally, we reviewed other closely related works that, while not meeting our strict inclusion criteria, are still highly relevant to ReDoS. For instance, papers such as [111, 246, 451] propose improved regex matching algorithms. Although mitigating ReDoS was not their primary focus, these works provide valuable insights and serve as foundational building blocks for ReDoS defenses, making them significant for academics and researchers in the field.

We begin by discussing the main existing research directions (§7.3.1), distinguishing between detection, prevention, mitigation, and other related works. We then explore the assumptions made about the attackers' capabilities or the underlying systems (§7.3.2), along with the definitions of ReDoS vulnerabilities or successful attacks provided in prior works (§7.3.3). Finally, we discuss how the studied papers were evaluated (§7.3.4).

7.3.1 Main Existing Research Directions

Figure 7.2 provides an overview of the main research directions in the study of ReDoS vulnerabilities, categorized into detection, prevention, mitigation, and complementary studies. Detection focuses on identifying ReDoS vulnerabilities in regexes and understanding their exploitability. Prevention aims to address the root causes of ReDoS

vulnerabilities by improving regex engines or restricting vulnerable constructs. Mitigation, on the other hand, focuses on minimizing the impact of ReDoS attacks by deploying runtime safeguards. We discuss these categories in detail below.

7.3.1.1 Detection

Detection is a critical area of research in ReDoS, with studies focusing on identifying vulnerable regexes across various ecosystems. Nine papers analyze open-source package ecosystems, such as npm [54, 459], pip [127, 521], and Maven [521, 296], while seven focus on testing live web applications [459, 40]. Additionally, 27 papers investigate ReDoS vulnerabilities using curated regex datasets [491, 307, 450, 449]. We categorize detection methods into five key approaches: *Heuristic-based Detection* (three papers), *Automata-based Detection* (seven papers), *Dynamic Detection* (four papers), *Hybrid Detection* (five papers), and *Machine Learning (ML)-based Detection* (one paper). Heuristic-based techniques rely on simple pattern rules to flag problematic regexes, offering speed but often at the cost of high false positive rates. Automata-based approaches, which model regexes using finite-state machines, provide formal accuracy but are computationally expensive. Dynamic detection evaluates regex behavior on specific inputs, revealing real-world vulnerabilities but requiring significant runtime resources. Hybrid methods combine static and dynamic techniques to balance precision and efficiency, while emerging ML approaches use regex datasets to predict vulnerabilities, opening new avenues but facing challenges like adversarial examples and limited training data. This categorization highlights the trade-offs in speed, accuracy, and false positive/negative rates among different techniques, offering a structured view of existing detection strategies and their potential for future enhancement.

Heuristic-based Detection (3 studies): There are many approaches that employ simple pattern-matching techniques to identify potentially vulnerable regexes. Studies by Kluban et al. [276, 275] and Davis et al. [123] utilize tools like regexploit [308], redos-detector [261], and safe-regex [125] to identify these vulnerabilities by searching for constructs such as infinite repeats (`a*b*a*`), branches ($a|b$), or nested quantifiers (`*`, `+`, `?`, `{n,m}`), which are often associated with ReDoS vulnerabilities. While these methods are fast, enabling the rapid analysis of large numbers of regexes, they typically focus on syntax rather than semantics, leading to potential accuracy issues. For instance, Parolini et al. [377] demonstrated that while tools like regexploit achieved low false positive and negative rates, others like safe-regex and redos-detector exhibited significantly higher false positive rates. This highlights the primary limitation of heuristic-based approaches, which often require further analysis to confirm vulnerabilities.

Automata-based Static Analysis (6 studies): Automata-based static analysis is a powerful approach for detecting problematic regexes without requiring program execution. This method has been explored in several studies [408, 273, 377, 491, 469, 525], which employ automata models such as NFAs, DFAs, and enhanced automata to evaluate regex semantics and detect vulnerabilities that lead to ReDoS attacks.

NFA-based methods, such as those by Kirrage et al. [273] and Rathnayake et al. [408], construct NFAs to analyze patterns like `(prefix, pumpable string, suffix)` and identify vulnerabilities caused by exponential backtracking. Building on

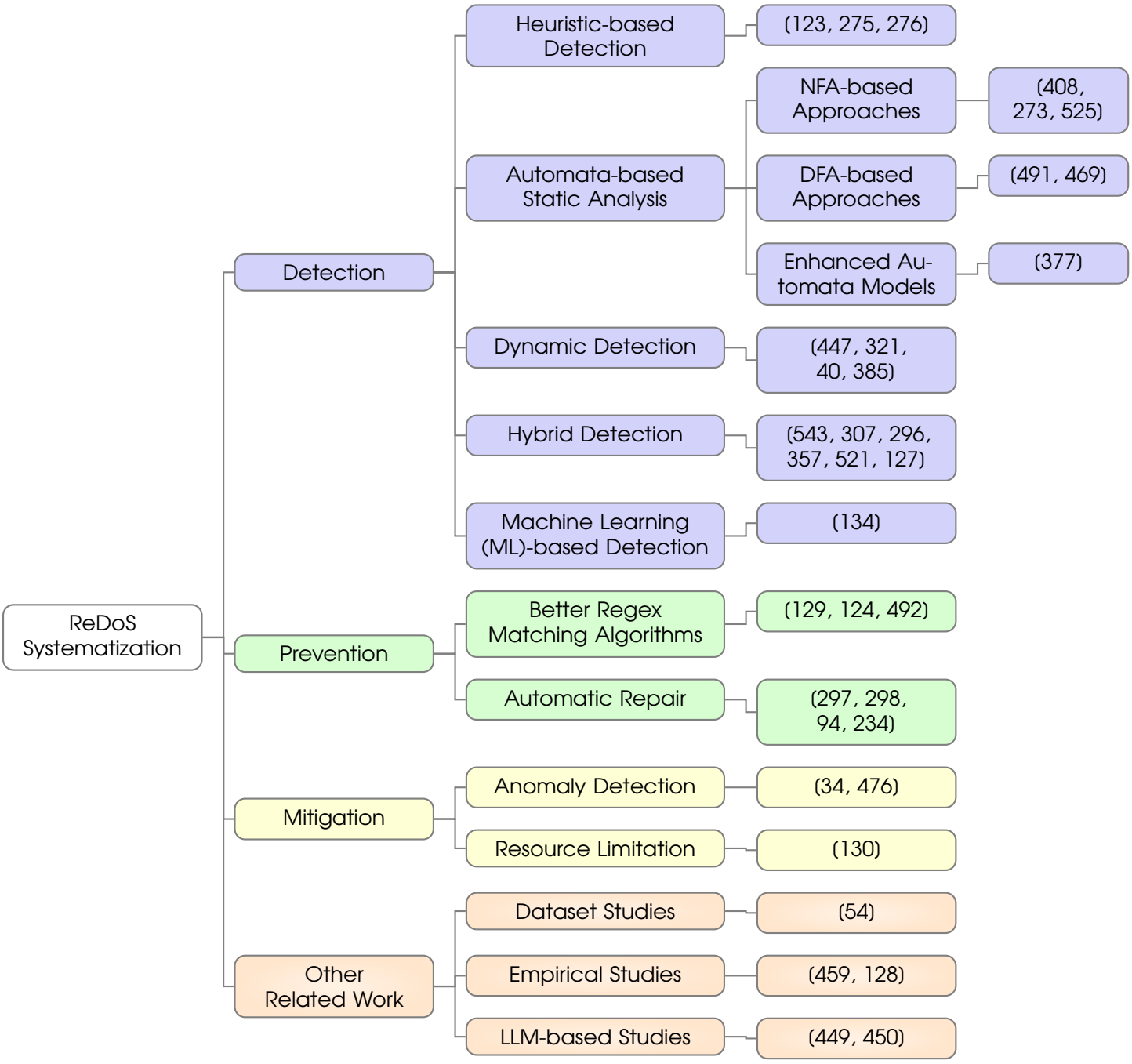


Figure 7.2: Systematization of ReDoS research papers categorized into detection, prevention, mitigation, and related studies, with subcategories and key references.

this, Weideman et al. [525] enhance NFAs with prioritization (pNFA) to reduce redundant state exploration, providing a more precise analysis of backtracking vulnerabilities.

DFA-based approaches, as demonstrated by Turoňová et al. [491] and Shen et al. [469], rely on deterministic automata simulations to evaluate vulnerabilities in non-backtracking engines. By incorporating specialized counting mechanisms, these approaches effectively detect problematic patterns, such as bounded quantifiers and repetitive constructs, while optimizing resource usage. This focus on determinism allows for efficient analysis of regexes without the ambiguity associated with backtracking.

Enhanced automata models offer a novel perspective on regex mathing. Parolini et al. [377] introduce tree semantics, extending beyond traditional NFA and DFA approaches by capturing the structural interactions of regex engines. This enables the detection of vulnerabilities that are difficult to represent in standard automata.

While these methods excel in early detection, they face challenges such as false positives from over-approximations, false negatives from under-approximations, and high computational complexity for analyzing intricate regex structures. Despite these trade-offs, automata-based static analysis remains a foundational tool for addressing ReDoS vulnerabilities.

Dynamic Analysis (4 studies): Dynamic approaches [447, 321, 40, 385] to detecting ReDoS vulnerabilities involve executing regexes with various inputs and analyzing their runtime behavior to identify potential issues. For instance, Shen *et al.* [447] developed ReScue, a tool that uses an evolutionary genetic algorithm to generate time-consuming strings capable of triggering ReDoS vulnerabilities. Similarly, Barlas et al. [40] and McLaughlin et al. [321] utilized dynamic analysis with fuzzing techniques, while Petsios et al. [385] introduced SlowFuzz, which uses automated fuzzing to find inputs causing worst-case algorithmic behavior.

Hybrid Approaches (6 studies): Existing static analysis methods for detecting ReDoS vulnerabilities often face a trade-off between precision and recall. For example, Davis et al. [127] and Shen et al. [447] reported low F-1 scores of 44.94% and 3.57%, respectively [296]. To address these limitations, dynamic validation is frequently employed alongside static analysis, resulting in hybrid approaches. Tools such as NFAA [543], Revealer [307], ReDoSHunter [296], Badger [357], and RENGAR [521] combine static and dynamic methods to mitigate false positives. For instance, hybrid approaches like ReDoSHunter integrate static and dynamic techniques to improve detection accuracy, while tools such as NFAA refine dynamic testing with insights from static analysis [543]. These methods effectively balance the strengths of both static and dynamic detection, reducing the likelihood of inaccurate results.

Machine Learning (ML)-based Detection (1 study): ML techniques are also gaining traction in ReDoS detection. Unlike hybrid methods, which explicitly integrate static and dynamic analysis, ML approaches aim to predict vulnerabilities and detect anomalies based on patterns in data. For example, Demoulin et al. [134] introduced a semi-supervised learning model to analyze resource utilization and identify potential vulnerabilities. This method triggers alerts and provides mitigation strategies by detecting anomalies in resource consumption. While ML approaches offer promise, they face challenges such as reliance on labeled data, vulnerability to adversarial inputs,

and difficulty in addressing context-specific patterns that may not generalize well across diverse regex use cases.

7.3.1.2 Prevention

Prevention strategies focus on addressing the root causes of ReDoS vulnerabilities, either by improving regex matching algorithms or by repairing vulnerable regex patterns. These approaches aim to prevent ReDoS attacks entirely by designing systems that eliminate worst-case performance scenarios, thus avoiding the need for runtime detection or mitigation.

Better Regex Matching Algorithms (3 studies): Researchers have proposed developing new regex matching engines to address ReDoS vulnerabilities at a fundamental level. Davis et al. [129, 124] introduced memoization-based optimization to speed up regex matching by caching intermediate results, significantly reducing redundant computations. Turoňová et al. [492, 491] proposed novel counting automata matching algorithms that further optimize counter management during matching to efficiently handle regexes with bounded repetitions. These new engines often employ finite-state machine-based approaches instead of backtracking, inherently avoiding the exponential time complexity that leads to ReDoS vulnerabilities. Additionally, some methods impose restrictions on regex features to ensure faster and safer processing.

Automatic Repair (4 studies): Automatic repair techniques aim to proactively address vulnerabilities in regexes to mitigate ReDoS attacks. Several studies have explored programmatic frameworks to detect and repair problematic regex patterns. For example, Li et al. [298] developed FlashRegex, a programming-by-example (PbE) tool that repairs or synthesizes regex from provided matching examples. However, FlashRegex struggles with complex regex features, such as lookarounds and backreferences [297]. To address these limitations, RegexScalpel [297] employs a localize-and-fix approach, which identifies fine-grained vulnerabilities and applies predefined repair strategies to improve recall and precision. In a complementary effort, Hassan et al. [234] focus on identifying specific regex structures or "anti-patterns" that are prone to ReDoS vulnerabilities. Their method analyzes regex patterns to uncover and repair potential vulnerabilities early in the development process, avoiding the need for runtime execution. This approach complements programmatic frameworks by systematically identifying and addressing problematic constructs.

Other tools, such as Chida et al. [94], also leverage PbE algorithms but face challenges related to dependency on user-provided examples, which can result in incomplete repairs [296, 521]. These methods, while effective for certain regex patterns, often struggle with advanced or highly customized regex constructs, underscoring the need for more comprehensive solutions capable of addressing complex vulnerabilities. Moreover, improving the generalization and robustness of automatic repair frameworks remains a key area of ongoing research.

Prevention techniques address ReDoS vulnerabilities by improving the efficiency and resilience of regex engines or by repairing vulnerable regex patterns. While they significantly reduce the risk of attacks, these methods require careful configuration to ensure compatibility with complex patterns and real-world scenarios.

7.3.1.3 Mitigation

Mitigation strategies aim to detect ReDoS attacks in live systems and mitigate their impact in real-time. These approaches focus on monitoring runtime behavior to identify anomalies in resource consumption, such as excessive memory usage or prolonged processing times, and applying defensive measures to minimize the effects of ongoing attacks. Unlike prevention, which aims to eliminate vulnerabilities before execution, mitigation provides a dynamic response to attacks as they occur.

Anomaly Detection (2 studies): Anomaly detection techniques are central to mitigation strategies, leveraging real-time monitoring to identify unusual patterns indicative of ReDoS attacks. ML-based approaches have proven effective in this domain [34, 476]. For example, Tandon et al. [476] employ the Elliptic Envelope model to analyze runtime features such as request time, memory usage, and CPU cycles, enabling the system to detect deviations from normal behavior. Bai et al. [34] introduce a feedback loop mechanism that continuously adjusts defensive measures based on real-time performance metrics, providing robust protection against zero-day ReDoS attacks.

While promising, ML-based anomaly detection methods are not without challenges. For instance, Tandon et al. [476] report a false positive rate of 1.3% and a false negative rate of 0.4%, which could still impact practical deployment. Furthermore, these systems can be vulnerable to adversarial attacks, where carefully crafted inputs are designed to bypass detection mechanisms [34]. These limitations underscore the need for more resilient and adaptive approaches to anomaly detection, ensuring robustness against both known and emerging threats.

Limiting Resource Consumption (1 study): To mitigate ReDoS attacks, Davis et al. [130] proposed using first-class timeouts as a defense against Event Handler Poisoning (EHP) in Node.js. Their prototype, `Node.cure`, effectively limits processing time with minimal performance overhead. Similarly, runtime environments like PHP, Perl, and .NET have adopted resource-limiting mechanisms to curb backtracking and excessive processing. In PHP, the pcre.backtrack_limit and pcre.recursion_limit directives control the number of backtracks and recursion depth during regex matching [386]. Perl's PCRE2 library allows setting evaluation limits via the eval function, with a default recursion depth limit of 10M. In .NET, the RegexOptions.MatchTimeout property enables developers to specify a maximum regex processing duration. While these strategies effectively cap resources to reduce ReDoS risks, they are often too coarse-grained, making it challenging to balance performance and avoid false positives in complex regex patterns or large datasets.

7.3.1.4 Other Related Work

Beyond approaches for addressing ReDoS directly, several complementary studies provide valuable insights into datasets, real-world analyses, and novel frameworks.

Dataset Studies (1 study): Bhuiyan et al. [54] introduced SecBenchJS, a benchmark dataset for evaluating the security of JavaScript applications, including regex vulnerabilities. Such datasets enable standardized testing and improve ReDoS detection tools.

Empirical Studies (2 studies): Staicu et al. [459] analyzed 2846 popular websites, finding 339 vulnerable to ReDoS attacks, highlighting the risks of server-side JavaScript. Davis et al. [128] studied regex portability across eight programming languages, uncovering semantic and performance differences in 25% of regexes. These studies underscore the need for safer regex practices and consistent regex engine implementations.

LLM-based Studies (2 studies): Siddiq et al. [449] proposed Re(gEx|DoS)Eval, a framework for evaluating LLM-generated regexes using metrics like pass@k and vulnerable@k, demonstrating its use on T5, Phi-1.5, and GPT-3.5-Turbo. In another study, Siddiq et al. [450] categorized vulnerable regex patterns and analyzed developer discussions, finding that GPT-3.5-Turbo often generates regexes with polynomial vulnerabilities. These studies provide benchmarks, empirical data, and frameworks that complement prevention, detection, and mitigation strategies, paving the way for robust ReDoS solutions.

7.3.2 Threat Models

A realistic threat model is essential in any security study, as it provides a framework for assessing the potential findings. That is, it enables the identification of the system's weaknesses (*i.e.,* vulnerabilities) that adversaries could exploit in real-world environments. A strong threat model might lead to unrealistic findings, assuming too powerful attackers. In Table 7.2, we outline the granularity at which different threat models are defined in the analyzed papers. While most studies examine the exploitability of regexes in isolation, some do extend their analysis to the surrounding context, such as within library code. However, only a few studies define threat models at the application level. Threat models defined at the code fragment level can lead to false positives—vulnerabilities that are exploitable in isolated regexes but unreachable in actual code.

Among the papers we reviewed, only 10 (28%) explicitly define a threat model, while the rest implicitly assume one. We identify four assumptions in the existing threat models:

Attacker Controls Input (35 studies): The assumption that an attacker can control inputs to a vulnerable ReDoS location is crucial for exploitability, as highlighted in various studies [40, 234]. In real-world software, including web applications and systems, inputs often originate from untrusted sources, thereby increasing the potential for exploitation [385]. However, whether these inputs can reach the vulnerable regex depends on the surrounding code. All the reviewed papers, whether their threat models focus on the application [459, 40, 543, 134], library [54, 521, 321, 296], or code fragment level [123, 491, 307], assume that the attacker can manipulate the input to exploit the vulnerability in the regex. However, this assumption may not always be applicable. For instance, consider a library that only reads and parses a configuration file on a server, which is not controlled by an attacker. In such cases, the likelihood of an attacker being able to influence the input and exploit the ReDoS vulnerability is significantly reduced, if not eliminated.

Attacker's Input Reaches Super-Linear Regex (34 studies): This condition necessitates that the attacker's input be matched against a specific type of regex.

Table 7.2: Overview of ReDoS threat models in prior work. Each column shows whether the study analyzes code fragments (standalone regex snippets), use within libraries, or full applications, and whether it assumes the attacker can send an arbitrary number of requests. The symbol ● denotes the paper covers that aspect in detail; – means it does not.

Paper	Code Fragment	Library	Application	Arbitrary Number of Requests
Su et al. [469]	●	–	–	100kB
Davis et al. [123]	●	–	–	–
Turoňová et al. [491]	●	–	–	1.5K–9K
Liu et al. [307]	●	–	–	128
Staicu et al. [459]	–	●	●	82K
Davis et al. [127]	●	●	–	100K–1M
Siddiq et al. [450]	●	–	–	–
Shen et al. [447]	●	–	–	128
Barlas et al. [40]	–	–	●	–
Siddiq et al. [449]	●	–	–	–
Davis et al. [128]	●	–	–	–
Kluban et al. [275]	●	–	–	–
Bhuiyan et al. [54]	●	●	–	–
Noller et al. [357]	●	–	–	–
Wang et al. [521]	●	●	–	–
Rathnayake et al. [408]	●	–	–	–
McLaughlin et al. [321]	●	●	–	1M
Li et al. [296]	●	●	–	–
Parolini et al. [377]	●	–	–	128
Kirrage et al. [273]	●	–	–	–
Wüstholz et al. [543]	●	–	●	140K
Petsios et al. [385]	●	–	–	–
Kluban et al. [276]	–	●	–	–
Demoulin et al. [134]	–	–	●	–
Chida et al. [94]	●	–	–	–
Bai et al. [34]	–	–	●	50K
Li et al. [297]	●	●	–	1M
Davis et al. [124]	●	–	–	–
Li et al. [298]	●	–	–	–
Hassan et al. [234]	●	–	–	–
Davis et al. [130]	–	●	●	–
Davis et al. [129]	●	–	–	10K–100K
Tandon et al. [476]	–	–	●	–
Weideman et al. [525]	●	–	–	–
Turoňová et al. [492]	●	–	–	500K–50M

While all the papers considering threat models at the library and code fragment levels implicitly assume this condition, it is a significant assumption. With the exception of Barlas et al. [40], most studies presume this might be achievable in practice. In reality, achieving this could require the attacker to have access to detailed information about the server-side logic, API documentation, or the internal workings of the web service. Staicu et al. [459] incorporate web server implementations into their threat model and examine whether the input can trigger the vulnerability. However, this fingerprinting approach can be noisy, leading to both false positives and false negatives.

Use of Backtracking or Slow Regex Engine (33 studies): This condition assumes the attacker has in-depth knowledge of the specific regex engine employed by the server. On one hand, it might be feasible to remotely determine the type of engine in use (e.g., PCRE, RE2), but pinpointing the exact version with its specific weaknesses can be quite challenging. On the other hand, assuming a slow or outdated engine might lead to inaccurate results, making the attack scenario unrealistic in many real-world situations. Except for Barlas et al. [40] and Turoňová et al. [491], all the studies make this assumption as part of their threat model.

Arbitrary Request Assumption (32 studies): This condition represents a strong assumption that the attacker can repeatedly send requests with large input lengths without interference from other security mechanisms. Most studies, with the exceptions of [447, 307, 377], either disregard restrictions on request size and frequency or necessitate large inputs to trigger vulnerabilities. For context, the maximum size of HTTP header requests and responses in `nginx` [98] and `Apache Tomcat` [169] is 8K, which is significantly smaller than the input lengths assumed in most works. For instance, Davis et al. [127] require input lengths of 100K-1M to trigger a 10-second slowdown, which would be impractical in real-world system settings. Furthermore, most studies fail to consider potential mitigation strategies employed by real-world systems, such as rate limiting, intrusion detection systems, or firewalls, which can significantly impede attacks. In realistic scenarios, attackers might face various system-imposed limitations, including bandwidth constraints or restrictions on the number of requests [434] they can send. As a result, the majority of studies, except for [447, 307, 469], assume in their threat models that attackers can perform requests unhindered, which may not reflect real-world conditions.

While the inclusion of a realistic threat model is essential for evaluating security findings accurately, many existing ReDoS studies make unrealistic, strong assumptions about attacker capabilities and system configurations.

7.3.3 ReDoS Definitions

Most of the considered papers define a ReDoS vulnerability in terms of the slowdown that can be caused directly, with a limited number of characters. This variation is summarized in Table 7.3. Notably, there is no consensus on what constitutes a vulnerability. For instance, some studies establish thresholds for the number of allowed operations during matching that differ significantly. Liu et al. [307] and Wang et al. [521] use a threshold of 10^5 instructions, while Shen et al. [447] and Siddiq et al. [449] consider 10^8 instructions, and Parolini et al. [377] set it at 10^{10} instructions. Adding to the confusion, Sung et

Table 7.3: Summary of ReDoS vulnerability definitions in reviewed studies. The literature uses inconsistent criteria, with over half (19/29) defining ReDoS based on slowdown. ◐ denotes incomplete details; ● indicates use of heuristics; – means the study did not use or report that criterion.

Paper	Slowdown	Number of Instructions	Heuristics
Su et al. [469]	–	>10000	–
Davis et al. [123]	–	–	●
Turoňová et al. [491]	10s, 100s	–	–
Liu et al. [307]	–	10^5	–
Staicu et al. [459]	5s	–	–
Davis et al. [127]	10s	–	–
Siddiq et al. [450]	1s	–	–
Shen et al. [447]	–	10^8	–
Barlas et al. [40]	1s	–	–
Siddiq et al. [449]	1s	10^8	–
Davis et al. [128]	10s	–	–
Kluban et al. [275]	–	–	●
Bhuiyan et al. [54]	2s	–	–
Wang et al. [521]	–	10^5	–
McLaughlin et al. [321]	10s	–	–
Li et al. [296]	1s	–	–
Parolini et al. [377]	–	10^{10}	–
Wüstholz et al. [543]	◐	–	–
Petsios et al. [385]	◐	–	–
Kluban et al. [276]	–	–	●
Demoulin et al. [134]	◐	–	–
Chida et al. [94]	–	–	●
Bai et al. [34]	$\mu + 3\sigma$	–	–
Li et al. [297]	10s	–	–
Li et al. [298]	◐	–	–
Hassan et al. [234]	–	–	●
Davis et al. [130]	1s	–	–
Davis et al. [129]	◐	–	–
Turoňová et al. [492]	10s, 100s	–	–

al. [472] define vulnerabilities at 10^6 and 10^7 instructions. It remains unclear why these values are deemed appropriate, as heavy CPU loads do not always indicate the presence of a vulnerability.

Moreover, the inconsistency extends to the measurement of introduced slowdowns. Various papers use different metrics to quantify slowdowns, further complicating the landscape and hindering comparability across studies. Turoňová et al. [491] measure slowdowns in $10s$ and $100s$, Staicu et al. [459] in $5s$, Davis et al. [127, 128] in $10s$, Siddiq et al. [450], Li et al. [296] and Barlas et al. [40] in $1s$, Bhuiyan et al. [54] in $2s$. Meanwhile, McLaughlin et al. [321] use $10s$, and Bai et al. [34] define a problematic slowdown as the average response time (μ) plus three times the standard deviation (3σ), to account for significant deviations from the norm. These excessively high thresholds likely miss many potential vulnerabilities that could cause significant issues under real-world conditions. In our experiments, we saw that under one-second slowdowns can be weaponized against a target server, with modest attacker resources [55]. Other papers rely on heuristics to define a vulnerability. For example, Davis et al. [123] utilize safe-regex [125], which uses static analysis rules to determine whether a regex is vulnerable. These tools are unsound and report both false positives and negatives.

The lack of consensus on vulnerability thresholds, slowdown metrics, and measurement methodologies highlights significant inconsistencies in ReDoS research, underscoring the need for standardized definitions and evaluation criteria.

7.3.4 Evaluations

We carefully studied the evaluations of the papers in our dataset, as shown in Table 7.4. Our analysis reveals two key aspects regarding ReDoS research:

Language: Regex matching is comparatively slow in programming languages like JavaScript, Java, Python, and Ruby [128], leading to a significant amount of research focusing on detecting and defending ReDoS attacks in these languages. Reflecting this emphasis, 19 (54%) papers address JavaScript, 18 (51%) address Java, and 9 (25%) address Python, with JavaScript being particularly prominent.

Evaluation Methodology: Out of the reviewed studies, only 7 (18%), such as Demoulin *et al.* [134], Tandon *et al.* [476], and Staicu *et al.* [459], performed an end-to-end simulation to evaluate their proposed attack or defense methodologies. Moreover, only Bai *et al.* [34] measured the system's performance degradation during an attack as perceived by benign users. The remaining 29 studies (82%) rely solely on theoretical or localized analysis, which might not capture real-world attack scenarios and defenses accurately, potentially overestimating the prevalence of ReDoS vulnerabilities.

The lack of comprehensive end-to-end evaluations in current research hampers the ability to accurately gauge the real-world effectiveness and impact of ReDoS attacks and defenses.

7.4 ReDoS Mitigations in Regex Engines

In §7.3, we discuss many papers that measure ReDoS or propose defenses. They often assume that the matching of a regex engine can be modeled as a backtracking search

Table 7.4: Summary of evaluation methods used in the reviewed studies. Most works (33/34) measure the success of a ReDoS attack from the perspective of the attacker, rather than from its impact on benign users (*i.e.*, Quality of Service (QoS) degradation). ● indicates that the criterion or platform was explicitly considered; ◐ denotes incomplete details; – means the study did not use or report that criterion.

Paper	Attack Simulation	Measure QoS Degradation	JavaScript	Java	Python	Ruby	PHP	Go	Perl	Rust	C#
Su et al. [469]	–	–	●	●	–	–	●	●	●	●	●
Davis et al. [123]	–	–	●	–	–	–	–	–	–	–	–
Turoňová et al. [491]	–	–	●	●	●	●	●	–	●	–	●
Liu et al. [307]	–	–	●	●	●	●	●	–	–	–	–
Staicu et al. [459]	◐	–	●	–	–	–	–	–	–	–	–
Davis et al. [127]	–	–	●	–	●	–	–	–	–	–	–
Siddiq et al. [450]	–	–	–	●	–	–	–	–	–	–	–
Shen et al. [447]	–	–	–	●	–	–	–	–	–	–	–
Barlas et al. [40]	◐	–	●	–	–	–	–	–	–	–	–
Siddiq et al. [449]	–	–	–	●	–	–	–	–	–	–	–
Davis et al. [128]	–	–	●	●	●	●	●	●	●	●	–
Kluban et al. [275]	–	–	●	–	–	–	–	–	–	–	–
Bhuiyan et al. [54]	–	–	●	–	–	–	–	–	–	–	–
Noller et al. [357]	–	–	–	●	–	–	–	–	–	–	–
Wang et al. [521]	–	–	●	●	●	–	–	–	–	–	●
Rathnayake et al. [408]	–	–	–	–	●	–	–	–	–	–	–
McLaughlin et al. [321]	–	–	●	–	–	–	–	–	–	–	–
Li et al. [296]	–	–	–	●	–	–	–	–	–	–	–
Parolini et al. [377]	–	–	–	●	–	–	–	–	–	–	–
Kirrage et al. [273]	–	–	–	●	–	–	–	–	–	–	–
Wüstholz et al. [543]	◐	–	–	●	–	–	–	–	–	–	–
Petsios et al. [385]	–	–	–	–	–	●	–	–	–	–	–
Kluban et al. [276]	–	–	●	–	–	–	–	–	–	–	–
Demoulin et al. [134]	●	–	●	–	–	–	–	–	–	–	–
Chida et al. [94]	–	–	●	–	●	–	–	–	–	–	–
Bai et al. [34]	●	●	●	–	–	–	–	–	–	–	–
Li et al. [297]	–	–	–	●	–	–	–	–	–	–	–
Davis et al. [124]	–	–	●	●	●	●	●	●	●	●	–
Li et al. [298]	–	–	–	●	–	–	–	–	–	–	–
Hassan et al. [234]	–	–	●	●	●	●	●	–	●	–	●
Davis et al. [130]	◐	–	●	–	–	–	–	–	–	–	–
Tandon et al. [476]	●	–	–	–	–	–	●	–	–	–	–
Weideman et al. [525]	–	–	–	●	–	–	–	–	–	–	–
Turoňová et al. [492]	–	–	–	–	–	–	–	–	–	–	●

Table 7.5: Summary of ReDoS defenses in the implementations of major programming languages.

Language (*Implementation*)	Nature of the Defense	Year	LOC to Implement	On by Default?	ReDoS Safe?
JavaScript (*Node.js—V8* [170, 396])	A Non-backtracking Engine: Thompson-style	2021	1558 (10.6%)	✗	Partial
Ruby (*MRI/CRuby* [99])	Memoization, Resource Capping: Time-based	2022	1100 (4.7%)	✓	Partial
C# (*.NET* [326])	Resource Capping: Time-based; Brzozowski Derivatives Engine	2012, 2022	5417 (34.7%)	✗	Partial
Java (*OpenJDK* [369])	Caching	2016	1712 (35.5%)	✓	Partial
PHP (*Zend Engine* [213, 214])	Resource Capping: Counter-based	–	–	✓	Partial
Perl (*perl5* [383])	Caching, Resource Capping: Counter-based	–	–	✓	Partial
Rust (*rustc* [177])	A Non-backtracking Engine: Thompson-style	–	–	✓	Safe
Go (*gc* [480])	A Non-backtracking Engine: Thompson-style	–	–	✓	Safe
Python (*CPython* [173])	–	–	–	✗	Not Safe

(§7.3.2). However, it is unclear whether this assumption holds in practice, particularly in the context of modern regex engines.

In this section, we examine the regex engines of major programming languages to answer two questions: (1) *Do the measurements and assumptions in the academic literature accurately reflect the latest versions of these engines?* and (2) *What is the impact and adoption of ReDoS defenses proposed by researchers in real-world regex engines?*

We answer these questions in two ways. First, a systematic analysis was conducted on regex engines in nine popular programming languages (§7.4.1). This analysis drew upon first-party sources such as source code, issue trackers, language documentation, blog posts from regex engine maintainers, and CVE reports. We also examined academic literature that describes regex engines. Second, we measured whether super-linear regexes improved performance in the latest versions of these engines (§7.4.2).

7.4.1 Regex Engine Analysis

In this section, we begin by outlining the selection criteria used to choose the regex engines for our analysis. Then, for each engine, we examine the type and details of the ReDoS defense(s) implemented (if any), the update context of the defense (including the year of introduction), and the required lines of code (LOC) to implement it. Additionally, we assess whether the defense is enabled by default or requires developer intervention. We summarize our results on evaluating ReDoS defenses in modern regex engines in Table 7.5.

Table 7.6: Regex engine implementations selected for evaluating ReDoS defenses.

PL	Selected Impl. of the PL [Source Code]	Other Alternatives	Reason for Selection
JavaScript	Node.js—V8 [171, 397]	Deno, Bun, SpiderMonkey, JavaScriptCore	Dominates server-side JavaScript usage (98.8% market share) [401]
Ruby	MRI/CRuby [100]	JRuby, Rubinius, mruby, RubyMotion	Reference implementation [99]
C#	.NET [168]	Mono	Reference implementation [108]
Java	OpenJDK [109]	Amazon Corretto, Azul Zulu, Liberica JDK, Eclipse Adoptium...	Most widely used JDK of 2024 with a 20.8% usage rate [350]
PHP	Zend Engine [212]	HHVM, PeachPie, Quercus, Parrot	Standard PHP interpreter is driven by the Zend Engine [213, 214]
Perl	perl5 [384]	N/A (for Perl 5)	Only implementation for Perl 5 [383]
Rust	rustc [176]	mrustc, gccrs	Official and only fully functional compiler for Rust [177]
Go	gc [479]	gccgo, gofrontend, TinyGo, GopherJS, yaegi	Official compiler included in Go releases [480]
Python	CPython [174]	PyPy, Stackless Python, MicroPython, CircuitPython, IronPython, Jython	Reference implementation [173]

Selection Criteria: Many modern programming languages have multiple implementations, runtime environments, engines, interpreters, or compilers that might use different regex engines. For instance, JavaScript has multiple engines for executing the source code, which can be used in different environments, such as V8 in Node.js and SpiderMonkey in Mozilla Firefox. To address this complexity, we applied two key criteria when selecting a specific regex engine for languages with multiple options. First, we prioritized engines with substantial real-world adoption, as these are the most likely targets of ReDoS attacks in practice. Second, in cases where usage data is sparse or fragmented, we selected reference implementations that are officially endorsed and openly maintained, enabling thorough analysis of ReDoS defenses. Table 7.6 summarizes the regex engine implementations reviewed in our analysis. For each language, the table lists the selected implementation, notable alternatives, and the rationale for selection. These choices ensure both practical relevance and sufficient access to implementation details for evaluating ReDoS defenses.

7.4.1.1 JavaScript (*Node.js—V8*)

Original Algorithm: The main JavaScript engine, V8, previously used a backtracking Spencer-style algorithm. It was highly optimized for common-case regexes [215]. It uses an explicit automaton representation, optimizing it and then rendering it as native machine code for execution [110].

ReDoS Defense: In 2021, responding to ReDoS concerns, the V8 developers implemented an extra Thompson-style non-backtracking regex engine [57]. The new engine guarantees linear-time complexity for all supported regexes, but as a trade-off, it does not support some E-regex features. All regexes with unsupported E-regex features still have to use the backtracking engine. This update was introduced in V8 version 8.8, but in Node.js, it remains an experimental feature that is not enabled by default and must be activated via a feature flag. Thus, some Node.js applications may not yet benefit from the new non-backtracking regex engine.

Engineering Effort: The new engine is 1558 LOC. The original was 14741 LOC (excluding platform-specific assembly code). The effort was ∼10% of the original.

7.4.1.2 Ruby (*MRI/CRuby*)

Original Algorithm: Ruby's default regex engine is based on Onigmo [474], a fork of the Oniguruma [282] library, which is a Spencer-style backtracking regex engine.

ReDoS Defense: Until Ruby 3.2 (2022), Ruby's port of Onigmo did not have built-in mitigations against ReDoS. In Ruby 3.2, the developers added two ReDoS defenses [347, 448]. First, based on a community proposal,[1] Ruby implemented Davis et al.'s [129] memoization technique for Spencer's algorithm as a defense against ReDoS attacks. Interestingly, the Ruby developers used full memoization rather than selective memoization despite the higher space complexity. Ruby does not apply memoization when unsupported E-regex features are encountered, instead offering a timeout mechanism similar to C# as the second kind of defense.[2] The memoization defense is on by default.

Engineering Effort: The relevant pull requests added a total of 1100 LOC, compared to the Ruby 3.1.6 engine's footprint of 23384 LOC. The ratio is 5%.

7.4.1.3 C# (*.NET*)

Original Algorithm: The first two regex engines in .NET used backtracking algorithms [105, 104]. One was a simple opcode interpreter. The other used an optimized machine code representation like V8's [105, 487, 103, 223].

ReDoS Defense: In .NET v4.5 (2012), .NET introduced a timeout mechanism to prevent excessive backtracking [107]. The timeout is checked after at most $O(n)$ steps, where n is the length of the input, balancing fidelity with overhead [486, 106]. In .NET 7 (2022), Microsoft added a non-backtracking regex engine [486]. This engine is based on Moseley et al. [335] and Saarikivi et al. [431], with its core algorithm using Brzozowski derivatives and guaranteeing linear-time matches for supported regexes. The new engine raises an error upon encountering unsupported E-regex features, so the developers must use the backtracking engine in such cases. However, both the timeout mechanism and the non-backtracking engine are opt-in features.

[1] See `https://bugs.ruby-lang.org/issues/19104`.
[2] See `https://bugs.ruby-lang.org/issues/17837`.

Engineering Effort: The new engine comprises 5417 LOC. The existing .NET codebase for regexes was 15597 LOC (including multiple engines). Hence, the new engine expands the existing regex engine codebase by 35%. Microsoft researchers also created two research papers.

7.4.1.4 Java (*OpenJDK*)

Original Algorithm: OpenJDK Java's previous regex engine implemented an NFA-based Spencer-style backtracking algorithm for regex matching [370].

ReDoS Defense: As of Java 22 (2024), OpenJDK Java's regex engine does not document defenses against ReDoS. Despite the absence of official documentation, in the OpenJDK source code repository and issue tracker, we identified that some performance updates related to ReDoS issues were introduced in Java 9 (2016).[3] The OpenJDK maintainers introduced a bounded caching mechanism, improving problematic behavior for some ReDoS scenarios. This ReDoS defense is enabled by default and affects all OpenJDK Java versions $\geq$9.

Engineering Effort: Before the update, the regex-related codebase consisted of 4823 LOC. The memoization patch added 1712 lines (including empty lines and docstrings) to the Java 9 source code, which accounted for a 35% expansion.

7.4.1.5 Other ReDoS-Vulnerable Engines

PHP (Zend Engine), Perl (perl5), and Python (CPython) are also known to be using Spencer-style backtracking regex engines [180, 128], which are vulnerable to ReDoS attacks. According to Davis et al. [129], PHP [386] and Perl [310] utilize "counter-based caps" to prevent catastrophic backtracking as an alternative to C# and Ruby's "time-based caps". Python has neither built-in engine optimizations against ReDoS vulnerabilities nor a native timeout-like mechanism for regex evaluations. We have not found any new documented updates or defenses against ReDoS for these languages since the analysis of Davis et al. [129] in 2021.

7.4.1.6 Other ReDoS-Safe Engines

Two major programming languages, Rust (rustc, via its official regex crate) [175] and Go (gc) [481], have regex engines that were developed with ReDoS safety in mind. These engines simulate an explicit automaton representation using Thompson's algorithm [111]. They also do not support many E-regex features. They, therefore, offer linear-time match guarantees in input and automaton length.

Although not included in our analysis, independent third-party regex engines like RE2 (Google) [454] and Hyperscan (Intel) [520, 519] are well-known for ReDoS resilience. RE2, a Thompson-style engine, incorporates DFA state caching optimizations [335]. While generally ReDoS-safe and non-backtracking, RE2 can, in rare cases, exhibit backtracking behavior [129]. Similarly, Hyperscan is another efficient, ReDoS-safe

[3] See `https://github.com/openjdk/jdk/commit/b45ea89`.

Table 7.7: Versions used in the ReDoS defense measurements.

Language	Old Version	New Version
JavaScript (*Node.js—V8*)	v15.14.0	v22.2.0
Ruby (*MRI/CRuby*)	3.1.6	3.3.2
C# (*.NET*)	6.0.420	7.0.407
Java (*OpenJDK*)	8u342	23
PHP (*Zend Engine*)	5.6.40	8.3.7
Perl (*perl5*)	5.8.14	5.38.2
Rust (*rustc*)	1.12.1	1.78.0
Go (*gc*)	1.5.4	1.22.4
Python (*CPython*)	3.6.15	3.12.3

engine, utilizing hardware acceleration alongside Glushkov's NFA construction [193]. Neither engine supports E-regex features that require backtracking.

Overall, these developments in regex engines demonstrate significant progress in mitigating ReDoS risks but also highlight ongoing challenges in balancing the trade-off between safety and usability. Modern regex engines mitigate ReDoS with non-backtracking algorithms like memoization, Thompson-style simulation, or Brzozowski derivatives to assure linear-time matching. However, the benefit comes at the cost of reduced expressiveness. Non-backtracking engines lack support for many E-regex features. Defenses add timeouts or fallbacks for unsupported E-regexes to maintain codebase compatibility.

7.4.2 Updated Measure of ReDoS Vulnerabilities

As indicated in Table 7.5, several major regex engines have recently been updated to mitigate or eliminate ReDoS. We, therefore, wondered what proportion of ReDoS vulnerabilities have been addressed by state-of-the-art engines.

We answered this question through measurements on a sample from the Davis *et al.*'s [128] polyglot regex corpus, comprising ∼500K regexes extracted from open-source repositories across eight programming languages. We used the tool vuln-regex-detector [126], developed by Davis et al. [127] and based on prior work [408, 543, 525, 447], to identify potentially vulnerable regexes from the corpus. From its predictions, we randomly sampled 500 regexes labeled as exponential-time candidates (out of ∼3K predicted examples) and 500 labeled as polynomial-time candidates (out of ∼100K predicted examples). For each sampled regex, we used the same tool to generate corresponding attack input strings.

Then, we evaluated each regex–input pair on the regex engines described in Table 7.5, comparing older engine versions to their most recent counterparts. When ReDoS defenses were available but not enabled by default in the latest version, as in JavaScript and C#, we explicitly enabled them. Table 7.7 lists the evaluated runtime and regex engine

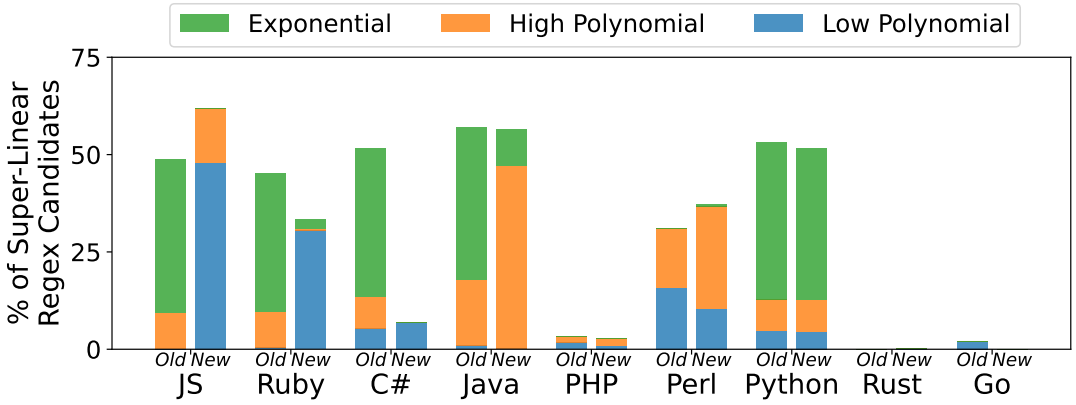


Figure 7.3: Performance of regex matching in old *vs.* latest engines in different programming languages.

versions. The older versions predate known or likely ReDoS-related mitigations, whereas the newer versions incorporate ReDoS defenses or related performance improvements. Following a parallel methodology that has been used in previous work [128, 127], we define a regex's behavior as:

- *Exponential*: The processing time exceeds five seconds with fewer than 500 pumps of an attack string.
- *High-Polynomial*: The processing time exceeds five seconds with 500 or more pumps of an attack string.
- *Low-Polynomial*: The processing time exceeds 0.5 seconds on an attack string but does not meet the exponential or high-polynomial behavior criteria.
- *Linear*: The regex match never times out on the attack string.

Our findings are illustrated in Figure 7.3. For each engine, we depict the number of regexes that exhibited super-linear behavior (as a fraction of the predicted input) in the pre- and post-ReDoS mitigation versions. Not all regexes were supported in all engines, so the candidate pool size is indicated for each engine. Observe that for the engines with mitigations, there is a substantial decrease in the number of viable super-linear regexes—in other words, these mitigations appear to be effective in addressing the common causes of super-linear behavior. Notably, exponential behavior is resolved in C# and JavaScript but persists in Ruby and Java, while polynomial behavior continues to manifest in JavaScript and Java.

7.5 Discussion

In this section, we distill and contextualize the main findings of our study, integrating insights from our literature review (§7.3) and engineering review (§7.4). For each idea, we discuss concrete future work directions for the community to explore.

(1) ReDoS is a peculiar vulnerability: Unlike traditional ones like SQL injections, ReDoS stands out as a contextual issue, more similar to micro-architectural attacks, where exploitability and impact are heavily dependent on the underlying system's architecture and deployment conditions. For example, a regex pattern vulnerable to catastrophic backtracking in older versions of Node.js is rendered safe in newer versions due to advancements in regex engines (§7.4). When detecting ReDoS vulnerabilities, it is not enough to analyze the code of a web application, without considering its deployment configurations. For future static analysis work on ReDoS—particularly those targeting multiple languages such as [234]—we recommend that they consider and model the language-, engine-, and platform-specific details and differences with respect to ReDoS. For example, a super-linear regex might be more serious in a single-threaded web application architecture like Node.js than in inherently parallel ones like Go. For dynamic and hybrid analyses, we recommend a system-level perspective, similar to Shan *et al.* [443], that deploys code under realistic conditions, taking relevant defenses into account.

(2) Strong assumptions about attackers: We find that prior work often makes strong assumptions about attackers (§7.3.2), such as assuming they can inject arbitrarily large payloads or control the input to any regex. Also, many papers neglect defenses or rely on rules of thumb, like regex matches should not exceed one second (§7.3.3). We think that future work should challenge these assumptions by performing empirical studies with real systems, including analyses of how many regexes are actually reachable via attacker-controlled inputs. We also need more empirical studies to show the community how modern applications are deployed in the wild—specifically, which configurations they run on and what defenses they employ.

(3) Developers push back: To the best of our knowledge, there has been no public description of a DoS attack that exploited ReDoS, so the risk posed by these vulnerabilities is still unclear, beyond academic measurement studies [459, 40]. Moreover, only a small fraction of the papers we analyzed (§7.3) discuss such attacks, often in unrealistic setups. We believe that these unclear risks together with problematic security assumptions are the culprits for ReDoS vulnerabilities being a contentious topic among developers. Snyk has stated that ReDoS is the most neglected vulnerability type [302], implying that developers often ignore such findings. Also, engineers at GitHub consider it a low severity vulnerability, "not particularly serious, but easy to create by accident, obscure to understand, and sometimes tricky to fix" [33]. More strident voices argue that ReDoS vulnerabilities are "malicious noise" and just lead to security fatigue [533]. Several ReDoS CVE advisories have been disputed by the maintainers (e.g., CVE-2023-39663, CVE-2022-42969, CVE-2019-11391), indicating disagreement between maintainers and reporters, which is also reflected in negative comments on GitHub issues. Table 7.8 presents ten representative examples of such comments from selected GitHub issues, along with URLs for further inspection. These discussions illustrate that developers often do not view ReDoS vulnerabilities as relevant or critical, which can lead to delayed responses or reluctance to address reported issues. While some developers prioritize security concerns, others downplay the severity of ReDoS, resulting in limited remediation efforts.

The blame for this perspective may lie with the cybersecurity academic community.

Table 7.8: Selected comments regarding ReDoS and corresponding issue URLs.

Comment	URL
D3 is almost never used to handle user input, so these ReDoS vulnerabilities are generally a huge waste of time.	https://github.com/d3/d3/issues/3939
indeed, but since you'd have to be attacking yourself to trigger the ReDOS in npm, it's not actually a vulnerability here.	https://github.com/npm/cli/issues/7902
None of these seem like they could be exploitable in real world scenarios. Please let us know if you disagree.	https://github.com/facebook/react-native/issues/46996
The few times there was an actual vulnerability, it was reported separately, and we released patches as soon as it was possible. You can always report real vulnerabilities here, but please do this if you understand the difference between a real vulnerability and a false positive. For example, a "Regex DDOS attack" can never be a real vulnerability for a development-time tool. If you're not sure, you're welcome to ask in this thread, but please keep it brief and to the point so that the thread doesn't become unreadable.	https://github.com/facebook/create-react-app/issues/11174
This kind of low-quality finding was assigned a CVE ID without any significant cross-checking.	https://github.com/pytest-dev/py/issues/287
So, a DoS can be caused by a package that's considered for download and installation (which can run arbitrary code), or by the site that serves such packages? I don't see how this can be exploited without the attacker being able to do much more damage; a fix will mainly silence automated checkers.	https://github.com/python/cpython/issues/102202
Most ReDOS vulnerabilities are self-attacks, meaning, not a vulnerability.	https://github.com/sarbbottam/eslint-find-rules/issues/349
Pretty much all complex regexes are vulnerable to ReDos. I've migrated some more important ones to use a token parsing approach instead of regexes.	https://github.com/nodemailer/mailparser/issues/378
When evaluating whether something is a vulnerability, you have to look at the attack vector and the respective cost of upgrading.	https://github.com/facebook/docusaurus/issues/10491
Anyone using any utils with inputs of arbitrary length runs a performance risk. Even built-ins aren't immune.	https://github.com/bestiejs/platform.js/issues/139

As noted in §7.3, researchers have performed large-scale measurements of ReDoS as well as produced new and more effective detectors for vulnerable regexes. Their measurement instruments have been integrated into popular security scanners (e.g., the integration of safe-regex into eslint). However, the resulting scanners analyze only the regex, without considering reachability constraints as well as other input constraints. When academics apply these tools, they are likely to filter false positives based on the regex's context. In contrast, practitioners sometimes make much ado about nothing. This suggests the need for empirical studies of academic tool uptake and more developer-friendly releases. More importantly, future work in the field should perform user studies with developers to better understand their challenges related to ReDoS—such as tool usability, security assumptions, and deployment configurations. In this way, we can better align the community's assumptions with those of practitioners.

(4) Engines have evolved significantly: It is encouraging that modern language runtimes changed their implementation in response to prior work on ReDoS (§7.4). These changes were often driven or informed by researchers (e.g., Ruby's adoption of Davis *et al.*'s defenses [124, 129]). We believe that this is a success story of effective collaboration between researchers and practitioners.

Our study shows that modern regex engines have evolved significantly, incorporating advanced techniques such as Thompson's algorithm, memoization, and resource capping to mitigate ReDoS. However, there are often redundant implementations that are triggered on demand by the engine, when matching a regex. Hence, future work should investigate the deployed fixes to find potential bugs or misbehaving corner cases. We advocate for comparative studies (such as ours in §7.4.2), where multiple regex engines are evaluated on the same regexes or datasets to systematically assess the strengths and weaknesses of their deployed defenses. The research community should also propose automatic solutions for migrating to updated engines. At the same time, research on ReDoS should take these advancements into account to ensure its findings remain applicable.

(5) Performance bugs vs. security vulnerabilities: Many of our findings may also extend to other algorithmic complexity vulnerabilities. While the threat of weaponizing slow computations is real, in practice, attackers still tend to favor brute-force, volume-based distributed DoS attacks. Researchers should aim to understand why that is the case and, simultaneously, continue to develop defenses against such potential future attacks. We also believe that there are many insufficiently explored, cross-disciplinary opportunities for ideas that aim to bridge the gap between performance, measurement, and security communities. For example, we argue that most performance bugs (e.g., redundant loop traversals [365]) can probably be lifted to a DoS attack when malicious intent is assumed. Thus, future work should further examine the relation between performance bugs and DoS attacks.

7.6 Data Availability

An artifact accompanying this chapter is available at `https://doi.org/10.5281/zenodo.15515222`. It includes: the CVE data analyzed in §7.2.2, the super-linear

regex corpus together with the reproduction scripts for the experiments described in §7.4.2, and the developer-discussion data and GitHub-crawler scripts.

7.7 Summary

In this work, we perform a multi-faceted study of the recent academic work on ReDoS and the new engineering realities in the mainstream programming languages, after initial fixes were deployed in most of them. We report on concrete examples of language runtimes that adopted defenses, which appear to be motivated by academic research. When studying the efficacy of these defenses, we find that they mitigate most ReDoS payloads, but they still leave significant room for attackers to maneuver. By surveying the academic work on ReDoS, we find that many papers in this domain use weak definitions and strong threat models, often considering any slowdown caused by regex matching as a security vulnerability. We advocate for future work to consider the recently-deployed fixes in mainstream programming languages and to evaluate ReDoS attacks under realistic deployment conditions (e.g., against a web application deployed in the cloud using reversed proxies and redundancy). We also propose restricting the attackers' capabilities to realistic payload sizes and modest bandwidths that comply with the asymmetric DoS setting. More importantly, future work should study the more general problem of weaponizing slow computation, which the attackers might leverage in the near future.

8

Studying Server-Side JavaScript Vulnerabilities

In the previous chapter, we examined Regular Expression Denial of Service (ReDoS) as a representative example of emerging, system-level vulnerabilities and highlighted the lack of consistent threat models and evaluation practices for such vulnerabilities. While systematizing existing knowledge clarifies how these vulnerabilities are conceptualized, it does not address a more fundamental limitation: the absence of realistic and reproducible evaluation artifacts for modern software systems.

This chapter addresses this limitation from an evaluation perspective, focusing on the challenge of evolving threat models and systemic risks (challenge C_3; see Introduction). As modern software systems increasingly rely on large dependency ecosystems and compositional architectures, assessing vulnerabilities requires executable artifacts that capture real-world behavior rather than synthetic code snippets or isolated examples. Without such benchmarks, the research community lacks reliable ground truth for comparing techniques, validating claims, or reasoning about exploitability under realistic conditions.

To address this gap, this chapter introduces SecBench.js, a benchmark suite of real-world vulnerabilities and executable exploits for server-side JavaScript in the npm ecosystem. The suite comprises 600 vulnerabilities spanning the five most common vulnerability classes in server-side JavaScript. Each entry includes an exploit payload and an oracle to validate successful exploitation, enabling systematic and reproducible evaluation. Beyond supporting the evaluation of vulnerability detection techniques, SecBench.js also facilitates the validation of public advisories, the identification of flawed fixes, and led to the discovery of 20 previously unknown vulnerabilities.

This chapter shares material with the corresponding publication [P6].

8.1 Motivation

JavaScript is one of the most popular programming languages, and its continued growth is supported by the npm ecosystem, a repository with more than two million reusable packages. Unfortunately, vulnerabilities are common in npm and pose a major threat to applications and the ecosystem as a whole. A large body of recent research is devoted to the security of the npm ecosystem in general, and server-side JavaScript, in particular. Many papers study specific classes of security problems, e.g., code injection [460, 184], ReDoS [459, 127], prototype pollution [294, 445], hidden property attacks [544], path traversal [198], supply chain attacks [146, 560, 477], outdated [560, 279, 378] or trivial [2] dependencies, and bugs in low-level code [78, 461]. There also is work on developing new defenses and detection techniques using static [184, 294, 295, 511, 477, 355, 354], dynamic [129, 124, 130, 268, 505, 506], or hybrid [460, 146] analyses.

Progress in a research community often gets fueled by a reusable benchmark. For example, such benchmarks serve to assess the relative merits of different contributions in databases [391], parallel processing [58], graphics [463], machine learning [97], and bug detection and repair [266].

Unfortunately, despite the importance of server-side JavaScript security, there currently is no comprehensive benchmark of vulnerabilities. Instead, researchers often rely on ad-hoc sets of programs collected manually by the authors of the respective papers. While benchmark sets are not without limitations, the lack of a security-oriented

benchmark suite for npm has important implications: (1) it slows down research, as authors of each paper have to look for a new set of benchmarks that satisfies their needs and may convince reviewers, and (2) it obfuscates the true merits of different techniques, as they are not compared across a single set of vulnerabilities. A reusable benchmark suite of npm vulnerabilities would alleviate these problems, and help improve experimental techniques and metrics in this area. Specifically, a successful benchmark should provide the following properties:

- *Realistic*: We aim at a benchmark suite built from a diverse set of real-world software, with as few modifications of the original code as possible. Such realism ensures that success on the benchmark is likely to generalize to other real-world security problems.
- *Executable*: We want the suite to include inputs that trigger the vulnerabilities and a runtime oracle that checks whether an attempt to exploit a vulnerability indeed triggers the effect anticipated by the attacker. Having such executable exploits serve as evidence that a vulnerability can be exploited, enables evaluating runtime detection and mitigation techniques, and allows for studying the runtime properties of vulnerabilities.
- *Two-sided*: We aim at a benchmark that includes both vulnerable and fixed versions of the code. Providing both sides is important for measuring the false positive rate of detection and mitigation techniques, but also for studying how vulnerabilities get fixed.
- *Vetted*: It is important to study each vulnerability in the benchmark suite to confirm its existence, ensure that it falls under a certain attack class, and generates appropriate metadata. This requirement is in contrast to large-scale, automatically gathered datasets, which are used, e.g., to train deep learning-based vulnerability detectors, and typically suffer from some noise.

To the best of our knowledge, there currently is no benchmark of vulnerabilities that matches all four of the above criteria. Table 8.1 summarizes the most closely related existing benchmarks and datasets. One group of them consists of synthetic vulnerabilities, either created manually [84], via template-based code generation [64], or automatically by mutating existing code [144]. These benchmarks lack realism and hence do not suit our needs. Another group consists of automatically gathered datasets, e.g., created based on commit messages that mention a vulnerability [160, 154]. These datasets do not come with executable exploits and lack manual vetting, which causes noise, e.g., in the form of unrelated code changes tangled with a commit [243]. Finally, the third group of benchmarks consists of manually curated vulnerabilities [236, 392], but lack executable exploits that use the vulnerabilities to trigger a security-relevant action. For example, MAGMA [236] provides inputs that show that vulnerabilities can cause a crash, but not that they can be further exploited by an attacker. Beyond the benchmarks and datasets listed in Table 8.1, there are collections of general bugs [266, 485], which are inspiring but do not focus on security-related problems.

The SecBench.js benchmark suite: This chapter presents SecBench.js, the first benchmark suite of JavaScript vulnerabilities that fulfills all four of the above

Table 8.1: Comparison with existing vulnerability benchmarks and datasets.

Benchmark/dataset	Language	Vulnerabilities	Realism	Executable exploits	Two-sided	Vetted
CGC [84]	C	590	✗	✓	✗	✓
Juliet [64]	C/C++, Java, C#	121,922	✗	✓	✓	✓
LAVA-M [144]	C	2,265	✗	✓	✓	✗
BigVul [154]	C/C++	3,745	✓	✗	✓	✗
Ferenc et al. [160]	JavaScript	1,496	✓	✗	✓	✗
VulinOSS [190]	various	17,738	✓	✗	✗	✗
Magma [236]	C	118	✓	✗	✓	✓
Ghera [331]	Java/Android	25	✓	✓	✗	✓
Ponta et al. [392]	Java	624	✓	✗	✓	✓
SecBench.js	JavaScript	600	✓	✓	✓	✓

requirements. The benchmark consists of 600 publicly reported vulnerabilities contained in widely-used advisory databases, such as Snyk and GitHub Advisories. To ensure realism, all vulnerabilities are included as-is, without any simplifications of the vulnerable packages. SecBench.js spans the five most common threat classes in the considered databases for benign, server-side JavaScript: path traversal, prototype pollution, command injection, denial-of-service, and code injection. Each vulnerability in SecBench.js includes an executable exploit, i.e., an attack input that triggers the vulnerability and causes behavior unexpected by the vulnerable software package. Additionally, we also provide test oracles to judge the success of each exploit. The benchmark is two-sided: if available, it includes a fix of the vulnerability where the provided exploit often fails. Finally, the benchmark is vetted, as we manually validate each vulnerability.

To showcase the usefulness of SecBench.js, we report several applications of SecBench.js that, without our suite, would be costly or error-prone to carry out:

- *Finding mislabeled versions*: By running our exploits on different versions of the vulnerable packages, we identify 168 versions in 19 packages that are incorrectly labeled as not vulnerable in the advisory database. In two cases, the mislabeling even affects the latest release of the package, meaning the packages suffer from zero-day vulnerabilities.
- *Finding flawed patches*: We apply four simple code transformations to the prototype pollution exploits in SecBench.js to test the deployed patches against variants of the original attack. This experiment identifies 18 flawed patches, i.e., new vulnerabilities, which at the time of writing have been assigned 12 CVEs.

- *Dynamically identifying sinks*: Using dynamic analysis, we automatically identify the code locations where the payload provided by an exploit reaches a sensitive API, called *sink*, for 94.5% of the vulnerabilities in our suite. The sink location can be used, e.g., to validate the reports of taint-style, static analyses.

In summary, this chapter contributes the following:

- The first benchmark suite of server-side JavaScript vulnerabilities. In contrast to existing benchmarks and datasets, SecBench.js is realistic, provides executable exploits, is two-sided, and has been manually vetted.
- Three concrete applications of the benchmark, which reveal previously unknown vulnerabilities and insights for future work.
- The entire benchmark is publicly available as open-source, well-documented, and packaged conveniently in a container image.[1]

8.2 Related work

Vulnerability datasets: There are different kinds of vulnerability datasets. One of them is benchmarks of vulnerable programs aimed to be used for evaluating static analyzers or fuzzers [64, 144, 236]. The most closely related such benchmark is Magma [236], which is also built from real-world vulnerabilities and comes with inputs to exploit them, but targets C instead of JavaScript. Moreover, their exploitation is limited to a crash, while our testing oracle asserts the success of security-relevant action. The AEG exploit generation system [30] aims at finding vulnerabilities and generating exploits automatically.

Another class of related work consists of large-scale datasets extracted in an automated manner, e.g., from version histories [190, 160, 154, 557], intended as training data for machine learning-based vulnerability detection. Due to their automated creation, these datasets do not come with exploits and suffer from some degree of noise (e.g., 53% true positives based on manual inspection [557]).

Finally, a third kind of dataset offers manually validated vulnerabilities and exploits for them [331, 392], similar to SecBench.js, but none of them targets JavaScript. Beyond vulnerabilities, other benchmark suites [58, 242, 63] are mainly designed to study a specific program area outside software security. To the best of our knowledge, we are the first to construct a benchmark of executable, real-world vulnerabilities in JavaScript code.

Bug benchmarks: Looking beyond the security domain, there are various benchmarks of general bugs, such as Defects4J [266], Bugs.jar [432], BugSwarm [485] for Java, BugsJS for JavaScript [224], and a set of JavaScript performance bugs [442]. Other benchmarks focus on concurrency bugs [304, 548], high-impact bugs [364], and non-functional bugs [405]. While many of these benchmarks also provide inputs to trigger the bugs, they do not focus on vulnerabilities.

[1] https://github.com/cristianstaicu/SecBench.js

npm and other package ecosystems: The prevalence of vulnerabilities in npm and other package ecosystems has motivated various studies and techniques to understand and identify ecosystem-level security issues. Zimmermann et al. [560] study ecosystem-level security threats in npm. Others study the impact of vulnerabilities on package dependency network [133], the phenomenon of "trivial" packages in npm [2], or the impact of ReDoS vulnerabilities [127]. Supply chain attacks are another problem of specific interest [146, 511]. Pashchenko et al. [378] report on an interview-based study to understand the (lack of) dependency management. All the above highlights the importance of security threats in large-scale package ecosystems. SecBench.js will help address these threats by providing a benchmark for evaluating future detection and mitigation tools.

JavaScript security: There are various techniques for detecting JavaScript vulnerabilities and for mitigating their exploitation, of which we discuss a representative sample. Many techniques detect a particular kind of vulnerability, such as injection vulnerabilities [460, 184], hidden property attacks [544], ReDoS vulnerabilities [459], sandbox breakout [12] and prototype polution [294, 445]. Others provide more general detection techniques, e.g., in the form of static extraction of taint specifications [462], dynamic taint analysis [268], and graph-based vulnerability detection [295]. There are mitigations against ReDoS [130, 124, 129], in the form of compartementalization [505], privilege reduction [506], and debloating of packages [279]. Beyond the JavaScript code itself, security-relevant bugs in the runtime implementation are another concern [78, 143]. We envision SecBench.js to help compare and improve techniques that detect JavaScript vulnerabilities and that mitigate their exploitation.

8.3 Methodology

In this section, we first discuss our threat model (§8.3.1) and then present our methodology for building the benchmark suite: collecting known vulnerabilities (§8.3.3, §8.3.4), adapting or newly developing proof-of-concept exploits (§8.3.5), and collecting additional metadata (§8.3.6).

8.3.1 Threat Model

We focus on vulnerabilities, i.e., security defects of a package that be exploited by an attacker. The reason for this focus is that vulnerabilities account for almost 60% of the problems reported in the considered advisory databases, making them a highly relevant target for a security benchmark. In contrast, we here do not consider malicious packages, i.e., packages that are intentionally developed and distributed with security risks. Most of the considered entries in our suite are libraries. By construction, these software components have incomplete threat models: Since it is not clear where the library's inputs come from, it is common to assume the worst-case scenario, i.e., that inputs are attacker-controlled. Our suite does not make any judgment on the feasibility of this threat model, but instead relies on the assessment of practitioners about the risk posed by such vulnerabilities.

Table 8.2: Number of unique vulnerabilities from the top ten classes available in the vulnerability databases.

Type of vulnerability	Nb. vulnerabilities
Malicious package	730
Cross-site scripting (XSS)	426
Path traversal	407
Prototype pollution	327
Regular expression denial of service (ReDoS)	264
Command injection	238
Code injection	215
Denial of service (DoS)	114
Use after free	60
Information exposure	28

We focus on packages that can be executed on the server-side, ignoring typical client-side issues, e.g., cross-site scripting and open redirects. Vulnerabilities in server-side JavaScript are particularly interesting because, unlike on the client side, the code does not run in a sandboxed environment. Instead, a vulnerable package, once exploited, can have serious effects on the broader environment, including reading environment variables, writing files, spawning new processes, and setting up network connections.

8.3.2 Types of Vulnerabilities

To be representative of the threats that practitioners are most interested in, we focus on the five most common classes of vulnerabilities in the advisory databases we consider. Table 8.2 shows the number of unique vulnerabilities for the top ten classes available in the vulnerability databases. Given our focus on benign, server-side JavaScript vulnerabilities, we ignore the first two classes and select the following five classes for inclusion into the benchmark. These classes are also targeted by recent work in this domain: command and code injection [506, 460, 184, 354, 506, 295], ReDoS [459, 127, 129, 124, 307, 34], prototype pollution [294, 295], and path traversal [295, 198].

The following briefly describes the five classes of vulnerabilities:

- **Prototype pollution:** This vulnerability enables an attacker to add or manipulate arbitrary properties to global object prototypes, which enables the attacker to tamper with the application logic. Attacks that exploit prototype pollution vulnerabilities often also make use of other vulnerabilities.

- **ReDos:** Regular expression denial of service (ReDoS) is a class of vulnerabilities that exploits poorly constructed regular expressions or faulty regular expression implementations. Doing so allows the attacker to make pattern matching a string against the regular expression take unusually long, and hence, lead to a denial of service.

- **Code injection:** This vulnerability allows an attacker to inject malicious code into an application, which will then be interpreted or executed by the application. This vulnerability typically stems from improper validation of input data.
- **Command injection:** Similar to the above, this vulnerability allows the attacker to execute arbitrary commands on the host operating system. Unlike code injection, it allows the attacker to leverage existing code to execute unexpected commands, usually within the context of a shell.
- **Path traversal:** Path traversal vulnerabilities (also known as directory traversal vulnerability) allow the attacker to access, create, write, or modify arbitrary files outside of an intended location in the host's file system. This vulnerability usually stems from insufficient validation or sanitization of user-supplied file names or paths.

8.3.3 Source of Vulnerabilities

We gather vulnerabilities from three different sources: Snyk, GitHub Advisories (previously called Npm Advisories), and Huntr.dev. These sources include thousands of vulnerabilities, often accompanied by an exploit, are popular among developers, and are extensively used in other work [506, 560, 133, 295]. For each considered source, we scrape its web interface to collect the number of vulnerabilities and their types, which provides the basis for further analysis.

8.3.4 Filtering of Candidate Vulnerabilities

We include in our suite only those vulnerabilities for which we can present an input that triggers a security-relevant action, as described below. Thus, we exclude vulnerabilities in packages (a) that we cannot install or that were deleted, (b) that we cannot reproduce, or (c) that are incompatible with our setup, e.g., operating system. When the public vulnerability report contains a proof-of-concept exploit, we adapt this exploit and integrate it into SecBench.js. Otherwise, we try to create an exploit ourselves, allocating a budget of one hour per vulnerability to this task.

As part of this filtering process, we manually inspect and validate each vulnerability, keeping only vulnerabilities that we successfully confirm to exist. This validation was performed mostly by three researchers over a period of eight months. Researcher 1 is an undergraduate student with solid JavaScript experience who curated the benchmark. Researcher 2, a second-year graduate student mainly working on analyzing the security of Node.js and JavaScript, manually tested and verified every PoC. Lastly, Researcher 3 is an expert researcher with extensive experience in the area, who reviewed the code of each commit and cross-checked data consistency.

8.3.5 Executable Exploits

Each exploit in SecBench.js contains five parts: (i) a setup phase in which the target package is imported and initialized, (ii) a sanity check that the security-relevant post-condition is not met before the actual payload, (iii) an API call that delivers the

problematic input to the vulnerable API, (iv) a check that the post-condition is met, and (v) a cleanup phase in which the side-effects of the exploit are reverted.

For example, the security-relevant action for injection vulnerabilities is to create a file on disk. As a sanity check, we verify that this file is not yet present on the disk at the beginning of the exploit. After the payload's execution, we assert the presence of the file. Thus, the payload needs to inject code that loads the file system API and creates the target file.

8.3.6 Patches of Vulnerabilities

Including the information about the patched version and the fixing commit in our suite is important for enabling studies of deployed fixes, as discussed in Section 8.5.2. Moreover, having fixed versions of the vulnerable code is essential for judging the precision of static vulnerability detection tools. Such tools should report a problem in the vulnerable version, but not in the fixed version.

We follow two strategies for extracting fixes for vulnerabilities. First, we scrape the considered sources to extract this information. If an advisory refers to a fix for the vulnerability, e.g., in the form of a pull request that fixes the vulnerability, then we include the fix into SecBench.js. Second, because some vulnerabilities are fixed, but the fix is not listed in the corresponding advisory, we search for a fix in case the first strategy is not successful. To this end, we run our exploit on the latest version of the package and check if it still works. In case the exploit fails, we manually analyze the failing exploit to understand if a fix was deployed. If a fix is present, we localize the code location of the fix and then analyze the repository's history to identify the fixing commit.

8.4 The SecBench.js Benchmark Suite

This section describes the details of SecBench.js, including what vulnerabilities it is composed of (§8.4.1), how the executable exploits ensure successful exploitations (§8.4.2), the implementation of the benchmark as a test suite (§8.4.3), and how the benchmark facilitates deploying program analyses that reason about vulnerabilities (§8.4.4).

8.4.1 Composition of the Benchmark

In total, SecBench.js consists of 600 vulnerabilities, each of which has an executable exploit. Table 8.3 shows how these vulnerabilities are distributed across the five classes of vulnerabilities, and the number of their metadata entries. For 48% of the vulnerabilities, we provide information about the fixed version and the fix commit. In particular, for nine entries, this information was not included in the corresponding security advisory, but we identified it using the second strategy described in Section 8.3.6. We provide CVE information for 67% of the exploits in our suite. The lack of metadata for some entries reflects the limitations of the underlying data, e.g., the fact that for some vulnerabilities, no CVE has been assigned or no patch has been deployed.

Table 8.3: Number of exploits included in SecBench.js, together with the number of metadata entries, for each considered vulnerability type.

Type of vulnerability	Nb. exploits	Fixed version	CVE information
Code injection	40	21	20
Command injection	101	41	90
Path traversal	169	19	80
Prototype pollution	192	126	158
ReDoS	98	78	59
Total	600	285	407

384 of the exploits in SecBench.js correspond to vulnerabilities reported both on Snyk and GitHub Security Advisories, while 203 and 7 were exclusively reported on Snyk and GitHub, respectively. For each vulnerability, our metadata links to the corresponding advisories in these databases. On average, each vulnerable package directly depends on 1.42 other packages, and is depended upon by 947.62 packages. This shows that the considered packages are relatively influential and that there are important interactions with third-party packages.

8.4.2 Ensuring Successful Exploitation

To ensure that each exploit in SecBench.js successfully uses the vulnerability to trigger an unforeseen, *security-relevant action*, each of the exploits comes with an *exploit oracle* to check this property. Similar to test oracles, these oracles judge the outcome of an executable exploit via assertions. Unlike regular test oracles, which typically check for desirable behavior, passing the exploit oracle means that the package is indeed vulnerable and that the exploit is successful at abusing the vulnerability.

For most entries in SecBench.js, we create exploits based on existing information, i.e., either a proof-of-concept (PoC) or a natural language description of a possible exploit given in the vulnerability database. A significant amount of work went into encoding this information in a uniform way, adapting it to our framework, and adding exploit oracles. As a measure to approximate this effort, we report the number of exploit assertions, which are all manually created by us: SecBench.js has a total of 1 244 assertions, thus, an average of 2.07 assertions per exploit. Additionally, for most exploits, we have a meta assertion on the number of oracle checks that must be performed during the execution. We found this to be useful for ensuring that all the asynchronous tasks are correctly executed.

An important observation is that exploit oracles and security-relevant actions are specific to a given class of vulnerabilities. For example, one can use a ReDoS vulnerability to trigger a slow computation in the main thread, or a prototype pollution to pollute the global scope. However, since these undesired actions are very different in nature, it is only natural that the oracles that assert their success are also different. The following discusses in detail each type of exploit oracle used by SecBench.js.

- *Code and command injections:* The oracle checks the existence of a custom-named

```
{
  "id": "CVE-2019-10744",
  "dependencies": {
    "lodash":"4.17.10"
  },
  "links": {
    "source1":"SNYK-JS-LODASH-450202",
    "source2":"GHSA-jf85-cpcp-j695"
  },
  "fixedVersion":"4.17.12",
  "fixCommit":"c3fd203b3be87a8177f7f00824033c95f981f984",
  "sinkLocation": "lodash.js:2573:21"
}
```

Figure 8.1: Metadata corresponding to the entry for lodash's vulnerability CVE-2019-10744. Links are abbreviated for brevity.

file on the disk. For most exploits, the security-relevant action uses the built-in `fs` module (code injections) or the `touch` UNIX utility (command injection) to create the file. At the beginning of an exploit, the oracle checks that the file is not present on the disk, and after the payload is executed, it asserts its existence.

- *Path traversal:* SecBench.js ships with a file called `flag.txt`, which the path traversal vulnerabilities aim to read. The oracle checks that the content served by the vulnerable npm package is the same as the one in this file. As the absolute path to this file varies from machine to machine, SecBench.js dynamically adjusts the payload to point to the indicated file.

- *Prototype pollution:* The oracle checks the existence of a variable called `polluted` in the global scope. At the beginning of an exploit, the oracle checks that such a value is not present, and once the payload has been executed, it asserts its existence. We note that this is a very strict oracle and that there are prototype pollution attack vectors that do not allow such an action to be performed, e.g., because they only allow for adding properties to specific built-in APIs, such as `Function.prototype`. Since the security impact of such pollutions is limited, we focus on a powerful oracle that checks for the most serious version of these attacks.

- *ReDoS:* The oracle checks that the target payload takes more than a configurable duration d to execute, with $d = 1$ second as the default. While this simple oracle may, in principle, lead to both false negatives and false positives, we minimize their likelihood by (i) dimensioning our exploits so that in our setup the computation takes significantly longer than d, and (ii) the duration d is long enough so that benign computations rarely trigger such long operations.

```
1  test("prototype pollution in lodash", () => {
2    // setup
3    const mergeF = require("lodash").defaultsDeep;
4    const payload = '{"constructor": {"prototype": {"polluted": "yes"}}}';
5    // sanity check
6    expect({}.polluted).toBe(undefined);
7    // exploit
8    mergeF({}, JSON.parse(payload));
9    // oracle check
10   expect({}.polluted).toBe("yes");
11   // cleanup
12   delete Object.prototype.polluted;
13 });
```

Figure 8.2: Exploit for `lodash`'s vulnerability CVE-2019-10744.

8.4.3 Implementation

At a high level, SecBench.js is a set of unit tests, where each successful exploit is considered a passing test. We use the Jest[2] testing framework, the most popular JavaScript testing framework[3]. SecBench.js relies on package managers, such as `npm` or `yarn`, to distribute the vulnerable package versions. Each vulnerability is contained in a separate folder containing a metadata file with information about the vulnerable version, the fix, the exact location, and external resources, as well as a JavaScript test file with the exploit and the oracle checks. The folder-based structure allows for multiple vulnerable versions of the same package. For example, for `lodash`, SecBench.js includes four different prototype pollution vulnerabilities and one ReDoS vulnerability.

In Figure 8.1, we present the metadata entry for the vulnerability CVE-2019-10744, an example entry in our suite. Clients of the suite can install the vulnerable version `4.17.10`, e.g., by running `npm install` in the folder containing this file. Figure 8.2 shows the test that implements the exploit. It first sets up the test by importing the vulnerable package and initializing relevant constants, and then asserts that the target property is not inadvertently present in the global scope. After that, the test triggers the payload by passing it to the vulnerable module. Finally, the test assesses the presence of the target property and hence, the exploit's success. Some tests in SecBench.js have more complex setup and teardown phases. For example, for path traversal vulnerabilities, we often need to start the target package in a separate process to act as a web server, perform an HTTP request, and finally, stop the server. Nonetheless, the high-level structure of all our tests is similar to the one discussed above.

To simplify batch installation, SecBench.js organizes its metadata and entries in a hierarchical structure. At first, we aggregate the individual vulnerability folders by vulnerability class, offering an aggregated `package.json` file that refers to all the individual vulnerabilities. For example, all ReDoS vulnerabilities are comprised in the `redos` folder, containing a `package.json` with 98 internal dependencies. Running

[2] `https://jestjs.io/`
[3] `https://2020.stateofjs.com/en-US/technologies/testing/`

the package manager in that folder will download all 98 vulnerabilities of that class, together with their dependencies. Similarly, SecBench.js also allows for batch installing all the 600 vulnerable packages, by providing a main `package.json` that refers to the individual ones corresponding to the five vulnerability classes considered in the benchmark. On a standard laptop, it takes about twelve minutes to run `npm install` in the main folder of SecBench.js, i.e., installing all the 600 vulnerable packages, with their exploits.

Once installed, a user can run all tests sequentially or in parallel. By default, `jest` runs all the tests in parallel, achieving the following test execution times on a standard laptop: 57s, 13s, 518s, 12s, and 156s, for code injection, command injection, path traversal, prototype pollution, and ReDoS, respectively. The two slowest classes are ReDoS, for which the payloads trigger a significant slowdown in the target packages, and path traversal, for which we run the tests sequentially because multiple of them try to serve requests on the same HTTP port.

8.4.4 Deploying Analysis Code

An important use case we envision for our suite is the development of runtime analyses to detect and defend against exploits. To this end, one needs to deploy analysis code that runs along with our exploits. We outline below two ways in which analysis code can be injected. First, due to the integration between Jest and Babel[4], an off-the-shelf transpiler for JavaScript, one can deploy runtime instrumentation using Babel. Second, a widely-used dynamic analysis technique for scripting languages is monkey patching, i.e., augmenting the semantics of built-in APIs to include analysis code. By leveraging Jest's setup phase, one can define custom analysis files that are executed before each test and can thus alter the test's global scope. We successfully use this analysis technique to extract the sink location for the vulnerabilities in our suite (§8.5.3).

8.5 Applications of SecBench.js

The availability of SecBench.js enables studying properties of vulnerabilities via dynamic analysis at a previously impossible scale. The following shows three examples of such studies. We first study how many versions of a given package are affected by our exploits, and whether this information matches the version constraints provided in the advisory (Section 8.5.1). Then, we show that applying simple mutations to our exploits helps identify zero-day vulnerabilities in insufficiently patched packages (Section 8.5.2). Finally, we present a dynamic analysis to precisely identify the location where an attacker-provided value gets passed to a sensitive API (Section 8.5.3).

8.5.1 Finding Mislabeled Vulnerable Versions

Vulnerability reports often indicate the versions of a package that are affected by a vulnerability, which is useful for guiding clients of the package to upgrade their dependencies. Since these reports are manually curated, they may be inaccurate, e.g.,

[4] `https://babeljs.io/`

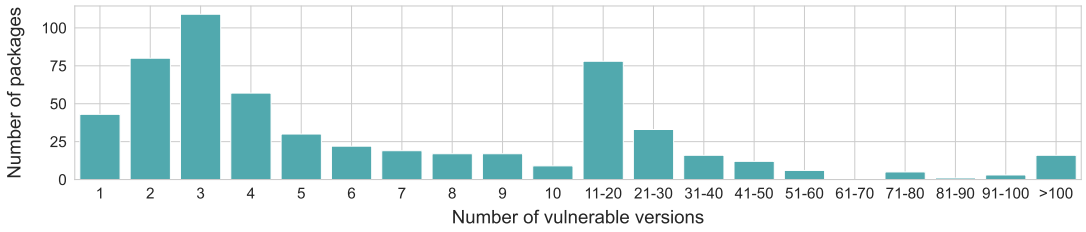


Figure 8.3: Distribution of the number of vulnerable versions of a package that are exploitable with SecBench.js.

due to regressions and flawed fixes. The exploits provided in SecBench.js allow for automatically checking which versions of a package can be successfully exploited. The following shows how cross-checking this information against the version ranges reported to be vulnerable helps to find mislabeled vulnerable versions.

Experimental Setup For every package in SecBench.js, we iteratively update it to every one of its versions and attempt to run the exploit against that particular version, storing the test outcome for each attempt. Because running an exploit on all versions of a package is non-trivial, this setup yields an underapproximation of versions that are affected by an exploit. For example, legacy versions of a package may not be compatible with our setup, or there may be breaking changes of the used APIs, along the history of the project. Finally, we check if all the vulnerable versions, as identified with SecBench.js, are flagged accordingly by the corresponding vulnerability database.

Results Most of the considered packages have less than 20 versions, while there are a handful of packages with thousand of versions. Figure 8.3 shows how many of these versions are found to be vulnerable with the exploits in SecBench.js. The mean number of vulnerable versions per package is 19.1, with 50% of the packages having ≤ 5 vulnerable versions, while `@firebase/util` has a maximum of 1 487 vulnerable versions. Overall, the figure shows that most vulnerabilities affect multiple versions, sometimes even more than 100.

For each package in SecBench.js, we check if there are inconsistencies between the vulnerable versions we detected and the original advisory. We manually check every mismatch and confirm 168 vulnerable versions in 19 packages that are incorrectly flagged as not vulnerable by Snyk. We are in the process of reporting these inconsistencies with associated examples to Snyk.

Examples In Figure 8.4, we show examples of mismatches between SecBench.js and the Snyk database. As mentioned earlier, our analysis might produce false negatives; thus, we only focus on cases where the analysis flagged a version as vulnerable, but the public vulnerability report claims the version to not suffer from the vulnerability. We discover three classes of mismatches. First, there are cases like `changeset` or `underscore.string`, where the constraint provided in the security advisory fails to capture legacy versions, released before the actual interval specified by the constraint.

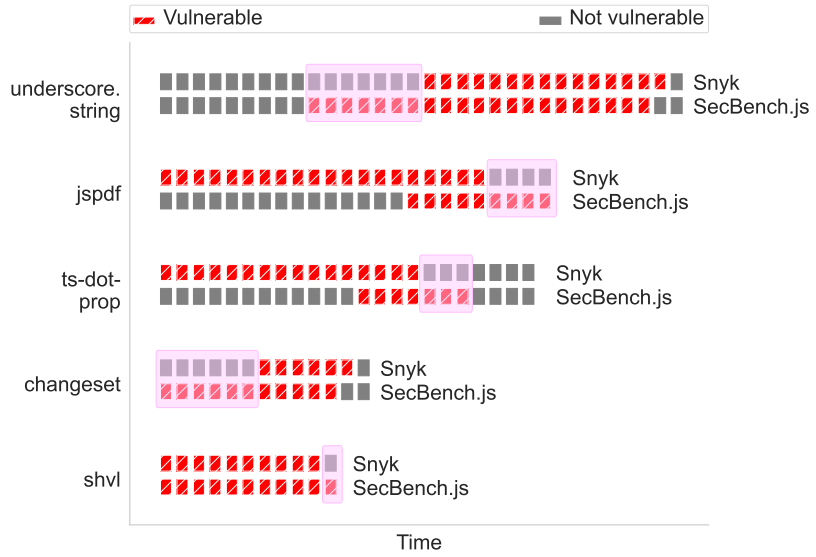


Figure 8.4: Examples of disagreements between our dynamic results and the Snyk database. Each box represents a version of the package. For each package, the lower line shows the assessment in SECBENCH.JS, while the upper line shows the available vulnerable version information in the Snyk database. The two lines are synchronized in the sense that two boxes corresponding to the same x-point depict the same package version. The pink overlay points to versions that are falsely labeled as non-vulnerable in the Snyk database.

Second, there are cases like `ts-dot-prop` when the constraint fails to capture versions after the indicated interval. Such errors might give users a false sense of security, since audit tools like `snyk-cli` would flag these versions as safe to use. Nonetheless, the affected releases might still be considered as legacy. Finally, for two packages, the latest version of the package is incorrectly flagged as not vulnerable, hence, representing zero-day vulnerabilities. Since developers are usually encouraged to migrate whenever possible to the latest version, having these versions incorrectly flagged as not vulnerable anymore is a severe problem and gives a false sense of security.

For example, the initial security advisory for `jspdf` identifies the following regular expression vulnerable to ReDoS, in version 2.3.0 and earlier:

```
/^data:(\w*\/\w*);*(charset=[\w=-]*)*;*$/
```

The regular expression consists of a wild card in the second capturing group `(charset=[/\w=-]*)*` which triggers the so-called "catastrophic backtracking". Since there was no exploit available for this vulnerability, we developed our own by studying the original vulnerable expression and found the following input to trigger the problem:

```
data:/charset=charset=charset=charset=charset=charset=charset=charset=
    charset=charset=charset=charset=charset=!
```

The advisory indicates that the problem was fixed in a later version and links to a commit that refines the regular expression into the following:

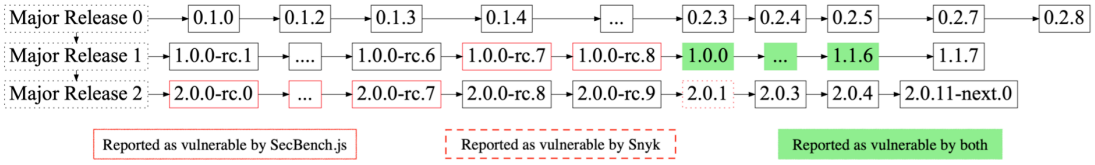


Figure 8.5: Example of a disagreement between our dynamic results and the Snyk database in the `mithril` package.

```
/^data:(\w*\/\w*);*(charset=(?!charset=)[\w=-]*)*;*$/
```

The look-ahead is supposedly preventing the catastrophic backtracking. However, because the patch uses a negative look-ahead to match the character literals *charset* =, it is still possible to perform a ReDoS attack, e.g., using the following input:

```
data:image/jpeg;charset=xcharset=xcharset=xcharset=xcharset=xcharset=↷
    xcharset=xcharset=xcharset=xcharset=xcharset=xcharset=↷
    xcharset=xcharset=xcharset=xcharset=xcharset=xcharset=↷
    xcharset=xcharset=xcharset=xcharset=xcharset=x!base64
```

As another interesting example, consider the case of the `mithril` package shown in Figure 8.5. This package has three major releases, all of which were concurrently maintained for some time. Due to this parallel maintenance, the natural ordering of versions does not match with the publishing order. By natural order, we mean that a version n.0.0 gets published before (n+1).0.0 or n.1.0, and the later version would be considered the latest version. But in the case of `mithril`, version 1.1.7 was published on September 23, 2019, i.e., after version 2.0.4, which was published on August 18, 2019, and marked as latest on npm.

These cases of parallel versioning make it difficult to correctly specify the constraint that indicates vulnerable versions. To specify the vulnerable versions for `mithril`, the Snyk database uses this complex constraint:

$$>= 1.0.0 \ < 1.1.7, >= 2.0.0 \ < 2.0.3$$

However, since versions like 1.1.7 are released later than 2.0.3, it is not trivial for developers to determine whether a given version they are using is vulnerable or not. We identify multiple release candidate versions, e.g., 2.0.0-rc.0 that are actually vulnerable, but that are not captured by the constrained above. As a result of such mislabeling, developers may get a false sense of security and accidentally use vulnerable versions.

Implications The findings in this section show that the ability to cross-check manually specified version ranges against the executable exploits in SecBench.js is useful to detect mislabeled versions. We also find that accurately specifying which versions are affected by a vulnerability is non-trivial, e.g., due to multiple major versions being maintained in parallel.

Table 8.4: Mutations for identifying problematic fixes, and the zero-day vulnerabilities identified by each mutation.

Mutation	Security advisories
`"__proto__"` → `["__proto__"]`	CVE-2021-23518, CVE-2021-23760, CVE-2021-23507 CVE-2021-23497, CVE-2021-23460, CVE-2021-23558 CVE-2022-25354, CVE-2022-25296, CVE-2022-25352
`"__proto__"` → `"constructor.prototype"`	CVE-2022-22143, CVE-2022-24279
`"__proto__": {...}` → `"constructor": {"prototype": {...}}`	CVE-2021-23470

8.5.2 Finding Flawed Fixes

When addressing a vulnerability, developers may sometimes overfit to a proof-of-concept provided in the original advisory. As a result, the published fix may not fully address the problem, leaving some attack vectors open to be exploited. For example, for a prototype pollution vulnerability, one can pollute the global object using both the paths `obj.__proto__` and `obj.constructor.prototype`. Hence, a fix that considers only one of these paths would be flawed.

Experimental Setup To detect flawed fixes based on SecBench.js, we design three simple mutations, shown in the first column of Table 8.4. The last two capture the case described earlier, while the first mutation corresponds to a type confusion problem[5]. We then update all the vulnerable packages in SecBench.js to their latest versions and only consider those packages for which the original exploit does not work, i.e., they are supposedly fixed. We then apply each of the three mutations above to the exploit in SecBench.js and rerun the modified exploit. If the test succeeds, we have identified a flawed fix for the corresponding package.

Results In total, we find thirteen, four, and one zero-day vulnerabilities for the three mutations, respectively. We reported all these issues to the maintainers and, we got assigned twelve new CVEs for our findings, as shown in Table 8.4.

Examples Let us consider the case of `convict`, a popular package from Mozilla for managing configuration files. In response to the original vulnerability report for this

[5] https://snyk.io/blog/remediate-javascript-type-confusion-bypassed-input-validation

package, the authors deployed a fix[6] in the `set` method, by including the `if` statement below:

```
1 const path = k.split('.')
2 const childKey = path.pop()
3 const pKey = path.join('.')
4 if (!(pKey == '__proto__' || pKey == 'constructor' || pKey == 'prototype↷
      ')) {
5  const parent = walk(this._instance, pKey, true)
6  parent[childKey] = v
7 }
```

The check prevents writing paths like `"__proto__.x"`, but not composed ones like `"constructor.prototype.x"`, where `pKey` is assigned `"constructor.prototype"`. In response to our report, the maintainers deployed a more sophisticated fix that also considers this additional attack vector.

Implications An alternative to the experiments described in this section would be to manually analyze all the deployed patches for the 192 prototype pollution vulnerabilities in our suite. While this is doable, the effort would be considerably higher than writing the simple mutations in Table 8.3 and running the suite three times. Moreover, we can run such experiments on a regular basis, for all future versions of the target packages, to detect possible regressions. Thus, we conclude that an executable vulnerability database, such as SecBench.js, is useful for validating deployed fixes and for identifying regressions.

8.5.3 Localizing Sink Calls

Most vulnerabilities can be described as a taint analysis-style flow of data from a source to a sink. A sink here is a built-in API that is used to deliver the payload. For example, a typical sink for command injection is `child_process.exec()`. Having precise information about the sink location is useful to better understand a vulnerability. Moreover, knowing the sink location provides a ground truth for evaluating taint-style analyses that report potential vulnerabilities. The following shows how to use SecBench.js to localize sink calls via dynamic analysis.

Experimental Setup To extract sink locations, we use a simple yet effective dynamic analysis. We run each exploit in a prepared environment in which relevant sink APIs are hooked. For example, for prototype pollution we add a custom setter on the property `polluted` in `Object.prototype`, i.e., the property that our payloads are trying to set. Each time an exploit sets that property, our analysis code is triggered, and we inspect the stack trace to extract the location that triggers the property set. Similarly, we hook `child_process` APIs for command injection, `fs` for path traversal, regular expression matching for ReDoS, and `eval` and `Function` for code injection. In case of multiple sink calls, we only extract the first sink location.

[6] https://github.com/mozilla/node-convict/commit/688c46afe099b44512665dee6263eacd9f4f71a8

Results Applying the dynamic analysis to all vulnerabilities in SecBench.js finds a sink location for 567 of 600 exploits (94.5%). The remaining cases are missed due to complex behavior calls not captured by our simple dynamic analysis. For example, for path traversal exploits, the main process creates a new detached process, where the actual sink calls happen. However, due to the newly created process, the dynamic analysis fails to trace the sink calls in a few cases.

Implications Having a large set of precisely identified sink locations provides a basis for evaluating existing and future taint-style analyses. Such analyses, both static and dynamic, typically warn users about unexpected flows from an API entry point to a sink location. The sink locations obtained using SecBench.js will serve as a ground truth for evaluating the results of taint-style analyses.

8.6 Discussion

This section discusses the application of SecBench.js in practice, its relationship with other language ecosystems, and some of its key limitations.

Applying SecBench.js: We see several opportunities for applying SecBench.js in future security research. First, SecBench.js is well-suited for evaluating mitigation techniques: a plethora of security systems [506, 505, 460, 129] aims to mitigate or eliminate unintended behaviors during the execution of a program. SecBench.js can be used to empirically characterize the success of these systems, including both mechanisms and policies safeguarding program execution. SecBench.js's ability to trigger vulnerabilities and confirm the anticipated side-effects via runtime oracles is critical here, including the decision to offer automation and minimize other side-effects that could interfere with mitigation techniques.

Second, SecBench.js can be used to evaluate both static and dynamic vulnerability detection techniques, i.e., whether a tool or a system designed to infer legitimate behavior or detect certain classes of attacks indeed succeeds in such inference or detection. The existence of both vulnerable and non-vulnerable versions of the code in SecBench.js is important for this goal, as they can be used to characterize precision and recall. Indeed, recent work already uses SecBench.js to validate a novel kind of taint analysis [96].

Finally, SecBench.js can also be used for in-depth studies and analyses of real-world vulnerabilities and the features that enable such vulnerabilities in practice, e.g., code and object complexity or source-target distance in the object graph.

Relation to other languages and ecosystems: The choice of a particular language and runtime environment—server-side JavaScript and Node.js—necessarily affects the classes of threats that are part of a benchmark suite. Server-side JavaScript means that a vulnerable component does not run in the sandboxed environment of a web browser. Once exploited, a vulnerable package can thus have serious effects on the broader environment in which the program is executing, including reading environment variables, writing files, spawning new processes, and setting up network connections.

Many of these threat classes are common in other languages and ecosystems, such as Python (PyPI), Ruby (Rubygems), and Java (Maven Central).

A few threats that are part of SecBench.js are not commonly found in other environments and are due to design decisions related to the semantics and implementation of JavaScript. For example, prototype pollution attacks are due to the combination of mutable intrinsics and runtime resolution available in the JavaScript language. As another example, ReDoS attacks are due to cooperative task scheduling present in the JavaScript runtime environment. These behaviors exist in other environments (mutable intrinsics in Smalltalk; cooperative scheduling in Lua) but have not received the attention they have received in JavaScript.

At the same time, some other classes of vulnerabilities that are possible or even common in other environments are not part of SecBench.js. One example is vulnerabilities stemming from the lack of memory safety, common in components developed in memory-unsafe languages, such as C and C++. Another example is cross-site scripting attacks, common in components targeting front-end web applications.

8.7 Threats to Validity

SecBench.js is a vetted benchmark suite, i.e., all vulnerabilities have been manually inspected to validate their existence and that their associated exploits and metadata are proper. Despite our best efforts, the results of this manual inspection may nevertheless be subject to occasional errors. The validity of any conclusions drawn based on SecBench.js is limited to the packages, vulnerabilities, exploits, and classes of threats included in the suite. In particular, this means that using the suite does not allow to generalize results to other programming languages or environments. Given the importance of server-side JavaScript, especially from a security point of view, we consider SecBench.js to nevertheless offer a valuable contribution.

SecBench.js is intended for research and thus provides infrastructure to automate, instrument, and validate the execution of the exploits included in SecBench.js. There is a small chance that this infrastructure might accidentally interfere with security policies or mechanisms associated with the artifact being evaluated. For example, SecBench.js's creation of new processes might interfere with systems that detect or mitigate process-related limits; and its execution in a dedicated container environment might interfere with policies related to limitation and prioritization of operating-system resources. Some of these limitations would be present in any evaluation infrastructure, i.e., even with ad-hoc benchmarks i.e., while others can be ameliorated through careful engineering of the artifact under evaluation.

8.8 Data Availability

The benchmark suite, including all associated code and data, is available as open-source at:

`https://github.com/cristianstaicu/SecBench.js`

8.9 Summary

Computer science research and development depend crucially on benchmarks that provide a common foundation for evaluating techniques, systems, and solutions. Security research for server-side JavaScript currently lacks a comprehensive set of real-world executable benchmarks, collected and analyzed systematically. As a result, researchers are forced to evaluate their contributions ad hoc, hampering the direct comparison among different techniques—a problem we have repeatedly faced ourselves and heard from others when developing systems targeting defensive software security. This work makes a first step towards addressing this problem by introducing SecBench.js, a benchmark suite of vulnerabilities and executable exploits for server-side JavaScript. The benchmark comes as a fully automated package that offers four desirable properties:

- *Realistic:* SecBench.js includes the original versions of 600 vulnerable npm packages, which were reported and confirmed by the community. These packages are used by thousands of real-world projects and are included without any modification to the benchmark.
- *Executable:* All 600 exploits presented in SecBench.js come with an executable proof of concept that exercises the vulnerability. All the proofs of concept validate that the exploit is successful with an appropriate oracle.
- *Two-sided:* 48% of the exploits in SecBench.js include not only the vulnerable but also a fixed version of the code, which can be used to study mitigation and bug-fixing techniques. For the remaining vulnerabilities, there was no fix available at the time of creating the benchmark.
- *Vetted:* The authors of this work manually analyzed and cross-checked each entry in the suite.

We perform several experiments to show the usefulness of the suite: identify flawed fixes and inaccuracies in the security advisories, extract vulnerability locations, and identify deployed patches. As a result of these experiments, we uncover 20 zero-day vulnerabilities, for which we were assigned 12 CVEs. We believe that these initial results show the potential of executable vulnerability databases like SecBench.js, and we hope that the community will join in the effort of extending and maintaining this benchmark suite.

9

Improving Call-Graph Analysis in JavaScript

In the preceding chapters, we addressed the systemic nature of emerging vulnerabilities (challenge C_3) and the need for rigorous, executable benchmarks such as SecBench.js to evaluate them. However, effective vulnerability detection fundamentally depends on understanding a program's control flow. In this chapter, we confront a primary technical barrier to this understanding: the structural opacity introduced by modern dynamic languages such as JavaScript.

This chapter addresses the challenge of rising complexity and abstraction (challenge C_4; see Introduction) in modern software systems, where dynamic languages such as JavaScript and layered frameworks introduce control-flow patterns that cannot be fully determined statically. Features such as higher-order functions, late binding, and runtime object manipulation cause call relationships to depend on execution context rather than source structure. As a result, static analysis alone cannot construct complete or precise call graphs, leaving critical gaps for downstream analyses.

To address this limitation, we introduce Graphia, a learning-based system for JavaScript that augments existing call graph construction by identifying missing edges. We formulate the task as *link prediction* over whole-program graphs, allowing Graphia to leverage graph neural networks (GNNs) to capture non-local dependencies that static analysis alone fails to resolve. We show that, even with limited training data, Graphia ranks true call edges among the top five candidates in 72% of cases, enabling downstream analyses to focus on a small set of high-confidence candidates.

This chapter shares material with the corresponding publication [P7].

9.1 Motivation

Call graphs depict the function invocations within programs, enabling the creation of interprocedural program analyses. The use of static call graphs in program analysis traces its roots back to 1970, when Barth et al. [43] and Graham et al. [210] pioneered this area. An ideal call graph should possess two properties: (1) soundness, indicating the ability to resolve all function call invocations, and (2) completeness, implying the avoidance of any false invocation edges. However, Rice's theorem [417] asserts that achieving both these properties is generally impractical. Consequently, existing analyses strive for reasonable trade-offs between soundness and precision while considering performance constraints. As an example, the state-of-the-art call graph construction tool WALA [145] has a recall value of only 62% [25] and a 70% ratio of false edges [501], when applied to JavaScript programs.

This is mainly because in languages like JavaScript, where functions are first-class citizens, constructing a call graph is a chicken-and-egg problem. To resolve certain call sites, one needs a sophisticated interprocedural analysis, which in turn requires a call graph. To tackle this predicament, researchers advocate for improving the foundational pointer analysis, integral to most call graph construction algorithms [71, 315, 475]. Since achieving a flawless pointer analysis is generally impractical, as per Rice's theorem, analyses aim for a better scalability-precision trade-off. For instance, implementing a context-sensitive analysis with WALA for pointer analysis only marginally reduces the false positive rate (by 8.6%) compared to a context-insensitive analysis [501], but significantly impacts the overall performance cost.

Recently, researchers have turned to machine learning to address challenges in static call graph construction. Utture et al. [501] propose cGPruner, which reduces false edges by 23% using random forest classifiers. AutoPruner by Thanh et al. [101] combines structural features from cGPruner with semantic features from CodeBERT and RoBERTa, using a neural classifier for enhanced call graph edge classification. While these approaches help reduce the number of false positives, we are not aware of any prior work that aims to reduce the number of false negatives, i.e., unresolved call sites. We note that industrial static analysis frameworks, such as WALA, already have a low false positive rate, since they are configured to produce precise but incomplete call graphs [24], instead of generating numerous spurious edges. In our experiments, we measure that this decision results in 60% unresolved call sites for CodeQL, a popular static analysis tool for JavaScript.

Therefore, in this work, we propose Graphia, an approach that uses machine learning to assist state-of-the-art JavaScript call graph construction in reducing the number of unresolved call sites. This is an inherently difficult problem due to the complexities of the language [86], e.g., higher-order functions, dynamic property accesses, and prototype inheritance. Thus, to support such advanced features, the analysis needs to consider the whole context around an unresolved call site. To this end, we propose representing the entire program under analysis using a graph neural network (GNN) and modeling the call graph augmentation problem as link prediction. Leveraging the comprehensive view provided by this holistic code representation, the GNN proves effective in capturing intricate relationships across the code base to handle advanced, non-local patterns like higher-order functions or dynamic property accesses. To the best of our knowledge, ours is the first work that proposes a learning process that acts on the entire program representation without involving an embedding step. This is analogous to the interprocedural paradigm in traditional static analysis, in which multiple functions are analyzed at once.

An alternative design would be to follow something more akin to intraprocedural analysis. That is, to use simpler machine learning algorithms combined with powerful code embeddings that incorporate rich structural information from the vicinity of the relevant program element, e.g., abstract syntax tree [17, 312, 551], or data flow graphs [14, 49, 299, 555, 558]. However, a common drawback of these approaches is their limited contextual scope. The majority of existing embedding techniques only consider the code, abstract syntax tree [91], or data flow information [558] within the function bodies of callers and callees [470]. This narrow focus essentially hides crucial contextual details from the downstream models. This limitation is particularly noticeable in dynamic languages like JavaScript, where coding practices such as anonymous functions and higher-order functions are prevalent. Handling such code patterns involves managing complex interprocedural relations, e.g., tracking the passing of function pointers. This underscores the necessity for a different formulation of the learning task to consider the context of the entire code base.

There are multiple challenges to be solved before applying graph neural networks to entire programs. First, how to represent the code to take advantage of the existing GNN architectures. Code representations like abstract syntax trees tend to be sparse, contradicting the dense graphs on which link prediction is traditionally applied, e.g.,

collaboration networks among scientists [552]. Thus, we propose a more dense program representation that utilizes both syntactic and semantic edges among the program elements. Concretely, we connect together nodes that refer to the same identifier. This enhancement of the AST forces parts of the code that handle the same concept closer together, facilitating the GNN's ability to trace likely data flow relations.

Another critical challenge is obtaining training data, considering the limitations of existing approaches and the lack of good data sets in this space. We propose extrapolating from the static edges produced by the call graph generation tool we aim to extend. Such tools are cheap to run, but as discussed above, they might miss a lot of valuable edges. GNNs were shown to be effective at link prediction in other scenarios where the ground truth is limited. For example, Zhang and Chen [552] remove 10% of the edges in the ground truth and use them as a test set. Thus, we hypothesize that using solely such static edges, Graphia can approximate the computation of these tools and generalize the link prediction to blind spots of the tools, e.g., caused by bugs. However, the very difficult unresolved call sites are fundamentally out of the distribution points for this training setup since the baseline tool cannot produce such links. For such cases, we propose extracting dynamic edges using instrumented executions of unit tests and including such edges in the training mix. We also use dynamically extracted edges as ground truth for the experiment, extracted either from the target project or from a different one.

We evaluate Graphia on the call graph of 50 popular npm libraries. For each statically unresolved call site, we judge the model's success in ranking all potential candidate edges from the call site to any other function definition in the project. Our evaluation reveals that in 42% of cases, Graphia places the correct edge on the first position in the ranking, and in 72% of instances, it predicts the correct edge within the top five ranks. To further understand the impact of our design choices, we conduct two ablation studies. The first study explores the effect of our suggested code representation, demonstrating a performance drop of 61% without its inclusion. The second study investigates the impact of our proposed node features, revealing a 13% performance drop when these features are omitted. Thus, the results show that Graphia is able to assist the existing call graph creation solutions with resolving difficult call sites. They also show that applying graph neural networks to entire programs is feasible, allowing the model to perform intraprocedural inferences.

In summary, our contributions are as follows:

- We present Graphia, an approach for improving the recall of existing JavaScript call graph creation methods. Concretely, we model the call graph enhancement problem as link prediction with graph neural network, leveraging both structural and semantic information from code, to handle unresolved call sites.
- To the best of our knowledge, we are the first to show that link prediction using graph neural networks is feasible on holistic, multi-file program representations.
- Through extensive experiments, we show that Graphia can learn from imperfect ground truths, i.e., either directly using the static edges produced by a baseline static analysis tool or a combination of such static edges and dynamic ones extracted from unit test executions.

```
1 var lexer = Object.create(this.lexer);
2 if (lexer.showPosition) {
3     errStr = 'Parse error on line ' + (yylineno+1)
4         + ":\n" + lexer.showPosition() + "\nExpecting "
5         + expected.join(', ') + ", got '"
6         + (this.terminals_[symbol] || symbol) + "'";
7 }
```

Figure 9.1: Caller function code snippet from `formula-parser` JavaScript library

```
1 var lexer = function (){
2   var lexer = ({
3     showPosition: function () {
4       var pre = this.pastInput();
5       var c = new Array(pre.length + 1).join("-");
6       return pre + this.upcomingInput() + "\n" + c + "^";
7     }
8   });
9   return lexer;
10 })();
11 parser.lexer = lexer;
```

Figure 9.2: Callee function code snippet from `formula-parser` JavaScript library

- We conduct an ablation study to gain deeper insights into our approach, affirming the effectiveness of our proposed code representation.

9.2 Motivating Example

In Figure 9.1 and 9.2, we present an illustrative example to highlight the rationale behind our approach and underscore the limitations of existing methods that exclusively consider the code in the vicinity of the call site. In line 1, an object of the `lexer` type is instantiated and assigned to the variable `lexer`. Advancing to line 4, we observe the invocation of the `showPosition` property of the lexer object. Notably, this property is not locally defined but implemented as a function elsewhere in the source code (Shown in Figure 9.2). Resolving the call site via property resolution requires resolving the type of `this.lexer`, which lies beyond the caller function's scope. Concretely, the anonymous callee function in line 3 of Figure 9.2 is attached to a named property of a local object literal. This object is then returned from an immediately invoked function and assigned to a `lexer` property of a parser object. In line 1 of Figure 9.1, such a parser object is referenced via `this` and instantiated via prototype inheritance. It then serves as the base object in the invocation of `showPosition`. This example highlights the challenge posed by JavaScript's dynamic features. To reason about such complex call sites, an analysis needs to trace property reads and writes, reason about complex types constructed via idiomatic object initialization, and understand prototype inheritance. All these abilities are far from trivial and lie beyond the reach of most

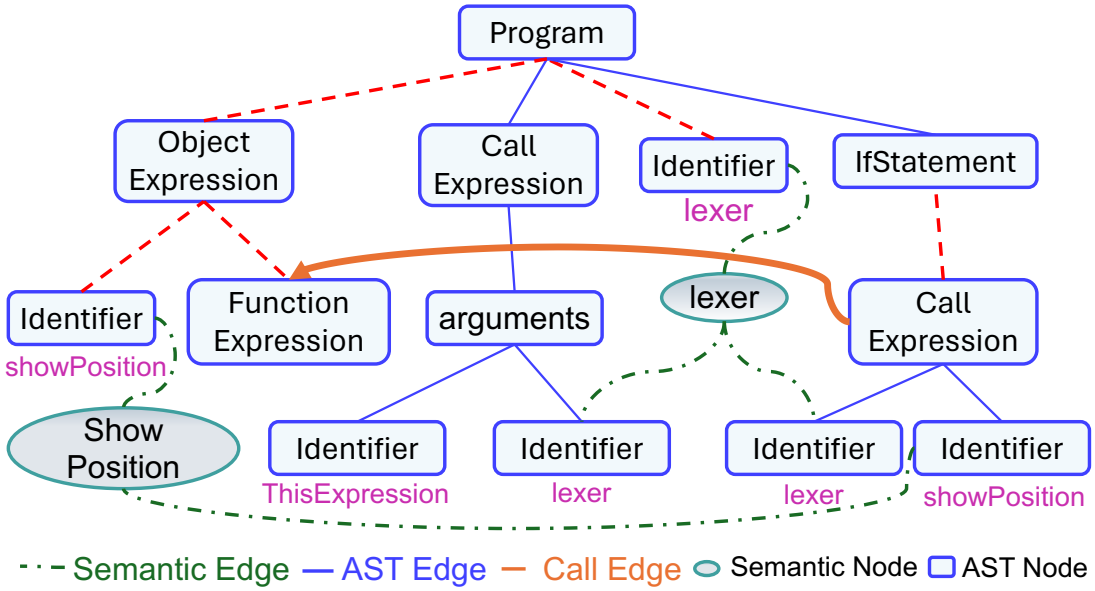


Figure 9.3: Combined code representation containing both syntactic and semantic nodes and edges. The graph corresponds to the code examples in Figure 9.2 and Figure 9.1, but it was simplified for conciseness.

static JavaScript analyses.

The initialization of the `lexer` object outside the function boundary on line 1 makes it difficult to infer the complex relationship between the caller and the invoked object's property. Models like AutoPruner [101], leveraging CodeBERT, may struggle to capture these nuances without considering the broader code context and runtime information. Recognizing that the `showPosition` property is essentially implemented as a function, requires a deeper understanding of the surrounding code and its dynamic behavior. Models lacking this contextual awareness may encounter difficulties inferring these intricacies, underscoring the significance of our proposed approach.

To illustrate how our proposed approach can overcome the limitations of existing methods, let's examine Figure 9.3. We represent the program's source code as a graph, incorporating different edge types to model both syntactic and semantic relationships between various nodes. The foundation of our program graph is the abstract syntax tree, produced using an off-the-shelf parser. Each syntax node is labeled with the associated node type produced by the parser. By employing structural representations and simulating data flow information through semantic edges, modern graph neural network models can adeptly infer long-distance relationships in interprocedural analysis. It is crucial to acknowledge that GNNs traditionally encounter challenges in leveraging long-range interactions, as their learned representations often tend to be locally focused. However, by incorporating semantic edges, we establish connections between related code elements, empowering the model to capture long-distance interprocedural relations effectively. This integration of semantic edges within the GNN framework fosters a

deeper understanding of code dependencies and augments the ability to analyze intricate relationships across different code segments.

In the provided example, our model can extrapolate that when the `showPosition` method of the `lexer` object is invoked, it effectively calls a member method of that same object. Despite the deferred definition, these components are interconnected through semantic edges linking to `showPosition` and `lexer` nodes. The `showPosition` semantic node in our code representation (highlighted in green) enables the GNN model's ability to discern this non-local relationship. GNNs leverage both the code structure and feature vectors associated with each node to extrapolate this intricate relationship. This illustrates how our proposed approach enhances the model's capacity for a more nuanced and comprehensive understanding of complex relationships within the code base.

9.3 Related Work

This section extends the discussion of prior work sketched in the introduction to discuss additional uses of machine learning in program analysis.

Applications of Machine Learning to Static Analysis: Many studies have employed machine learning to enhance static analysis, particularly in addressing false positives [501, 101, 490, 166, 330]. The common approach involves using a static analysis tool to collect error reports, manually labeling them, selecting a feature set for the data, and training a classifier to identify false positives. While the overall methodology is consistent, the choices in bug-reporting tools, datasets, and feature sets vary, demonstrating adaptability to different scenarios in static bug analysis. For example, Utter et al. [501] employed hand-picked semantic features and a Random Forest model to reduce false positives in call graphs for Java programs. Thanh et al. [101] utilized a transformer-based approach to capture code embeddings and semantic features for false positive reduction. Tripp et al. [490] introduced ALETHEIA, a user-centric approach using statistical learning to refine static security analysis output based on user feedback. Flynn et al. [166] proposed an alert triaging tool that combines multiple classifiers, historical audit data, and hand-picked features. In contrast, our work distinguishes itself by aiming to construct the call graph from scratch rather than pruning an existing one. Furthermore, our approach considers the entire code structure, a factor not addressed in previous studies.

Code Embedding Using Deep Learning: Deep learning techniques have been widely applied to generate code embeddings, which serve as foundational representations for various downstream tasks. These tasks span a range of applications, including type inference [13, 393, 239, 381, 523], code completion [293, 412, 473], code clone detection [433, 306], program repair [92, 318], fixing bugs [142], optimization [115], fault localization [352, 309], among other analyses on source code [79, 289, 353]. Researchers have explored the use of deep learning in code embedding for diverse purposes. Notable examples include the work of Allamanis et al. [13] and Pradel et al. [393], who leveraged deep learning for type inference in projects like Typilus and Typewriter. Hellendoorn et al. [239] and Peng et al. [381] also utilized deep learning for type inference and static

analysis. In the realm of code completion, Li et al. [293], Raychev et al. [412], and Svyatkovskiy et al. [473] employed deep learning techniques to enhance code suggestion and completion mechanisms. Code clone detection has also benefited from deep learning, with Saini et al. [433] introducing the Oreo model for effective clone detection. Program repair, a critical aspect of software maintenance, has seen advancements through deep learning. Chen et al. [92] and Mashhadi et al. [318] developed Sequencer and applied deep learning for program repair tasks. Fault localization, crucial for debugging, has been addressed by Nguyen et al. [352] and Lou et al. [309] using deep learning approaches. Analyzing source code from various perspectives has also been explored. Brown et al. [79], LeClair et al. [289], and Nguyen et al. [353] employed deep learning for language understanding, neural code search, and vulnerability curation, respectively.

Feng et al. [158] introduced CodeBERT, a pre-trained model to enhance code-text tasks, particularly in code search. Karampatsis et al. [267] incorporated contextual embeddings into a JavaScript corpus to improve program repair by utilizing context-specific information. Guo et al. [220] introduced Graphcodebert, which leverages data flow during pre-training to enhance understanding of semantic relations in code. Alon et al. developed code2vec [17] and code2seq [16], tailored for method name prediction within code snippets. These models highlight the significant impact of deep learning in source code analysis, improving tasks like code search, program repair, and method name prediction.

As previously highlighted, existing embedding approaches lack consideration for the entire code structure and often struggle to emulate interprocedural analysis. The limitations in existing methods underscore a gap in addressing the holistic understanding of code, particularly in scenarios involving complex structures and interprocedural dependencies. This gap emphasizes the need for more comprehensive approaches that can capture the intricate relationships and nuances within entire codebases, enabling more robust and accurate representations in the realm of source code analysis.

GNN for Program Analysis: Graph neural networks have gained significant prominence in handling various graph-structured data and have found widespread application in numerous general-purpose tasks. In the realm of code processing, researchers have leveraged GNNs to address various tasks, including code summarization [162], expression generation [76], code edition [547], and type inference [13]. Notably, Allamanis et al. [5] utilized GNNs with an augmented graph incorporating abstract syntax trees and data flow information to tackle downstream code processing tasks. However, similar to previous work, their approach focuses on function code snippets rather than capturing the entirety of the program, including multiple files.

9.4 Methodology

Figure 9.4 illustrates the overall structure of our proposed approach. We begin by parsing the code using an off-the-shelf parser to generate an abstract syntax tree. We employ a pruning algorithm to reduce the size of this tree, selectively removing non-essential nodes while preserving the tree's overall structure. Semantic nodes are then introduced for distinct identifiers and then connected via semantic edges to each of their usages inside the syntax tree. This enables the model to reason about identifiers' usage

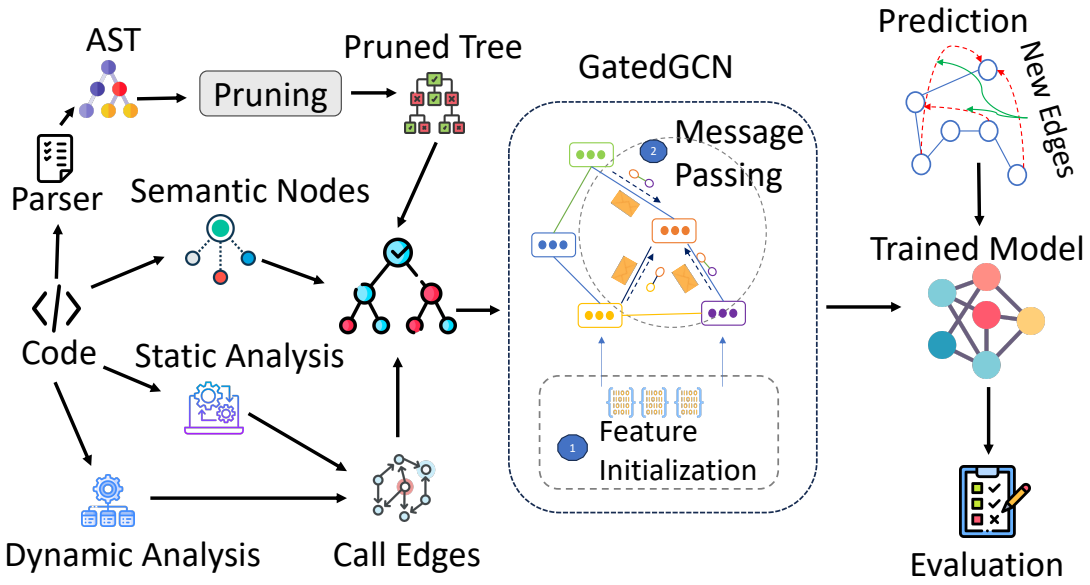


Figure 9.4: The overview of GRAPHIA framework.

and potentially track data flows. The pruned AST, enriched with semantic information, is now a graph, which is input to the graph neural network. Each node is initialized with a feature vector, and the model is trained using both static- and dynamically-derived edges. We use existing static call graph creation tools and instrumented execution of unit tests to collect the ground truth call edges, which we connect to the underlying code representation. Post-training, the model predicts new edges for statically unresolved call sites. For each such call site, GRAPHIA produces a list of ranked function definitions to which the call resolves. We evaluate the model's success by analyzing how high in the ranking it places hard-to-analyze call edges.

9.4.1 Generating Pruned AST

The first step in our methodology involves parsing the source code using a JavaScript parser. The parser transforms the raw code into an abstract syntax tree (AST), a hierarchical data structure that represents the syntactic structure of the program. This AST serves as the foundation for our subsequent analysis and modeling. In theory, this code representation suffices to enable a sophisticated machine learning model to perform arbitrary complex analysis tasks. In practice, however, this representation is too low-level, containing many details about the syntax of the programs instead of focusing on semantics. Recognizing this challenge, we implement a carefully designed pruning algorithm that selectively removes intermediary non-terminal nodes, such as those representing expression statements, binary expressions, and literals. The goal is to reduce the size of the tree while preserving its overall structure and connectivity.

The pruning algorithm operates on the AST's graph representation $G = (V, E)$, using a `parentChildDict` to map node relationships. It iterates through each

node v, evaluating it against predefined criteria for removal. Qualified nodes are disconnected from their parents, their children are reassigned to the parent node, and the `parentChildDict` is updated to maintain the connectivity of the tree's structure. For instance, when an expression statement contains multiple sub-binary expressions, all child binary expressions are removed, and their children are transferred to the parent expression statement. The process reduces AST size, preserving critical node relationships and leading to 60% faster model training.

9.4.2 Increasing Connectivity via Identifiers

Our second strategy is the introduction of semantic nodes into the pruned AST to enrich the graph's connectivity and reduce the average distance between nodes. These semantic nodes, representing distinct identifiers within the codebase, are connected to corresponding identifier usage nodes through semantic edges, depicted by green lines in our visualizations. These edges carry variable weights during training, allowing the model to discern and utilize semantic relationships more effectively. Integrating semantic nodes and edges into the pruned AST creates a comprehensive graph structure that encapsulates the source code's syntactic and semantic characteristics. We hypothesize that semantic nodes and edges allow the model to relate parts of the model that handle the same semantic elements, e.g., the same method name or global variable.

9.4.3 Generating Call Edges Using Static and Dynamic Analysis

Obtaining reliable ground truth is vital for effective model training, especially in the domain of program analysis, where both the precision and completeness of data are crucial. Traditional tools often struggle to meet these requirements simultaneously, leading to a common situation where tools sacrifice completeness for the sake of precision [24]. To accommodate this reality, we adopt popular static analysis tools to generate accurate analysis data. Nonetheless, while precise, these tools miss a lot of edges; thus, the training data represent an under-approximation of the actual call graph. This data, while not capturing the entire graph, reliably represents true call relationships within a codebase. To generate negative examples, we choose random pairs of call sites and function definitions for which there is no edge in the ground truth.

To complement the statically-extracted data, we also perform a dynamic analysis on those programs in the data set with good unit tests. For this, we utilized the test scripts that come with these programs, instrumenting them using source-to-source transformations to capture dynamic call edges, i.e., at each function entry point, we log the call stack to identify the invoking call site. A shortcoming of using existing unit tests is that they might not execute the entire program functionality, leading to still incomplete call graphs. Despite this limitation, the combination of static analysis and the dynamic insights from test executions allows us to study how the different types of call edges (static or dynamic) influence the performance of Graphia and simulate realistic scenarios in which the training data is incomplete.

Table 9.1: Feature set of GNN model.

Feature	*Description*
node_type	Type of the AST node
name	Associated identifier for the AST node
number_of_parameter	Number of parameters for callee
number_of_argument	Number of arguments for caller

9.4.4 Training and Evaluation

Inspired by the feature selection criteria outlined by Utture et al. [501], we initialize each node in our graph neural network model with a feature vector comprising four selected features that model the surrounding program context. We show these features in Table 9.1, and they capture the corresponding AST node type and name, together with the number of arguments or parameters for function invocation or definition nodes. These features are indispensable during the training and inference stages of the model, as discussed in Section 9.5.5. Our approach harmoniously combines syntactic simplification through pruning with semantic enrichment via semantic nodes and edges. This structured integration forms the foundation of our GNN input, enabling efficient information propagation and learning across nodes, thereby significantly enhancing the model's performance in program analysis.

Following the construction of the input graph and initializing nodes, the graph neural network undergoes a dedicated training phase. The model adeptly learns intricate relationships within the program graph by leveraging the approximate ground truth discussed earlier. Through iterative optimization of model parameters during training, the GNN enhances its accuracy prediction capacity. Post-training, the model is deployed to predict new edges within the program graph. Concretely, we train a graph neural network for each target program using a single dataset. We split this dataset into three subsets: 80% training, 10% validation, and 10% testing. We use five layers for our graph neural network. In our training, we set a maximum of 500 epochs, with a stipulation to cease training if the learning rate falls below 1×10^{-5}. Simultaneously, we maintain a batch size of 32,768 for each epoch, a figure strategically chosen to optimize both computational efficiency and training effectiveness. During each epoch, we select all edges in the validation set as true positives, and to ensure balance, we include an equivalent number of negative edges. These negative edges are composed of random pairings between call sites and function definitions that do not have an existing edge in the ground truth.

9.4.5 Evaluation Metrics

Assigning appropriate evaluation metrics is crucial for validating machine learning pipelines. While standard metrics like precision, recall, and F1-score are commonly used, they may be misleading for call graph generation due to dataset imbalance and failure to consider the entire candidate space. The ROC curve, another prevalent method, also has limitations in this context. Call graph datasets often have significantly more negative edges than positive ones, and traditional metrics don't account for the extensive

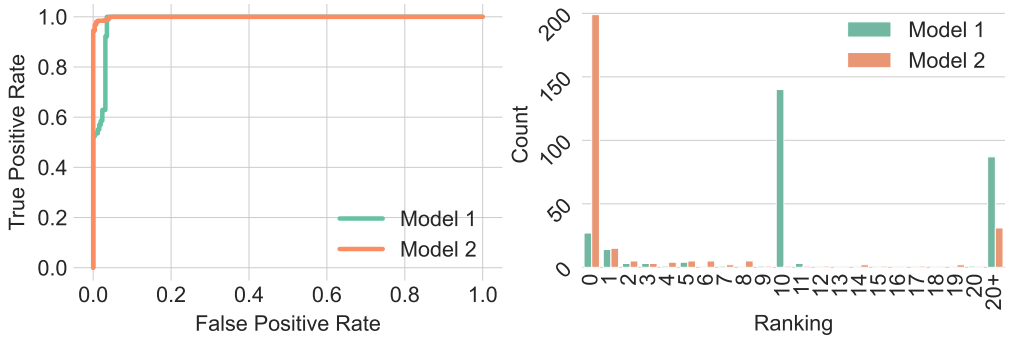


Figure 9.5: Performance of two models with similar ROC curves, but very different callees ranking ability.

candidate space in real-world scenarios. This reliance on conventional metrics may not accurately reflect the model's performance in practical applications.

To illustrate this further, let's examine the ROC curve alongside Precision, Recall, and F1-score metrics for two distinct models. In Figure 9.5, the ROC curve illustrates the comparative performance of two models. Despite the visual indication that Model 2 outperforms Model 1, a granular examination using precision, recall, and F1-score at a 95% true positive rate presents a paradox. The point corresponds to 0.89 threshold value for Model 1, and 0.58 for Model 2. This means that any edge with a probability score higher than these values will be recommended as an output call edge. Model 1 yields 96% precision, 98% recall, and 97% F1-score, while Model 2 boasts 100%, 94%, and 96%. However, these numbers do not reflect the underlying data distribution in our domain. Let us now consider the results in Figure 9.5, where we depict the performance of the model configuration above on the test set. For each call site, we ask the model to produce the highest 20 candidate predictions, sort this list, and observe which position is the edge in the test set. Despite Model 1 appearing as the optimal choice based on the F1-score, a substantial proportion of its predicted call edges rank 11th or lower among all candidates, with over 30% ranking at 20 or beyond. This prevalence of potentially false call edges raises significant concerns about the practical usability of Model 1 in real-world scenarios. It is worth noting what exactly causes the discrepancy between the two sets of metrics. The graph on the left in Figure 9.5 is produced using a balanced set of true positive-true negative edges, while the one on the right considers all the potential edges for each call site.

As a result, we choose to evaluate our model based on the ranking of predicted call edges, offering a more comprehensive and realistic measure of their utility in practical applications. This approach acknowledges the realities of our concrete problem domain, providing a nuanced understanding beyond the limitations of conventional metrics. Formally, we establish the definition of rank as follows. Let R_i represent the ranking of the $i-th$ call site, and P_i represent the predicted probability (or score) assigned by the model to the i-th call edge being a valid edge. The ranking R_i is calculated as the relative position of the $i-th$ call edge in the ordered list of predicted probabilities $P_1, P_2, ..., P_n$, where n is the total number of possible call edges, for the call site of interest. Mathematically, the ranking R_i can be expressed as:

$$R_i = \text{Rank}(P_i)$$

Here, $\text{Rank}(P_i)$ denotes the rank or position of the predicted probability P_i within the sorted list of all predicted probabilities. The smaller the rank, the higher the relative positive ranking of the call site. This definition captures the notion that the ranking of a predicted function definition is determined by how well its predicted probability compares to those of all other candidate functions, reflecting its relative position in the model's output.

9.4.6 Implementation

To implement Graphia, we utilized various open-source tools, each serving a distinct role in our workflow. First, we use CodeQL [187] for static analysis, building custom queries to accurately extract call edges from the source code. We verified the validity of the query by contacting the CodeQL team and asking if the extracted call graph reflects the one used internally by the framework. For dynamic analysis, we used Babel [32], a JavaScript compiler. We created a custom Babel pass along with a configuration file that enabled the automatic loading of this pass. This setup facilitated us to instrument the code and capture dynamic behaviors during runtime, which allowed us to extract call edges by logging the call stack at each function entry. We implemented our graph neural network using the Deep Graph Library (DGL) [518] for graph construction and manipulation and PyTorch [251] for model development and training. In particular, our neural network leverages the GatedGCN architecture implemented by Dwivedi et al. [148]. This implementation offers a versatile and ready-to-use framework suitable for various GNN architectures, providing a robust foundation for Graphia.

9.5 Empirical Evaluation

In this section, we first introduce our research questions and datasets, then present our empirical results and answer the research questions. Finally, we put our results into perspective with a discussion on how to use the proposed model.

9.5.1 Research Questions

Our evaluation aims to answer the following questions:

RQ1: *Does Graphia effectively predict call edges in JavaScript graphs?* This inquiry assesses Graphia's proficiency in ranking suitable callees on the basis of a static call graph generated using CodeQL. That is, we verify empirically if Graphia is powerful enough to approximate the computation performed by a state-of-the-art static call graph construction tool. The evaluation involves testing on a dataset comprising the 50 most popular large-scale npm libraries, focusing on Graphia's capacity to rank true edges relative to other potential candidate edges.

RQ2: *Can Graphia predict novel call edges?* This question delves into the system's capability to predict new edges that CodeQL, a static analysis tool, did not identify. Given the lack of automatic ground truth for these call sites, we resorted to dynamic

Table 9.2: Summary statistics of the selected npm packages, including node and edge counts and lines of code.

Metric	*Maximum*	*Minimum*	*Mean*	*Median*
Lines of Code (LOC)	2.5M	2K	216K	47K
Number of Nodes	705K	3K	84K	44K
Number of Edges	1.1M	5K	130K	67K
Call Edges	42K	271	2959	1034

analysis to obtain true edges that are beyond the reach of static analysis. This approach allows us to assess the ranking of these potential candidates and answer the question.

RQ3: *Which components of* Graphia *contribute to its performance?* This investigation examines the utility of Graphia's proposed code representation and features. Through an ablation study, we assess the contribution of individual components by selectively removing them and observing the resulting changes in Graphia's performance.

RQ4: *Can dynamic analysis enhance* Graphia*'s performance?* Considering Graphia's reliance on underapproximate ground truth, this question explores the potential performance boost achievable by integrating additional call edges from dynamic analysis. The assessment involves incorporating dynamic edges during training and observing the impact on Graphia's performance on test data.

RQ5: *Can* Graphia *handle intricate scenarios such as higher-order functions, cross-file invocations, and indirect* `apply()` *or* `call()` *invocations?* This inquiry evaluates Graphia's proficiency in addressing the complexities of the JavaScript language. The evaluation partitions the dataset into different subsets, and the rank of edges within each category is measured to assess Graphia's effectiveness in handling diverse and complex cases.

9.5.2 Dataset

We carefully curated a dataset of 50 npm packages from the top 1000 most depended-upon packages [358], focusing on entries that each exhibited a significant level of functional interconnectivity. Our selection criteria emphasized packages with at least 250 non-built-in call edges identified through static analysis. This threshold was specifically chosen to ensure the model has enough data to learn from. The resulting dataset is substantial, encompassing approximately four million nodes and six million edges, with a notable 116K of these edges being non-built-in call edges that are key to understanding inter-function call dynamics. The description of our dataset is presented in Table 9.2.

9.5.3 Effectiveness of Graphia

To address RQ1, we independently train Graphia on each package in our dataset and aim to predict statically-known edges. For each call site to be resolved, the target callee function may be any function defined in the current project. The findings, illustrated in Figure 9.6, underscore the efficacy of Graphia in accurately predicting known edges through the utilization of statically generated call edges. To conduct our experiment, we employed a random data split, reserving 10% for testing purposes. Notably, our

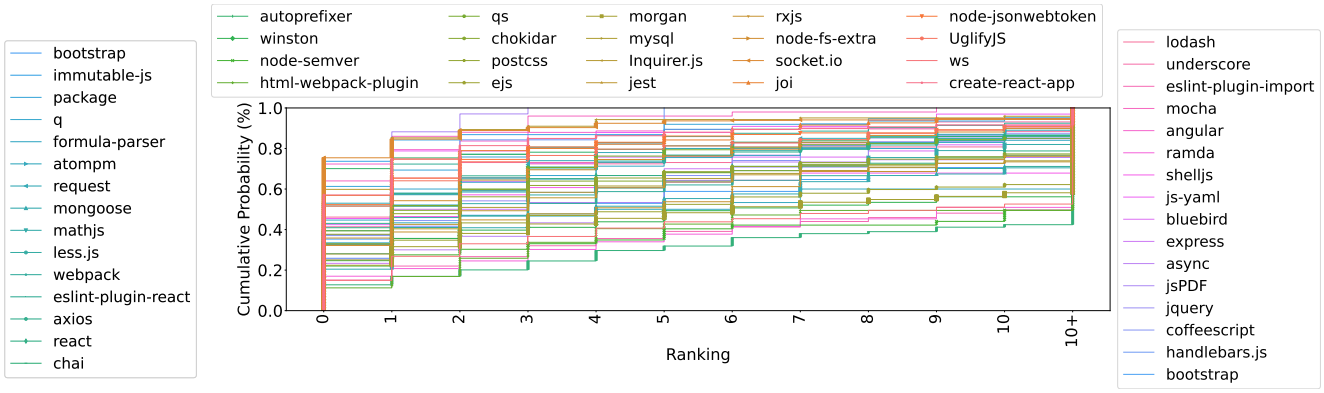


Figure 9.6: Distribution of Edge Prediction Ranking in GRAPHIA. This graph shows the performance of GRAPHIA on multiple npm libraries, in a static setup where both training and testing data come from the baseline static analysis.

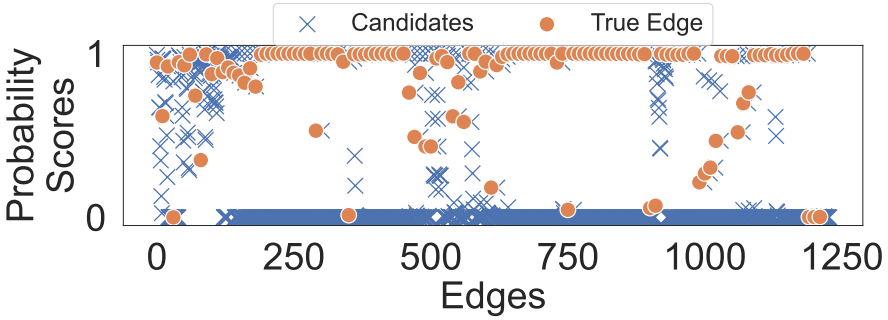


Figure 9.7: Confidence Score of Candidate Edges in *express* - Each point on the x-axis is an individual call site, for which we plot the highest ten predictions. The true edges are depicted by orange circles, and other candidates by blue crosses.

test set exclusively comprises call edges derived from the output of the static analysis tool. The results demonstrate that GRAPHIA can successfully approximate CodeQL's static analysis, successfully forecasting the true edge at rank 0 in 42% of cases across the evaluated libraries. Additionally, in 72% of instances, our model accurately predicts a new edge within the top five rank.

These statistics are derived using a weighted average approach, where the influence of each library on the overall result is proportional to its size. This method ensures that larger or more complex libraries, which typically present more challenging prediction scenarios, have a corresponding impact on the average performance metric, providing a more accurate and representative assessment of the model's capabilities. While there is a large variance across the entries in the dataset, the results generally show that GRAPHIA can handle a wide variety of real-world code and is able to resolve call sites under realistic scenarios, in which the callee function may be located anywhere in the project.

Figure 9.7 presents the confidence score of 10 highest ranked predictions, for each call site in the *express* library. These results clearly demonstrate that true edges, denoted by

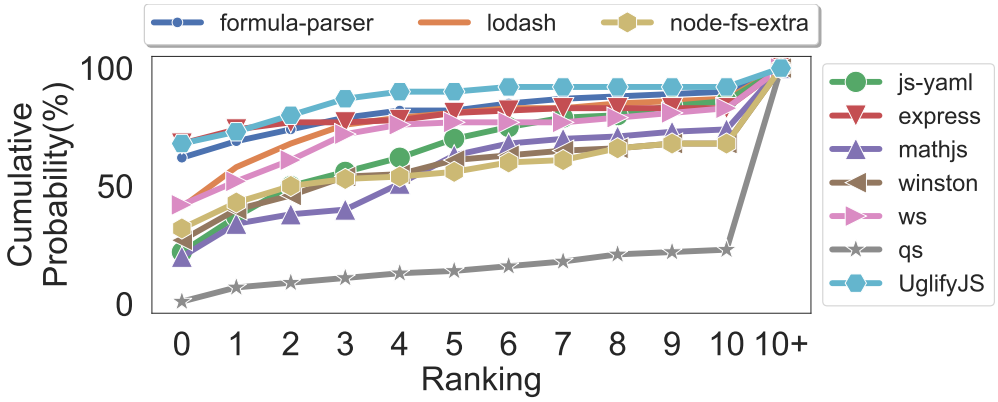


Figure 9.8: Distribution of GRAPHIA's Performance in Predicting Statically-Unknown Edges - We train the model with static ground truth, but evaluate it with dynamically-extracted ones.

orange circles, are predominantly positioned at the top of the graph, indicating higher confidence in the prediction. In contrast, the blue crosses representing other candidates are mainly clustered at the bottom, with signifying lower probabilities. This distinct separation in the graph highlights GRAPHIA's effectiveness in distinguishing between true and false edges across all call sites.

> GRAPHIA successfully approximates the static call graph construction by identifying statically-known edges at rank 0 in 42% of cases. Furthermore, it can predict call edges within the top 5 rank in 72% of instances. GRAPHIA also tends to associate a high confidence score for true edges in our dataset.

9.5.4 GRAPHIA's Ability to Generalize

To assess GRAPHIA's capability in predicting novel edges that elude static analysis, we conducted tests with unknown edges that the static analysis tool could not discover. To establish ground truth for performance evaluation, we employed a dynamic analysis approach using Babel-based instrumentation to extract dynamic function invocations. Utilizing available unit tests for each library, we focused on ten large npm libraries with substantial test coverage, recognizing that unit tests may not cover the entire code base, resulting in partial ground truth for dynamic call edges. We trained the model exclusively with static call edges, consistent with the setup in 9.5.3, and assessed the rank of each edge relative to other potential candidates. Figure 9.8 visually represents the efficacy of GRAPHIA in resolving call sites that CodeQL cannot handle.

In the case of the `express` and `UglifyJS` libraries, GRAPHIA successfully predicts true edges in 68% of cases at rank 0. However, in the most challenging scenario with the `qs` library, GRAPHIA struggles to predict new edges at rank 0. We suspect this is the case because most of the call sites that are not solved by CodeQL in this project are from a JavaScript bundle file containing thousands of lines of code and intricate ways of invoking functions via an emulated require method. The model probably has

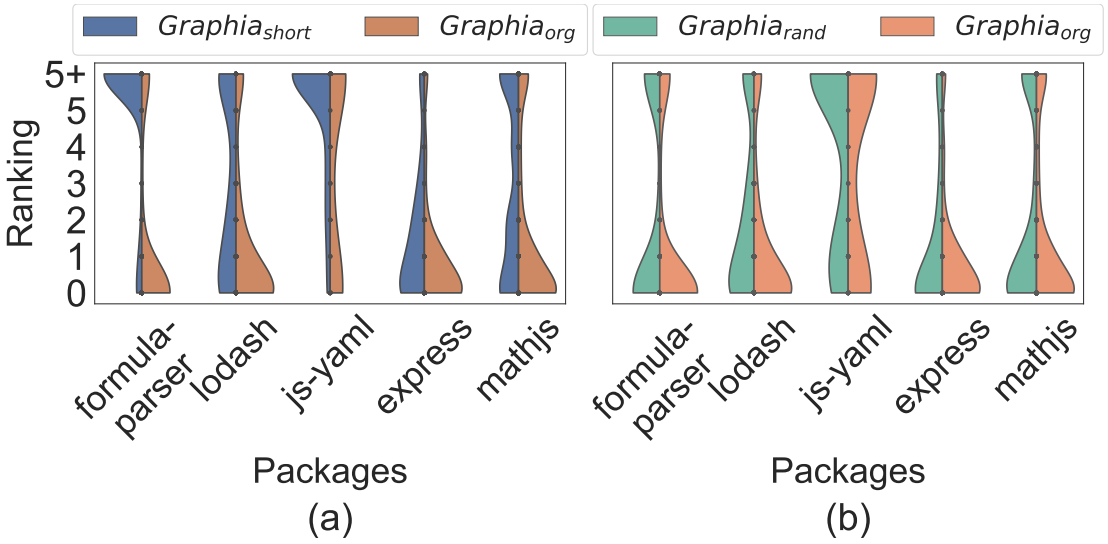


Figure 9.9: Performance Comparison: GRAPHIA with Call Nodes ($Graphia_{short}$) vs. Original Design ($Graphia_{org}$) (a), and Performance Analysis: GRAPHIA with Null Features ($Graphia_{rand}$) vs. Original Design ($Graphia_{org}$) (b).

few examples in the ground truth to aid it in handling such complex cases. Nonetheless, these results show that GRAPHIA can predict call edges that are beyond the reach of existing static analyses, even when trained exclusively with ground truth produced by such analyses.

> GRAPHIA successfully predicts novel edges missed by static analysis, achieving a 68% success rate for `express`. Furthermore, on average, it predicts almost 72% of edges within the top 5 rank, demonstrating GRAPHIA's ability to analyze diverse code.

9.5.5 Ablation Study

To address this question, we conducted two distinct ablation studies focusing on the significance of the code representation and the proposed features. In the first experiment, we assess the importance of our proposed code representation by removing the structure that binds certain code elements. Figure 9.9a presents the outcomes of these experiments, where $Graphia_{short}$ denotes GRAPHIA utilizing only $FunctionExpression$ and $CallExpression$ nodes along with their associated features, and $Graphia_{org}$ represents GRAPHIA in its original structure. As anticipated, the removal of the code structure has a substantial impact on the model's performance. The performance exhibited a 61% drop for `formula-parser` at rank 0. Similarly, the model's performance drops by 31%, 31%, 18%, and 23% for `lodash`, `js-yaml`, `express`, `mathjs`, respectively. In essence, if both syntactic and semantic structures are removed from GRAPHIA, there is a noticeable decline in its performance, particularly in handling complex long-distance relations. The observed performance reduction suggests that, in the absence of a struc-

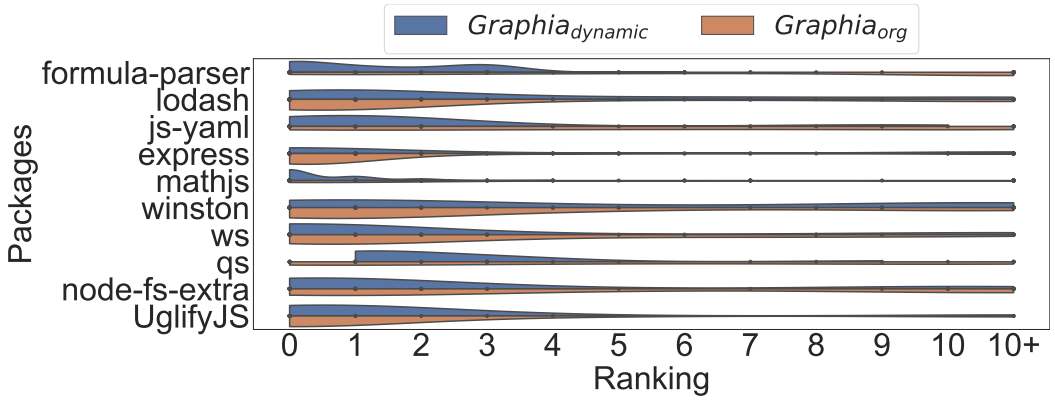


Figure 9.10: Impact of Integrating Dynamic Edges on GRAPHIA's Performance - The orange violin plots represent models trained solely with static ground truth, while the blue violins depict models additionally trained with dynamically-extracted edges.

tured representation, graph neural networks struggle to infer intricate interprocedural relations, given the absence of a clear path for message passing. This underscores the critical role of both syntactic and semantic nodes and edges in ensuring GRAPHIA's effectiveness.

Next, we investigate the importance of the node features outlined in Table 9.1. The results of this analysis are depicted in Figure 9.9b, where $Graphia_{rand}$ and $Graphia_{org}$ represent GRAPHIA with null node features and the original structure, respectively. While the impact is less pronounced than the code structure ablation, the absence of node features still leads to notable differences, particularly at slightly higher ranks. Specifically, the performance experiences drop of 13%, 10%, and 8% for *formula-parser, js-yaml, express* respectively, when considering higher ranks. This underscores the importance of the proposed features, embedded in the nodes, suggesting that these features play a crucial role in GRAPHIA's effectiveness.

GRAPHIA's effectiveness significantly diminishes without syntactic and semantic structure. The model's output is further impacted when node features are absent, particularly at higher ranks, emphasizing their importance in the overall model's predictive capability.

9.5.6 Boosting Performance with Dynamic Edges

In addressing the previous questions, our model was trained solely on static analysis call edges, yielding an under-approximated call graph. Here, we explore enhancing GRAPHIA's performance by integrating dynamic and static analysis. By merging dynamic and static call edges as per the approach detailed in 9.5.4, we aim to harness dynamic analysis's capability to capture intricate patterns beyond static analysis. Although attaining full coverage in dynamic analysis poses challenges, we hypothesize that our model can generalize to unexplored code segments. In Figure 9.10, we compare statically trained models with hybrid ones incorporating dynamic edges for each package. Notably, for `formula-parser`, `js-yaml`, `express`, and `mathjs`, adding dynamic edges

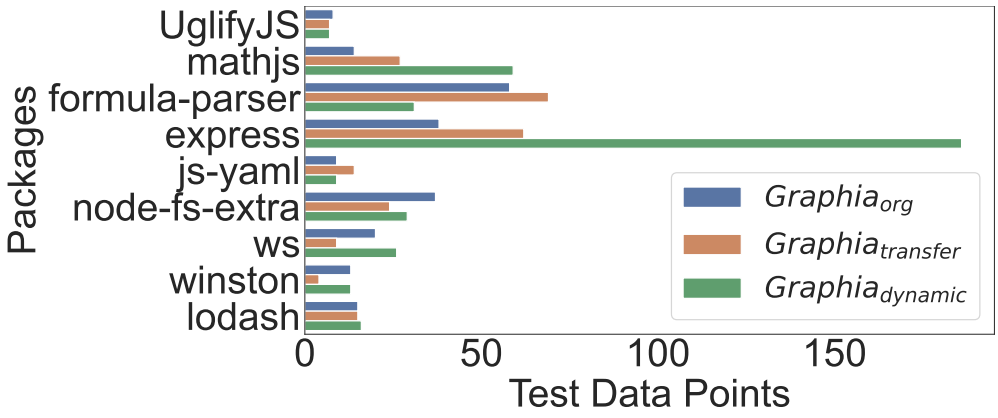


Figure 9.11: Impact of Transfer Learning on GRAPHIA's Performance. The blue bars represent models trained solely with static ground truth, while the green bars depict models trained with dynamically-extracted edges. Additionally, the orange bars signify models trained with data from four other projects.

improved rank 0 predictions by 106%, 50%, 17%, and 31%, respectively. Conversely, `lodash` saw a nearly 40% reduction at rank 0. However, dynamic edges had a more pronounced impact at higher ranks, with up to a 42% improvement for `qs` at rank 5 and at least an 18% increase for most libraries at the same rank. The variability in GRAPHIA's rank 0 performance stems from limitations in dynamic analysis. Although it offers insights into runtime behavior, it often generates a limited number of complex dynamic call edges due to coverage challenges. This data scarcity can lead to overfitting during model training, hampering generalization. Nevertheless, even partial dynamic analysis significantly complements incomplete static data, enhancing the overall model's performance.

To emphasize the impact of integrating dynamic edges, we conducted transfer learning experiments using a five-fold cross-validation approach. We trained models on combined static and dynamic edge graphs from four projects, relying only on static edges from the fifth. Figure 9.11 shows that adding dynamic edges significantly boosts `mathjs`, `formula-parser`, `express`, and `js-yaml` performance at rank 0 by 92%, 18%, 63%, and 55% respectively. This shows the potential for performing dynamic analysis for only a handful of tested projects and reusing this knowledge to augment call graph construction for other projects.

GRAPHIA's integration of dynamic call edges showed mostly positive effects. For example, it notably enhanced rank 0 predictions for half of the considered programs by up to 106%, with improvements reaching as high as 54% within rank 5.

9.5.7 Dataset Partitioning: Predicting Hard Edges

Treating all call edges uniformly does not give any insight into why some edges are not resolved by the static analysis in the first place, and into understanding the failure modes of the model. As noted earlier, there are many difficult-to-handle JavaScript language

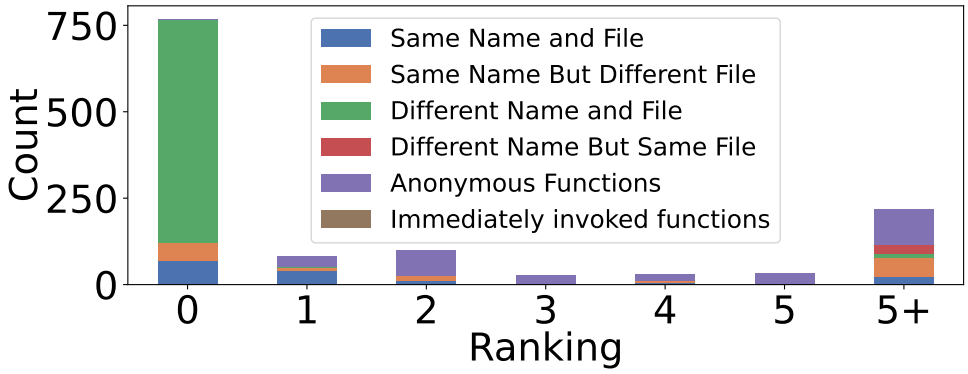


Figure 9.12: Efficacy of GRAPHIA in Predicting Call Edges Across Various Edge Types in JavaScript - Highlighting a 98% success rate at rank 0 for functions with different names across files and a 63% success rate in identifying anonymous function calls within rank 5. This figure depicts call sites from the `express` package.

constructs that lead to a failure in resolving call sites. In this research question, we aim to partition the dataset based on the call site type and study the model's performance on each partition. We consider six distinct categories, facilitating a comprehensive evaluation of model performance. The assessment, as depicted in Figure 9.12, focuses on the specific case of *express* package, revealing GRAPHIA's ability to infer relations even across multiple files successfully.

Notably, in test scenarios involving functions with distinct names located in separate files, GRAPHIA impressively predicts 98% of new edges at rank 0. This is vital in cases where functions are defined as function expressions assigned to variables and later invoked through aliases, a common practice in JavaScript. Such non-local, distant relationships are challenging for static analysis tools to detect. Additionally, GRAPHIA shows remarkable performance with anonymous functions, correctly predicting 63% of new edges within rank 5. This indicates its strength in handling anonymous function calls, where static analysis tools often struggle. These results highlight GRAPHIA's versatility and effectiveness in dealing with the complexities of JavaScript code, particularly in scenarios involving higher-order or anonymous functions that pose significant challenges for traditional static analysis methods. The model's successful handling of various difficult edge types vouches for its applicability to real-world scenarios, where JavaScript code bases are typically rich in diverse functional constructs.

The considered case study shows that GRAPHIA can handle hard features of the JavaScript language. For cases where the call site and the callee are in different files, GRAPHIA places the correct function definition at rank 0 in 98% of the cases. For calls to anonymous functions, GRAPHIA places the correct function definition in the top 5 ranks in 63% of the cases.

9.6 Threats to Validity

The primary threat lies in using CodeQL to establish ground truth for training Graph Neural Network (GNN) models. Although effective, CodeQL can introduce false negatives into the derived ground truth, and, more rarely false positives. This limitation may lead to a likely underestimation of the actual prevalence of call edges, potentially compromising the effectiveness of GNN models and biasing the training. Consequently, their ability to generalize accurately to real-world scenarios may be hindered. While achieving a sound and complete static analysis is a challenging task, leveraging multiple static analysis tools alongside CodeQL could produce more reliable training data understanding of the data, thereby mitigating the impact of false negatives and false positives.

The second threat to validity concerns the generalizability of the results to programs beyond our dataset. We attempted to mitigate this threat by considering a large set of popular, real-world libraries of various sizes.

9.7 Data Availability

We provide an extensive replication package containing all our scripts (for training the model, parsing the code, and running the analyses) and data (serialized graph representation for the considered packages and the set of all 163k call edges):

https://github.com/kal-purush/ml4call-graph

9.8 Summary

In this chapter, we show that link prediction with graph neural networks can be applied to entire JavaScript programs. We propose a code representation that increases the graph connectivity by connecting identifiers with the same name. We show that Graphia, our link prediction approach, can learn both from a purely static ground truth produced by existing static analysis tools and from dynamic edges generated by running the unit tests. The results show that Graphia can assist existing call graph approaches in generating hard edges involving dynamic property accesses or higher-order functions. Considering the promising results, we plan to experiment with alternative code representations, e.g., by integrating control flow information and with different downstream analyses.

10
Concluding Remarks

10.1 Conclusion: The Socio-Technical Reality of Modern Software Engineering

This dissertation studied modern software engineering as a socio-technical system in which technical design choices interact with economic, linguistic, and human contexts. It argues that modern software engineering cannot be understood through the lens of code and algorithms alone. The empirical evidence across the core chapters supports this thesis, revealing a digital ecosystem fractured by economic disparity, challenged by linguistic diversity, and exposed to context-dependent vulnerabilities. By combining large-scale web measurements, novel dataset construction, and advanced program analysis techniques, this work identifies these structural gaps and provides practical tools to address them.

10.1.1 Economic Context and Digital Disparity in the Web Ecosystem

The first major contribution of this dissertation, detailed in Chapter 3, examines prevailing assumptions about the global web ecosystem. Through a large-scale empirical study of 200,000 websites across twenty countries balanced between developed economies (e.g., USA, Germany, Japan) and developing economies (e.g., India, Brazil, Nigeria) we quantify the digital divide across five key dimensions of web development.

The findings offer a nuanced view of the "leapfrogging" hypothesis and of common assumptions about web performance in developing regions. Rather than clearly confirming or rejecting the idea that developing nations bypass legacy technologies to adopt modern standards, the data reveals a more complex reality characterized by structural adaptation alongside persistent optimization failures. We observe that websites in developing regions are, on average, 10% smaller in total byte size and exhibit significantly lower Document Object Model (DOM) complexity than their counterparts. This relative "lightness" appears to be a beneficial adaptation to resource constraints, resulting in shorter browser blocking times, challenging the assumption that poorer optimization necessarily implies uniformly worse performance. However, this structural simplicity is accompanied by weaker optimization practices. Our measurements indicate that websites in developing regions waste around 10% more bandwidth on improperly sized images and redundant JavaScript and CSS code compared to developed regions. This suggests that the smaller footprint stems primarily from simpler content structures rather than deliberate performance-oriented engineering.

The security and privacy landscape further reflects this economic divide, though not always in the expected direction. While HTTPS adoption remains lower in developing regions (88%) compared to near-universal adoption in developed countries (95%), we observe notable differences in software supply chain security practices. In several developed countries, a non-trivial fraction of websites include third-party libraries affected by high or critical known vulnerabilities, with observed rates exceeding those measured in any of the developing countries in our dataset. This suggests that increased reliance on complex third-party ecosystems in developed regions may introduce additional supply chain risks. In addition, despite stronger regulatory frameworks, websites in developed countries tend to deploy a higher number of third-party trackers, indicating

that stricter privacy regulations do not necessarily translate into reduced tracking in practice. Whether these disparities arise from differences in enforcement, business models, developer practices, or deliberate design trade-offs remains an open question for future socio-technical inquiry.

10.1.2 The Accessibility Crisis in the Global South

Expanding on the theme of disparity, Chapter 4 presented the first large-scale cross-national study of mobile web accessibility in the Global South, evaluating 100,000 websites across ten countries. This study shifted the focus from general usability to the specific exclusion of users with disabilities. The results are alarming. We found that only 40% of the evaluated websites meet critical accessibility guidelines for blind and low-vision users. The most common violations, such as missing alternative text (alt-text) for images, poor color contrast, and inadequate touch target sizes, are not merely technical oversights; they are barriers to fundamental rights. In a mobile-first world, where the smartphone is often the sole gateway to government services, banking, and education, an inaccessible mobile website is effectively a locked door.

Crucially, our research indicates that technical compliance cannot be separated from the broader sociopolitical context of governance and social stigma. We identified a strong correlation between regulatory frameworks and accessibility scores: countries with robust, enforced legislation (such as India and Brazil) demonstrated significantly higher adherence to WCAG compared to countries with weaker frameworks. However, governance does not exist in a vacuum; it is often undermined by deep-rooted societal stigma. In countries with higher violation rates, such as Bangladesh, Pakistan, and Vietnam, individuals with disabilities are frequently marginalized, leading to a systemic disregard for their digital needs. This stigma manifests in severe underreporting of disability statistics. For instance, Pakistan's national census reports a disability rate of only 0.48% (3.3 million), a figure starkly contradicted by British Council estimates of 27 million. This data gap distorts public policy and reduces the political will to invest in accessibility initiatives. Therefore, tackling the accessibility crisis requires a unified approach: technical tools and strong governance must work in tandem to dismantle the social stigma that fuels digital exclusion.

10.1.3 Linguistic Diversity as a Structural Challenge

The second thread of this dissertation examines the linguistic dimensions of software engineering. The web and its underlying technologies originated in an Anglocentric context, and this legacy continues to shape the modern digital experience. While multilingual web content has expanded significantly, accessibility support has not evolved at the same pace, particularly for languages that use non-Latin scripts.

Linguistic Mismatch in Accessibility: To study accessibility on the multilingual web, we constructed `LangCrUX`, the first large-scale dataset of 120,000 popular websites across 12 languages that primarily use non-Latin scripts, as described in Chapter 5. We used this dataset to conduct the first systematic analysis of linguistic mismatch in web accessibility. We found a widespread pattern where the visible content of a website

is presented in a local language, such as Thai, Arabic, or Hindi, but the accessibility metadata, including ARIA labels and alt-text, remains in English or contains meaningless text. For screen reader users who do not speak English, this makes the site partially or fully unusable. The screen reader switches between the native language for content and English for navigation cues, creating a cognitive burden that sighted users do not face.

We also found that existing automated accessibility testing tools do not account for this issue. Tools such as Lighthouse evaluate accessibility metadata without considering its language. As a result, websites may appear compliant while remaining inaccessible to users. To address this gap, we developed `Kizuki`, a language-aware extension for the Google Lighthouse auditing tool. `Kizuki` goes beyond checking whether alt-text exists and instead verifies whether the alt-text matches the language of the page. This represents a necessary step toward accessibility evaluations that reflect real user experience. Together, `LangCrUX` and `Kizuki` highlight the need to treat language as a core factor in accessibility engineering and provide practical tools to detect and address multilingual accessibility failures.

The Paradox of Multilingual Participation: Complementing the user-facing analysis, Chapter 6 examined the developer-facing reality of multilingualism through a longitudinal study of GitHub activity spanning 9.14 billion messages and 62,500 repositories. We showed a steady and sustained rise in non-English participation in open source, driven primarily by developer communities in China, Russia, South Korea, and other non-English-dominant regions.

Our analysis revealed that multilingualism is no longer confined to discussions and documentation but is increasingly present within source code itself, particularly in comments and string literals. This trend challenges the long-standing assumption of English as the exclusive working language of open-source development. At the same time, we identified a clear visibility penalty. Repositories operating in non-English or mixed-language settings tend to attract fewer contributors and receive less attention than English-dominant projects. This suggests that while participation barriers for non-English developers are gradually lowering, barriers to global reach and integration remain largely intact.

These findings point to the emergence of linguistic silos within open source. Multilingual practices support local inclusion but can limit cross-community collaboration in the absence of better language mediation. Addressing this tension will require improved translation support, language-aware tooling, and a rethinking of platform norms that continue to privilege English as the default medium of collaboration.

10.1.4 Redefining Vulnerability Analysis for Modern Systems

The final part of this dissertation (Chapter 7, 8, 9) examines the technical challenges of securing modern systems. It focuses on how evolving software architectures, emerging vulnerability classes, and increasing system complexity strain existing security analysis practices. This part addresses three interconnected challenges: the need for system-level reasoning beyond isolated code fragments, the lack of reproducible evaluation for emerging vulnerabilities, and the limitations of traditional static analysis techniques

when applied to dynamic, compositional systems.

Systematizing ReDoS: We provided a comprehensive Systematization of Knowledge (SoK) for Regular Expression Denial of Service (ReDoS). By synthesizing academic literature with an engineering review of modern regex engines, including V8, .NET, and Python, we identified a critical disconnect between research assumptions and deployed systems. A large portion of prior work models attackers with unconstrained capabilities, such as arbitrarily long inputs or an unrestricted number of inputs, while real-world environments impose practical limits through rate limits, request size caps, timeouts, and resource isolation.

Our analysis further shows that modern regex engines have evolved significantly. Many now incorporate linear-time matching strategies, bounded backtracking, memoization, or explicit execution limits, which invalidate or weaken several classical attack constructions. As a result, the presence of a theoretically "slow" regular expression does not necessarily translate into a practical denial-of-service vulnerability. Exploitability depends on a combination of engine behavior, configuration, workload, and surrounding system architecture.

By grounding ReDoS in both academic work and real-world engine behavior, this work reframes how ReDoS vulnerabilities should be interpreted and evaluated. Rather than treating ReDoS as a purely syntactic property of regular expressions, our findings emphasize the need to assess whether an attack remains effective under realistic deployment conditions. This perspective recalibrates the ReDoS research agenda, shifting attention from identifying pathological patterns in isolation to validating exploitability within concrete systems.

Benchmarking with SecBench.js: To address the reproducibility crisis in vulnerability research, we introduced SecBench.js, the first executable benchmark suite for server-side JavaScript vulnerabilities. By compiling 600 validated vulnerabilities with executable exploits, SecBench.js provides the ground truth necessary to rigorously evaluate vulnerability detection tools against real-world complexity. The necessity of this dynamic approach was underscored by our findings: we identified 168 package versions that were explicitly mislabeled as *safe* in public databases but remained exploitable in practice. This discrepancy underscores a critical fragility in the modern software supply chain, the reliance on static metadata over dynamic verification, and demonstrates that without executable benchmarks, the community remains blind to the true state of ecosystem security.

Beyond standard evaluation, SecBench.js enables analyses such as identifying flawed fixes, detecting regressions, and localizing security-critical behavior within complex dependency trees. More broadly, this chapter shows that executable benchmarks are essential for grounding vulnerability analysis in the real execution semantics of modern software systems.

Graphia: GNN-Based Call Graph Construction Finally, in Chapter 9, we addressed the limitations of traditional static analysis for dynamic languages through Graphia, a graph neural network (GNN) based approach to JavaScript call graph

construction. JavaScript's dynamic features, such as dynamic property access, higher-order functions, and runtime code evaluation, routinely defeat deterministic analysis techniques and lead to incomplete or imprecise call graphs.

GRAPHIA reframes call graph construction as a link prediction problem over a program graph, allowing learning-based reasoning to complement static logic. By leveraging structural and semantic relationships across the whole program, GRAPHIA recovers call edges that are difficult to infer through static analysis alone. Our evaluation shows that GRAPHIA ranks the correct target function within the top five candidates for 72% of unresolved call sites, demonstrating that learned representations can meaningfully reduce uncertainty in complex, multi-file JavaScript applications.

More broadly, this chapter shows that learning-based approaches can serve as practical complements to static analysis rather than replacements. By operating at the level of whole-program structure, GRAPHIA illustrates how system-level reasoning can help bridge the gap between highly abstracted software and the analytical visibility required for effective security analysis in modern software systems.

10.2 Future Work: Navigating the Frontiers of Socio-Technical Systems

The findings of this dissertation highlight the close coupling between human factors such as language, economics, culture, and demographics and technical systems. As modern software engineering continues to evolve, new challenges arise at these intersections. The growing use of AI-generated code, the increasing depth and fragility of software supply chains, and the rising sophistication of linguistically driven attacks call for a renewed research agenda. In the following, we outline directions for future work that build directly on the foundations established in this dissertation.

10.2.1 Language-Induced Security Risks

The linguistic diversity documented in Chapter 5 and 6 introduces a growing security challenge rather than a purely demographic shift. As software systems, development environments, and user interfaces adopt broader multilingual support, language encoding and rendering mechanisms become new attack surfaces. Attacks such as Glassworm and Trojan Source reveal that current static analysis and code review practices implicitly trust visual representations of code, an assumption that breaks down in Unicode-rich environments.

Current defenses are ill-suited to this threat model. Existing static analysis and code review tools largely assume a sound correspondence between what developers see and what the compiler or interpreter executes. They operate on byte streams or abstract syntax trees, but rarely account for how code is rendered in editors or perceived by humans. As a result, attacks based on invisible characters, bidirectional overrides, or homoglyph substitution can bypass both automated tools and manual review. Defensive responses often rely on coarse mitigations, such as banning entire Unicode categories, which conflict with legitimate multilingual usage and undermine inclusivity.

Future work should develop security analyses that explicitly incorporate linguistic and visual semantics. This includes rendering-aware tools that detect discrepancies between visual and executable code, as well as context-sensitive policies that distinguish benign multilingual content from malicious manipulation. In parallel, localized threat modeling is needed for phishing and impersonation attacks targeting non-Latin scripts, particularly in the Global South. Integrating linguistic context, regional threat intelligence, and visual integrity checks into security pipelines is essential to ensure that multilingual software systems remain both secure and usable.

10.2.2 A New Approach to Accessibility: Conversational AI Agents

The findings in Chapter 4 and 5 highlight the limitations of current accessibility approaches, which often emphasize formal compliance over actual usability. Techniques such as alt text or ARIA labels are necessary but frequently insufficient for complex and dynamic interfaces. For blind and low-vision users, interacting with modern applications through linear screen reader output can remain slow, fragmented, and cognitively demanding.

From Compliance to Conversation Future work should explore accessibility mediated through conversational AI agents that act as interactive intermediaries between users and software systems. Rather than passively reading interface elements or alt text, such agents could interpret visual and semantic content and respond to user queries in a dialog-driven manner. For example, instead of announcing every image on a page, a user could ask, "What is happening in this image?" or "Are there any warnings or prices shown here?" and receive a concise, relevant description. The interaction could naturally continue with follow-up questions, allowing users to refine their requests as part of an ongoing conversation.

This conversational model would allow users to actively steer the interaction, requesting only the information they need at a given moment. Importantly, this paradigm is not limited to web pages. Similar agents could support accessibility across mobile applications, desktop software, and other interactive systems where accessibility metadata is missing or inconsistently implemented.

Challenges: Feasibility, Cost, and Trust This approach introduces several open challenges that future research must address. Running multimodal AI agents locally may not be feasible on low-end or older mobile devices due to computational, energy, and cost constraints, particularly in the Global South. Future work should therefore explore lightweight models, hybrid on-device and cloud-based execution, and cost-aware deployment strategies that avoid excluding users with limited resources.

In addition, conversational accessibility agents must operate within secure and well-defined sandbox. Sensitive actions, such as form submission, purchases, or account changes, require explicit confirmation and transparent user control. Addressing the security, privacy, and usability implications of such systems is essential for conversational AI agents to become a practical and inclusive complement to existing accessibility practices.

10.2.3 Security and Usability Trade-offs of Multilingual Regex

Regular expressions sit at the core of input validation in modern software systems, yet most regex usage remains implicitly rooted in ASCII-centric assumptions. As applications expand to support global user bases, developers face a persistent tension between inclusivity and security. Regex patterns that are too restrictive exclude legitimate users by rejecting non-ASCII names or inputs, while overly permissive patterns risk semantic bypasses and logic errors.

Multilingual regex introduces an underexplored challenge: semantic correctness across character sets. Simple validation patterns such as `^[A-Za-z]+$`, often intended to accept "letters only," silently reject valid names containing non-Latin scripts, excluding large portions of the global population. Conversely, constructs such as `\w` or `\p{L}` can unintentionally admit characters from unexpected scripts, enabling homoglyph-based filter bypasses. These issues are compounded by inconsistencies across regex engines. Case folding, digit matching, and Unicode normalization behave differently in JavaScript, Python, and .NET, undermining developers' assumptions about regex portability and correctness[1][2].

Future work should address this gap through systematic measurement and targeted tooling that make multilingual regex behavior explicit and testable. In particular, this involves:

1. **Mining open-source repositories** to study how developers currently validate non-ASCII inputs (e.g., names written in local scripts), and classifying common anti-patterns that lead to either exclusion of legitimate users or semantic security bypasses.

2. **Fuzzing regex engines** with a specific focus on Unicode properties and large character sets to understand how multilingual inputs interact with engine semantics, including identifying Unicode constructs that trigger performance issues or inconsistent behavior across engines such as V8 and .NET.

3. **Developing safe-by-default regex primitives** or libraries for common multilingual data types (e.g., validating personal names or email addresses across languages according to relevant standards). These abstractions should encapsulate Unicode normalization and robustness concerns, reducing the cognitive and security burden on developers.

10.2.4 Benchmarking Regex Input Generation

Another issue with regular expressions concerns how their behavior is exercised through input string generation. In practice, regex validation is tested mainly by generating example strings, whether in unit tests, fuzzing, or security analyses. As a result, the effectiveness of regex testing depends critically on the quality and coverage of the generated inputs. In Chapter 7, we reviewed a broad class of techniques that generate adversarial input strings to probe worst-case performance behavior. This line of work

[1] https://openssf.org/blog/2024/06/18/know-your-regular-expressions-securing-input-validatio
[2] https://sethmlarson.dev/regex-$-matches-end-of-string-or-newline

demonstrates the effectiveness of automated string generation, but it is narrowly focused on performance degradation and denial-of-service risks. Outside this setting, string generation remains largely ad hoc, with limited guidance on how to test the broader semantic behavior of regexes.

More generally, there is no widely adopted benchmark or oracle for evaluating regex string generation tools beyond performance-oriented scenarios. Developers and researchers lack principled ways to assess whether generated inputs meaningfully exercise a regex's semantic space. Existing notions of coverage, such as NFA or DFA state and transition coverage, provide only limited insight. They fail to capture key constructs including lookaheads, lookbehinds, boundary conditions, and interactions between alternatives. As a result, a regex can achieve high structural coverage while still exhibiting semantic bugs or unintended behavior.

This motivates a shift from coverage of regex internals to *coverage of inputs*. Future work should focus on defining benchmarks and evaluation criteria for regex string generation tools. Given a regex, the objective should be to generate a compact yet expressive set of test strings that collectively expose intended and unintended behaviors. Such sets should include minimal and maximal matching strings, representative matches for each alternative, boundary cases around repetitions and optional components, and near-miss inputs that differ by a single character or position. The emphasis should be on producing few but informative inputs, rather than exhaustively enumerating the language.

Building on this idea, future research could establish a shared benchmark for regex input generation. Instead of scoring tools by raw coverage metrics, the benchmark would evaluate how effectively a generated input set distinguishes overly permissive behavior, unintended exclusions, or semantic inconsistencies across engines. Different generation strategies, automata-based enumeration, search-based methods, or learning-guided approaches, could then be compared on a common footing.

10.2.5 AI-Assisted Vulnerability Analysis of Bundled Artifacts

While Chapter 9 addressed the challenge of call graph composition at the source level, the deployment reality of the modern web introduces a deeper layer of *Structural Opacity*. Modern JavaScript applications are predominantly deployed as bundled artifacts [404], where minification, transpilation, and runtime loaders obliterate original file boundaries and semantic identifiers. As a result, identifying vulnerabilities inside these bundles remains a major open challenge, as program structure is effectively flattened into an opaque stream of logic that defies traditional static analysis.

Future work should explore the use of AI-assisted techniques, including large language models (LLMs), as a semantic recovery engine for bundled code. We propose a neuro-symbolic approach in which, rather than treating bundles as opaque inputs, models are used to "de-minify" artifacts by recognizing bundler-specific patterns (e.g., Webpack boilerplate) and inferring the semantic roles of renamed functions. When combined with lightweight static analysis to verify structural and control-flow constraints, such an approach could enable the reconstruction of approximate call graphs and dependency structures directly from production artifacts, bypassing the need for often-unavailable

source maps.

Beyond structural recovery, this direction offers significant promise for Software Composition Analysis (SCA). A critical limitation of current SCA tools is their inability to determine reachability within bundled code, frequently flagging vulnerabilities in libraries that have been tree-shaken or are never invoked. By reasoning over reconstructed execution behavior, AI-assisted analysis could distinguish between the mere presence of a vulnerable dependency and its active use along execution paths.

10.2.6 AI-Assisted Cybersecurity Analysis

AI-assisted techniques, particularly large language models (LLMs), have generated significant interest as potential tools for vulnerability detection and proof-of-concept (PoC) exploit generation. However, empirical evidence shows that current capabilities fall well short of this promise. Recent evaluations of autonomous vulnerability-finding agents report success rates of only about 38.9% under ideal conditions, dropping to as low as 11% for certain scenarios [305]. At the same time, broader reviews of the literature suggest a persistent "glass ceiling," with LLM-based approaches often stalling at roughly 27–30% success when generating working exploits for real-world vulnerabilities [524]. In practice, models frequently hallucinate exploits that do not execute, misidentify root causes, or fail to navigate the program structure required to reliably trigger vulnerabilities.

A key limitation is that LLMs lack grounding in the program's execution context. They generate code based on learned statistical patterns, without explicit models of control flow, data flow, or runtime state, which limits their ability to reason about execution-dependent vulnerabilities [102, 517, 228, 526]. This limitation becomes more pronounced in large or modular codebases, where vulnerability triggers depend on multi-step interactions across functions and objects. Benchmark contamination further obscures true progress, as memorization of public vulnerabilities can inflate apparent performance without improving generalization.

Future work should therefore focus on hybrid, neuro-symbolic approaches that integrate learning-based models, including LLMs, with program analysis artifacts. Rather than providing models with raw source code alone, systems should supply structured context such as stack traces, call graphs (e.g., generated by Graphia), and data-flow paths. For example, a stack trace can guide a model toward the relevant execution path when generating a PoC, while call graphs can constrain reasoning to reachable functions. Retrieval-augmented generation tailored to code can further enable models to query relevant program fragments on demand, reducing hallucination and improving reliability. In this setting, executable benchmarks such as SecBench.js provide a valuable foundation for future research, enabling systematic evaluation of AI-assisted techniques against real-world vulnerabilities with known ground truth and reproducible exploits.

10.3 Final Remarks

Software engineering is moving toward greater complexity, driven by global expansion and the rising abstraction of modern software stacks, particularly in web-based and JavaScript-heavy systems. This dissertation shows how economic and linguistic differences shape how systems are built, who they serve, and where they fail. It also highlights how emerging vulnerabilities challenge traditional security and software engineering methods, which are often ill-equipped to address these realities. Addressing these systemic limitations demands a shift beyond purely technical solutions toward a socio-technical perspective in which resilience and inclusivity are treated as foundational design constraints. By grounding future research in rigorous empirical methods and real-world deployment contexts, this approach aims to build software systems that better reflect how they are actually used. The broader goal is not only to improve individual tools, but to reshape how we reason about software systems as socio-technical artifacts that must remain secure, usable, and robust across languages, abilities, and economic contexts.

Bibliography

Author's Papers for this Thesis

[P1] Bhuiyan, M. H. M., Varvello, M., Staicu, C.-A., and Zaki, Y. Digital Disparities: A Comparative Web Measurement Study Across Economic Boundaries. In: *Proceedings of the ACM on Web Conference 2025*. WWW '25. Association for Computing Machinery, Sydney NSW, Australia, 2025, 1889–1900.

[P2] Bhuiyan, M. H. M., Varvello, M., Staicu, C.-A., and Zaki, Y. Non-Western Perspectives on Web Inclusivity: A Study of Accessibility Practices in the Global South. In: *2025 IEEE/ACM Symposium on Software Engineering in the Global South (SEiGS)*. IEEE. 2025, 59–64.

[P3] Bhuiyan, M. H. M., Varvello, M., Zaki, Y., and Staicu, C.-A. Not All Visitors are Bilingual: A Measurement Study of the Multilingual Web from an Accessibility Perspective. In: *Proceedings of the 2025 ACM Internet Measurement Conference*. IMC '25. Association for Computing Machinery, USA, 2025, 871–881.

[P4] Bhuiyan, M. H. M., Bala Kumar, M. K., and Staicu, C.-A. "Write in English, Nobody Understands Your Language Here": A Study of Non-English Trends in Open-Source Repositories. In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2026.

[P5] Bhuiyan, M. H. M., Çakar, B., Burmane, E. H., Davis, J. C., and Staicu, C.-A. SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS). In: *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '25. Association for Computing Machinery, 2025, 1659–1675.

[P6] Bhuiyan, M. H. M., Parthasarathy, A. S., Vasilakis, N., Pradel, M., and Staicu, C.-A. SecBench. js: An executable security benchmark suite for server-side JavaScript. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, 1059–1070.

[P7] Bhuiyan, M. H. M., De Stefano, G., Pellegrino, G., and Staicu, C.-A. Call Me Maybe: Enhancing JavaScript Call Graph Construction using Graph Neural Networks. *arXiv preprint arXiv:2506.18191* (2025).

Other Papers of the Author

[S1] Arifuzzaman, M., Bhuiyan, M., Gumus, M., and Arslan, E. Be SMART, Save I/O: A probabilistic approach to avoid uncorrectable errors in storage systems. In: *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2022, 256–266.

[S2] Bhuiyan, M. H. M. The Call Graph Chronicles: Unleashing the Power Within. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. Association for Computing Machinery, San Francisco, CA, USA, 2023, 2210–2212.

[S3] Bhuiyan, M. H. M. Developers' Practices, Web Accessibility, and Language Barriers: Analyzing Challenges and Opportunities for Inclusive Digital Development. In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, New York, NY, USA, 2025, 846–851.

[S4] Bhuiyan, M. H. M. and Staicu, C.-A. A Tale of Frozen Clouds: Quantifying the Impact of Algorithmic Complexity Vulnerabilities in Popular Web Servers. *arXiv preprint arXiv:2211.11357* (2022).

Other references

[1] Abanumy, A., Al-Badi, A.-B., and Mayhew, M. E-government website accessibility: in-depth evaluation of saudi arabia and oman. *The Electronic Journal of e-Government* 3, 3 (2005), 99–106.

[2] Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. Why do developers use trivial packages? an empirical case study on npm. In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, 385–395.

[3] Adepoju, S. A., Shehu, I. S., and Bake, P. Accessibility evaluation and performance analysis of e-government websites in nigeria. *Journal of Advanced Information Technology* 7, 1 (2016), 49.

[4] Adobe Inc. *JPEG vs. JPEG 2000: Which is better? | Adobe*. en-US. URL: `https://www.adobe.com/creativecloud/file-types/image/comparison/jpeg-vs-jpeg-2000.html`.

[5] Ahmad, T., Zaki, Y., Pötsh, T., Chen, J., Sathiaseelan, A., and Subramanian, L. Gaius: a new mobile content creation and diffusion ecosystem for emerging regions. In: *Proceedings of the Tenth International Conference on Information and Communication Technologies and Development*. ICTD '19. Association for Computing Machinery, Ahmedabad, India, 2019.

[6] Ahmed, M., Yan, Z., Islam, S., and Sunny, M. M. H. Accessibility analysis of bangladesh government websites based on wcag 2.0. *Int. J. Comput. Sci. Mob. Comput* 9, 3 (2020), 157–167.

[7] Aho, A. V. Pattern Matching in Strings. In: *Formal Language Theory*. Ed. by Book, R. V. Academic Press, Jan. 1980, 325–347.

[8] Akgul, Y. and Vatansever, K. Web accessibility evaluation of government websites for people with disabilities in turkey. *Journal of Advanced Management Science* 4, 3 (2016), 201.

[9] Alajarmeh, N. Evaluating the accessibility of public health websites: an exploratory cross-country study. *Universal access in the information society* 21, 3 (2022), 771–789.

[10] AlDahoul, N., Hong, J., Varvello, M., and Zaki, Y. Towards a world wide web powered by generative ai. *Scientific reports* 15, 1 (2025), 7251.

[11] Alebachew, Y. B., Ko, M., and Brown, C. Are we on the same page? examining developer perception alignment in open source code reviews. In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 2025, 57–67.

[12] Alhamdan, A. and Staicu, C.-A. {Sanddriller}: a {fully-automated} approach for testing {language-based}{javascript} sandboxes. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, 3457–3474.

[13] Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. Typilus: neural type hints. In: *Proceedings of the 41st acm sigplan conference on programming language design and implementation*. 2020, 91–105.

[14] Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[15] Allauzen, C., Mohri, M., and Rastogi, A. General algorithms for testing the ambiguity of finite automata. In: *Developments in Language Theory*. Ed. by Ito, M. and Toyama, M. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, 108–120.

[16] Alon, U., Brody, S., Levy, O., and Yahav, E. Code2seq: generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[17] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. Code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3 (2018), 1–29.

[18] Alsaggaf, W., Baaqeel, H., Alkhuraiji, S., and Brdesee, H. The inclusion of python as introductory computer programming in the preparatory year of higher education: modeling for students' perceptions. *IJCSNS International Journal of Computer Science and Network Security* 22, 3 (2022), 565–574.

[19] Amazon Web Services, I. *Regex Pattern Set Match Rule Statement - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced*. 2024. URL: `https://docs.aws.amazon.com/waf/latest/developerguide/waf-rule-statement-type-regex-pattern-set-match.html`.

[20] American Foundation for the Blind. *Access Barriers in Websites and Mobile Apps | American Foundation for the Blind*. en-US. URL: https://www.afb.org/research-and-initiatives/bdis-series/barriers-digital-inclusion-survey/access-barriers-web-mobile.

[21] Amjad, A. H., Shafiq, Z., and Gulzar, M. A. Blocking javascript without breaking the web. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2023.

[22] An, P., Shafi, R., Mughogho, T., and Onyango, O. A. Multilingual email phishing attacks detection using osint and machine learning. *arXiv preprint arXiv:2501.08723* (2025).

[23] Android Developers. *Localize Your App | App Architecture | Android Developers*. https://developer.android.com/guide/topics/resources/localization. 2024.

[24] Antal, G., Hegedus, P., Tóth, Z., Ferenc, R., and Gyimóthy, T. Static JavaScript call graphs: a comparative study. In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2018, 177–186.

[25] Antal, G., Hegedűs, P., Herczeg, Z., Lóki, G., and Ferenc, R. Is javascript call graph extraction solved yet? a comparative study of static and dynamic tools. *IEEE Access* 11 (2023), 25266–25284.

[26] Apple Inc. *VoiceOver User Guide for Mac — support.apple.com*. https://support.apple.com/en-gb/guide/voiceover/welcome/mac.

[27] Archive, H. *State of the Web*. https://httparchive.org/reports/state-of-the-web. 2021.

[28] Auer, P. *Code-switching in conversation: Language, interaction and identity*. Routledge, 2013.

[29] Avelino, G., Passos, L., Hora, A., and Valente, M. T. A novel approach for estimating truck factors. In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, May 2016, 1–10.

[30] Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., and Brumley, D. Automatic exploit generation. *Communications of the ACM* 57, 2 (2014), 74–84.

[31] B.V., E. *Accessing Event Data and Fields*. https://www.elastic.co/guide/en/logstash/current/event-dependent-configuration.html#conditionals. 2024.

[32] Babel Team. *Babel*. 2024. URL: https://babeljs.io/.

[33] Backhouse, K. *How to fix a ReDoS*. May 2023. URL: https://github.blog/2023-05-09-how-to-fix-a-redos/.

[34] Bai, Z., Wang, K., Zhu, H., Cao, Y., and Jin, X. Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021, 1575–1588.

[35] Balali, S., Steinmacher, I., Annamalai, U., Sarma, A., and Gerosa, M. A. Newcomers' barriers... is that all? an analysis of mentors' and newcomers' barriers in oss projects. *Computer Supported Cooperative Work (CSCW)* 27, 3 (2018), 679–714.

[36] Bank, W. *World Development Report 2016: Digital Dividends*. World Bank Publications, 2016. URL: https://www.worldbank.org/en/publication/wdr2016.

[37] Bank, W. *World Bank Open Data*. en. URL: https://data.worldbank.org.

[38] Baowaly, M. K., Hossain, M., and Bhuiyan, M. Accessibility analysis and evaluation of government-websites' in developing countries: case study bangladesh. *Computer Engineering and Intelligent Systems* 3, 4 (2012), 1–9.

[39] Barbareschi, G., Swaminathan, M., Pimenta Freire, A., and Holloway, C. Challenges and strategies for accessibility research in the global south: a panel discussion. In: *Proceedings of the X Latin American Conference on Human Computer Interaction*. 2021, 1–5.

[40] Barlas, E., Du, X., and Davis, J. C. Exploiting input sanitization for regex denial of service. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Association for Computing Machinery, Pittsburgh, Pennsylvania, 2022, 883–895.

[41] Barman, U., Das, A., Wagner, J., and Foster, J. Code mixing: a challenge for language identification in the language of social media. In: *Proceedings of the first workshop on computational approaches to code switching*. 2014, 13–23.

[42] Barth, A. *HTTP State Management Mechanism*. https://www.rfc-editor.org/rfc/rfc6265. 2011.

[43] Barth, J. M. An interprocedural data flow analysis algorithm. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 1977, 119–131.

[44] Baxter, G. and Sommerville, I. Socio-technical systems: from design methods to systems engineering. *Interacting with computers* 23, 1 (2011), 4–17.

[45] Bay, J. *Inside the Accessibility Tree*. https://developer.chrome.com/blog/full-accessibility-tree. 2021.

[46] Becker, B. A. Parlez-vous java? bonjour la monde!= hello world: barriers to programming language acquisition for non-native english speakers. In: *PPIG*. 2019.

[47] Becker, B. A., Murray, C., Tao, T., Song, C., McCartney, R., and Sanders, K. Fix the first, ignore the rest: dealing with multiple compiler error messages. In: *Proceedings of the 49th ACM technical symposium on computer science education*. 2018, 634–639.

[48] Belshe, M., Peon, R., and Thomson, M. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. https://www.rfc-editor.org/rfc/rfc7540. 2015.

[49] Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: a learnable representation of code semantics. *Advances in neural information processing systems* 31 (2018).

[50] Berglund, M., Van Der Merwe, B., and Van Litsenborgh, S. Regular Expressions with Lookahead. *JUCS - Journal of Universal Computer Science* 27, 4 (Apr. 2021), 324–340.

[51] Berglund, M., Van Der Merwe, B., Watson, B., and Weideman, N. On the Semantics of Atomic Subgroups in Practical Regular Expressions. In: *Implementation and Application of Automata*. Ed. by Carayol, A. and Nicaud, C. Vol. 10329. Springer International Publishing, Cham, 2017, 14–26.

[52] Bergsma, S., McNamee, P., Bagdouri, M., Fink, C., and Wilson, T. Language identification for creating language-specific twitter collections. In: *Proceedings of the second workshop on language in social media*. 2012, 65–74.

[53] Berkholz, D. *The Size of Open-Source Communities and Its Impact Upon Activity, Licensing, and Hosting — redmonk.com*. https://redmonk.com/dberkholz/2013/04/22/the-size-of-open-source-communities-and-its-impact-upon-activity-licensing-and-hosting/. 2013.

[54] Bhuiyan, M. H. M., Parthasarathy, A. S., Vasilakis, N., Pradel, M., and Staicu, C.-A. Secbench.js: an executable security benchmark suite for server-side javascript. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. IEEE Press, Melbourne, Victoria, Australia, 2023, 1059–1070.

[55] Bhuiyan, M. H. M. and Staicu, C.-A. A tale of frozen clouds: quantifying the impact of algorithmic complexity vulnerabilities in popular web servers. *arXiv preprint arXiv:2211.11357* (2022).

[56] Bhuiyan, M. H. M., Varvello, M., Staicu, C.-A., and Zaki, Y. Non-western perspectives on web inclusivity: a study of accessibility practices in the global south. In: *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. IEEE / ACM, Ottawa, Canada, 2025.

[57] Bidlingmaier, M. *An Additional Non-Backtracking RegExp Engine*. Jan. 2021. URL: https://v8.dev/blog/non-backtracking-regexp.

[58] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The parsec benchmark suite: characterization and architectural implications. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, 72–81.

[59] Bingler, S., West, M., and Wilander, J. *Cookies: HTTP State Management Mechanism*. https://httpwg.org/http-extensions/draft-ietf-httpbis-rfc6265bis.html. 2025.

[60] Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B. Does distributed development affect software quality? an empirical case study of windows vista. *Communications of the ACM* 52, 8 (2009), 85–93.

[61] Bischof, Z. S., Rula, J. P., and Bustamante, F. E. In and out of cuba: characterizing cuba's connectivity. In: *Proceedings of the 2015 Internet Measurement Conference*. IMC '15. Association for Computing Machinery, Tokyo, Japan, 2015, 487–493.

[62] Bishop, M. *HTTP/3*. `https://www.rfc-editor.org/rfc/rfc9114`. 2022.

[63] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. The dacapo benchmarks: java benchmarking development and analysis. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, 169–190.

[64] Boland, T. and Black, P. E. Juliet 1. 1 c/c++ and java test suite. *Computer* 45, 10 (2012), 88–90.

[65] Borges, H., Hora, A., and Valente, M. T. Predicting the popularity of github repositories. In: *Proceedings of the The 12th international conference on predictive models and data analytics in software engineering*. 2016, 1–10.

[66] Borges, H., Hora, A., and Valente, M. T. Understanding the factors that impact the popularity of github repositories. In: *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2016, 334–344.

[67] Borges, H. and Valente, M. T. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[68] Borodin, Y., Bigham, J. P., Dausch, G., and Ramakrishnan, I. More than meets the eye: a survey of screen-reader browsing strategies. In: *Proceedings of the 2010 international cross disciplinary conference on web accessibility (W4A)*. 2010, 1–10.

[69] Bosu, A. and Sultana, K. Z. Diversity and inclusion in open source software (oss) projects: where do we stand? In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2019, 1–11.

[70] Boucher, N. and Anderson, R. Trojan source: invisible vulnerabilities. In: *32nd USENIX security symposium (USENIX Security 23)*. 2023, 6507–6524.

[71] Bravenboer, M. and Smaragdakis, Y. Strictly declarative specification of sophisticated points-to analyses. In: *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 2009, 243–262.

[72] Bregolin, J. *Communication effort and the cost of language: Evidence from stack overflow*. Tech. rep. 2022.

[73] Bresson, X. and Laurent, T. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553* (2017).

[74] Brezis, E. S., Krugman, P. R., and Tsiddon, D. *Leapfrogging: A theory of cycles in national technological leadership*. 1991.

[75] British Council. *Moving from the Margins: Mainstreaming persons with disabilities in Pakistan | British Council*. en. URL: https://www.britishcouncil.org/research-insight/mainstreaming-persons-disabilities-pakistan.

[76] Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490* (2018).

[77] Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. Geometric deep learning: grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478* (2021).

[78] Brown, F., Narayan, S., Wahby, R. S., Engler, D., Jhala, R., and Stefan, D. Finding and preventing bugs in javascript bindings. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, 559–578.

[79] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[80] Buchanan, J. *Regex Parsing: Extract Data from Logs*. Mar. 2023. URL: https://newrelic.com/blog/how-to-relic/extracting-log-data-with-regex.

[81] Bureau of Internet Accessibility. *What Are the Differences Between Mobile and Website Accessibility?* en. URL: https://www.boia.org/blog/what-is-the-european-unions-web-accessibility-directive.

[82] Buzatu, B.-M. Data hound: Analysing non-English data smells in large code datasets. PhD thesis. Delft University of Technology, 2025.

[83] Casalegno, E. Usability of partially localised websites in switzerland: a study with screen reader users. *Unpublished MA Dissertation. University of Geneva, Switzerland* (2018).

[84] Caswell, B. *Cyber Grand Challenge Corpus*. Ed. by Lunge Technology. URL: http://www.lungetech.com/cgc-corpus/.

[85] Centre for Disability in Development. *KEY FOCUS AREAS*. en-US. URL: https://cdd.org.bd/about-disability/.

[86] Chakraborty, M., Olivares, R., Sridharan, M., and Hassanshahi, B. Automatic root cause quantification for missing edges in javascript call graphs (extended version). *arXiv preprint arXiv:2205.06780* (2022).

[87] Chaqfeh, M., Asim, R., AlShebli, B., Zaffar, M. F., Rahwan, T., and Zaki, Y. Towards a world wide web without digital inequality. *Proceedings of the National Academy of Sciences* 120, 3 (2023), e2212649120.

[88] Chaqfeh, M., Haseeb, M., Hashmi, W., Inshuti, P., Ramesh, M., Varvello, M., Subramanian, L., Zaffar, F., and Zaki, Y. To block or not to block: accelerating mobile web pages on-the-fly through javascript classification. In: *Proceedings of the 2022 International Conference on Information and Communication Technologies and Development*. ICTD '22. Association for Computing Machinery, Seattle, WA, USA, 2023.

[89] Chaqfeh, M., Zaki, Y., Hu, J., and Subramanian, L. Jscleaner: de-cluttering mobile webpages through javascript cleanup. In: *Proceedings of The Web Conference 2020*. 2020, 763–773.

[90] Chattopadhyay, A., Li, A. W., and Mamouras, K. Verified and efficient matching of regular expressions with lookaround. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP '25. Association for Computing Machinery, Denver, CO, USA, 2025, 198–213.

[91] Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).

[92] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[93] Chetty, M., Sundaresan, S., Muckaden, S., Feamster, N., and Calandro, E. Measuring broadband performance in south africa. In: *Proceedings of the 4th Annual Symposium on Computing for Development*. ACM DEV-4 '13. Association for Computing Machinery, Cape Town, South Africa, 2013.

[94] Chida, N. and Terauchi, T. Repairing DoS Vulnerability of Real-World Regexes. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2022, 2060–2077.

[95] Choudhary, M. *Viewpoint: The pitfalls of India's biometric ID scheme — bbc.com*. https://www.bbc.com/news/world-asia-india-43619944.

[96] Chow, Y. W., Schäfer, M., and Pradel, M. Beware of the unexpected: bimodal taint analysis. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, 211–222.

[97] Cohen, G., Afshar, S., Tapson, J., and Van Schaik, A. Emnist: extending mnist to handwritten letters. In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, 2921–2926.

[98] Community, N. *Module ngx_http_core_module*. 2024. URL: http://nginx.org/en/docs/http/ngx_http_core_module.html%5C#large_client_header_buffers.

[99] Community, R. *About Ruby*. 2024. URL: https://www.ruby-lang.org/en/about/.

[100] Community, R. *Ruby/Ruby*. Aug. 2024. URL: https://github.com/ruby/ruby.

[101] Le-Cong, T., Kang, H. J., Nguyen, T. G., Haryono, S. A., Lo, D., Le, X.-B. D., and Huynh, Q. T. Autopruner: transformer-based call graph pruning. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, 520–532.

[102] Le-Cong, T., Le, B., and Murray, T. Can llms reason about program semantics? a comprehensive evaluation of llms on formal specification inference. *arXiv preprint arXiv:2503.04779* (2025).

[103] Contributors, . *Compilation and Reuse in Regular Expressions*. Sept. 2021. URL: https://learn.microsoft.com/en-us/dotnet/standard/base-types/compilation-and-reuse-in-regular-expressions.

[104] Contributors, . *Regular Expression Behavior*. Sept. 2021. URL: https://learn.microsoft.com/en-us/dotnet/standard/base-types/details-of-regular-expression-behavior.

[105] Contributors, . *Backtracking in .NET Regular Expressions*. Aug. 2023. URL: https://learn.microsoft.com/en-us/dotnet/standard/base-types/backtracking-in-regular-expressions.

[106] Contributors, . *Best Practices for Regular Expressions in .NET*. Oct. 2023. URL: https://learn.microsoft.com/en-us/dotnet/standard/base-types/best-practices-regex.

[107] Contributors, . *What's New in .NET Framework*. June 2023. URL: https://learn.microsoft.com/en-us/dotnet/framework/whats-new/%5C#whats-new-in-net-framework-45.

[108] Contributors, . *C# Language Specification*. Mar. 2024. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction.

[109] Corporation, O. *Openjdk/Jdk*. Aug. 2024. URL: https://github.com/openjdk/jdk.

[110] Corry, E., Hansen, C. P., and Nielsen, L. R. H. *Irregexp, Google Chrome's New Regexp Implementation*. 2009. URL: https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html.

[111] Cox, R. *Regular Expression Matching Can Be Simple And Fast*. Jan. 2007. URL: https://web.archive.org/web/20240529192558/https://swtch.com/%5Ctextasciitilde%20rsc/regexp/regexp1.html.

[112] Creative, C. *UN Broadband Commission Adopts A4AI "1 for 2" Affordability Target*. en-US. Jan. 2018. URL: https://a4ai.org/news/un-broadband-commission-adopts-a4ai-1-for-2-affordability-target/.

[113] Crosby, S. Denial of service through regular expressions. In: USENIX Association, Washington, D.C., Aug. 2003.

[114] Crosby, S. A. and Wallach, D. S. Denial of service via algorithmic complexity attacks. In: *12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association, Washington, D.C., Aug. 2003.

[115] Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. Programl: graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536* (2020).

[116] Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. Social coding in github: transparency and collaboration in an open software repository. In: *Proceedings of the ACM 2012 conference on computer supported cooperative work*. 2012, 1277–1286.

[117] Dados, N. and Connell, R. The global south. *Contexts* 11, 1 (2012), 12–13.

[118] Daigle, K. and GitHub Staff. *Octoverse: The State of Open Source and Rise of AI in 2023 — github.blog*. https://github.blog/news-insights/research/the-state-of-open-source-and-ai/. 2023.

[119] Daigle, L. *WHOIS Protocol Specification*. Request for Comments RFC 3912. Internet Engineering Task Force, Sept. 2004. URL: https://datatracker.ietf.org/doc/rfc3912.

[120] Das, M., Fiannaca, A. J., Morris, M. R., Kane, S. K., and Bennett, C. L. From provenance to aberrations: image creator and screen reader user perspectives on alt text for ai-generated images. In: *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 2024, 1–21.

[121] Dattolo, A., Luccio, F. L., Pirone, E., et al. Web accessibility recommendations for the design of tourism websites for people with autism spectrum disorders. *International Journal on Advances in Life Sciences* 8, 3-4 (2016), 297–308.

[122] Davis, J. *The Regular Expression Denial of Service (ReDoS) Cheat-Sheet*. June 2020. URL: https://levelup.gitconnected.com/the-regular-expression-denial-of-service-redos-cheat-sheet-a78d0ed7d865.

[123] Davis, J., Kildow, G., and Lee, D. The case of the poisoned event handler: weaknesses in the node.js event-driven architecture. In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Association for Computing Machinery, Belgrade, Serbia, 2017.

[124] Davis, J. C. Rethinking regex engines to address redos. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Association for Computing Machinery, Tallinn, Estonia, 2019, 1256–1258.

[125] Davis, J. C. *Davisjam/safe-regex*. Aug. 2024. URL: https://github.com/davisjam/safe-regex.

[126] Davis, J. C. *Davisjam/Vuln-Regex-Detector*. Aug. 2024. URL: https://github.com/davisjam/vuln-regex-detector.

[127] Davis, J. C., Coghlan, C. A., Servant, F., and Lee, D. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Association for Computing Machinery, Lake Buena Vista, FL, USA, 2018, 246–256.

[128] Davis, J. C., Michael IV, L. G., Coghlan, C. A., Servant, F., and Lee, D. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Association for Computing Machinery, Tallinn, Estonia, 2019, 443–454.

[129] Davis, J. C., Servant, F., and Lee, D. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021, 1–17.

[130] Davis, J. C., Williamson, E. R., and Lee, D. A sense of time for JavaScript and node.js: First-Class timeouts as a cure for event handler poisoning. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, Aug. 2018, 343–359.

[131] Davis, J. C. On the impact and defeat of regular expression denial of service. PhD Thesis. Virginia Tech, 2020.

[132] Davis, M. and Leroy, R. *UAX #31: Unicode Identifiers and Syntax — unicode.org*. https://www.unicode.org/reports/tr31/.

[133] Decan, A., Mens, T., and Constantinou, E. On the impact of security vulnerabilities in the npm package dependency network. In: *Proceedings of the 15th international conference on mining software repositories*. 2018, 181–191.

[134] Demoulin, H. M., Pedisich, I., Vasilakis, N., Liu, V., Loo, B. T., and Phan, L. T. X. Detecting asymmetric application-layer Denial-of-Service attacks In-Flight with FineLame. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, July 2019, 693–708.

[135] Denny, P., Prather, J., Becker, B. A., Mooney, C., Homer, J., Albrecht, Z. C., and Powell, G. B. On designing programming error messages for novices: readability and its constituent factors. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, 1–15.

[136] Deque Systems. *axe-core/doc/rule-descriptions.md at develop · dequelabs/axe-core*. en. URL: https://github.com/dequelabs/axe-core/blob/develop/doc/rule-descriptions.md.

[137] Deque Systems. *List of axe 4.7 rules | Deque University | Deque Systems — dequeuniversity.com*. https://dequeuniversity.com/rules/axe/4.7.

[138] Destefanis, G., Ortu, M., Bowes, D., Marchesi, M., and Tonelli, R. On measuring affects of github issues' commenters. In: *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering*. 2018, 14–19.

[139] Developers, C. for. *Overview of CrUX | Chrome UX Report*. en. URL: https://developer.chrome.com/docs/crux.

[140] Developers, G. *Lighthouse performance scoring | Chrome for Developers — developer.chrome.com*. https://developer.chrome.com/docs/lighthouse/performance/performance-scoring.

[141] Diggs, J., Nurthen, J., Cooper, M., and MacLeod, C. *Accessible Rich Internet Applications (WAI-ARIA) 1.2*. https://www.w3.org/TR/wai-aria-1.2/. 2023.

[142] Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: learning graph transformations to detect and fix bugs in programs. In: *International Conference on Learning Representations (ICLR)*. 2020.

[143] Dinh, S. T., Cho, H., Martin, K., Oest, A., Zeng, K., Kapravelos, A., Ahn, G.-J., Bao, T., Wang, R., Doupé, A., et al. Favocado: fuzzing the binding code of javascript engines using semantically correct test cases. In: *NDSS*. 2021.

[144] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., and Whelan, R. Lava: large-scale automated vulnerability addition. In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, 110–121.

[145] Dolby, J., Fink, S. J., and Sridharan, M. Tj watson libraries for analysis (wala). *URL http://wala. sf. net* (2015).

[146] Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., and Lee, W. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139* (2020).

[147] DuMez, K. *Lighthouse performance scoring — graphite.dev*. https://graphite.dev/guides/lighthouse-scoring.

[148] Dwivedi, V. P., Joshi, C. K., Luu, A. T., Laurent, T., Bengio, Y., and Bresson, X. *Benchmarking graph neural networks*. 2023.

[149] Ebbertz, M. *Internet statistics: Distribution of languages on the Internet*. 2002.

[150] Eclipse Foundation. *Eclipse Community Forums: Eclipse Platform » Character Encoding Problem (on Windows) | The Eclipse Foundation — eclipse.org*. https://www.eclipse.org/forums/index.php/t/104105/. 2006.

[151] Englehardt, S. and Narayanan, A. Online tracking: a 1-million-site measurement and analysis. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, 1388–1401.

[152] Ethnologue. *What are the top 200 most spoken languages? — ethnologue.com*. https://www.ethnologue.com/insights/ethnologue200/.

[153] Fallible Inc. *retirejs: Port of RetireJS in Python*. URL: https://github.com/FallibleInc/retirejslib.

[154] Fan, J., Li, Y., Wang, S., and Nguyen, T. N. Ac/c++ code vulnerability dataset with code changes and cve summaries. In: *Proceedings of the 17th international conference on mining software repositories*. 2020, 508–512.

[155] Fanou, R., Tyson, G., Francois, P., and Sathiaseelan, A. Pushing the frontier: exploring the african web ecosystem. In: *Proceedings of the 25th International Conference on World Wide Web*. WWW '16. International World Wide Web Conferences Steering Committee, Montréal, Québec, Canada, 2016, 435–445.

[156] Fatima, U. Developer social networks/open source project networks: how programmers use github (2025).

[157] Felt, A. P., Barnes, R., King, A., Palmer, C., Bentzel, C., and Tabriz, P. Measuring HTTPS adoption on the web. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Kirda, E. and Ristenpart, T. USENIX Association, 2017, 1323–1338.

[158] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: a pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[159] Feng, Z., Steinmacher, I., Gerosa, M., Menezes, T., Serebrenik, A., Milewicz, R., and Sarma, A. The multifaceted nature of mentoring in oss: strategies, qualities, and ideal outcomes. In: *2025 IEEE/ACM 18th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE. 2025, 203–214.

[160] Ferenc, R., Hegedűs, P., Gyimesi, P., Antal, G., Bán, D., and Gyimóthy, T. Challenging machine learning algorithms in predicting vulnerable javascript functions. In: *2019 IEEE/ACM 7th International workshop on realizing artificial intelligence synergies in software engineering (RAISE)*. IEEE. 2019, 8–14.

[161] Fernandes, K., Paramananthan, S., Cockburn, L., and Nganji, J. Readily available but how accessible?: an analysis of the web accessibility of healthcare-related resources. *Journal of Accessibility and Design for All* 13, 2 (2023), 188–215.

[162] Fernandes, P., Allamanis, M., and Brockschmidt, M. Structured neural summarization. *arXiv preprint arXiv:1811.01824* (2018).

[163] fido128. *Answer to "CrUX: How much data is sufficient?"* Jan. 2024. URL: https://stackoverflow.com/a/77872898.

[164] Fielding, R. T. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. Internet Engineering Task Force (IETF) RFC 7230. 2014. URL: https://tools.ietf.org/html/rfc7230.

[165] Fielding, R. T. and Taylor, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)* 2, 2 (2002), 115–150.

[166] Flynn, L., Snavely, W., Svoboda, D., VanHoudnos, N., Qin, R., Burns, J., Zubrow, D., Stoddard, R., and Marce-Santurio, G. Prioritizing alerts from multiple static analysis tools, using classification models. In: *Proceedings of the 1st international workshop on software qualities and their dependencies*. 2018, 13–20.

[167] Fong, M. W. Technology leapfrogging for developing countries. In: *Encyclopedia of Information Science and Technology, Second Edition*. IGI Global, 2009, 3707–3713.

[168] Foundation, . *Dotnet/Runtime*. Aug. 2024. URL: https://github.com/dotnet/runtime.

[169] Foundation, A. S. *core - Apache HTTP Server Version 2.2*. 2018. URL: https://httpd.apache.org/docs/2.2/mod/core.html%5C#limitrequestfieldsize.

[170] Foundation, O. *About Node.js*. 2024. URL: https://nodejs.org/en/about.

[171] Foundation, O. *Nodejs/Node*. Aug. 2024. URL: https://github.com/nodejs/node.

[172] Foundation, O. *OWASP Top 10*. https://owasp.org/Top10/. 2021.

[173] Foundation, P. S. *Python Developer's Guide*. 2024. URL: https://devguide.python.org/.

[174] Foundation, P. S. *Python/Cpython*. Aug. 2024. URL: https://github.com/python/cpython.

[175] Foundation, R. *Rust-Lang/Regex*. Aug. 2024. URL: https://github.com/rust-lang/regex.

[176] Foundation, R. *Rust-Lang/Rust*. Aug. 2024. URL: https://github.com/rust-lang/rust.

[177] Foundation, R. *What is rustc?* https://doc.rust-lang.org/stable/rustc/. 2024.

[178] Freedom Scientific. *JAWS Screen Reader Keystrokes*. https://support.freedomscientific.com/Content/Documents/Manuals/JAWS/Keystrokes.pdf.

[179] Freedom Scientific. *JAWS® – Freedom Scientific — freedomscientific.com*. https://www.freedomscientific.com/products/software/jaws/.

[180] Friedl, J. E. F. *Mastering Regular Expressions*. en. 3rd ed. O'Reilly Media, Sebastopol, CA, Aug. 2006.

[181] Fujinami, H. and Hasuo, I. Efficient Matching with Memoization for Regexes with Look-around and Atomic Grouping. In: *Programming Languages and Systems*. Ed. by Weirich, S. Vol. 14577. Springer Nature Switzerland, Cham, 2024, 90–118.

[182] Garaventa, B. and Sumner, M. *Accessible Name and Description Computation*. https://www.w3.org/TR/accname-1.2/. 2025.

[183] García Garcinuño, Á. and Torres-del-Rey, J. Multilingual accessibility in human-screen reader interaction with web content: an exploratory study. *Tradumàtica* 22 (2024), 0426–449.

[184] Gauthier, F., Hassanshahi, B., and Jordan, A. Affogato: runtime detection of injection attacks for node. js. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 2018, 94–99.

[185] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In: *International conference on machine learning*. Pmlr. 2017, 1263–1272.

[186] GitHub. *GitHub Copilot · Your AI Pair Programmer — github.com*. https://github.com/features/copilot. 2021.

[187] GitHub. *CodeQL*. 2022. URL: https://codeql.github.com/.

[188] GitHub. *Global Distribution of Developers — octoverse.github.com*. https://octoverse.github.com/2022/global-tech-talent. 2022.

[189] GitHub Staff. *Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges — github.blog*. https://github.blog/news-insights/octoverse/octoverse-2024/#the-state-of-open-source. 2024.

[190] Gkortzis, A., Mitropoulos, D., and Spinellis, D. Vulinoss: a dataset of security vulnerabilities in open-source systems. In: *Proceedings of the 15th International conference on mining software repositories*. 2018, 18–21.

[191] Gleason, Gleason, C., Pavel, A., Liu, X., Carrington, P., Chilton, L. B., and Bigham, J. P. Making memes accessible. en. In: *Proceedings of the 21st International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS '19*. New York, NY, USA, 2019, 367–376.

[192] Gligor, V. D. A Note on the Denial-of-Service Problem. In: *1983 IEEE Symposium on Security and Privacy*. Apr. 1983, 139–139.

[193] Glushkov, V. M. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (Oct. 1961), 1–53.

[194] GNU Project. *When Free Software Isn't (Practically) Superior — GNU Project — Free Software Foundation*. https://www.gnu.org/philosophy/when-free-software-isnt-practically-superior.html. 2016.

[195] Golzadeh, M., Decan, A., Constantinou, E., and Mens, T. Identifying bot activity in github pull request and issue comments. In: *2021 IEEE/ACM third international workshop on bots in software engineering (BotSE)*. IEEE. 2021, 21–25.

[196] Golzadeh, M., Decan, A., Legay, D., and Mens, T. A ground-truth dataset and classification model for detecting bots in github issue and pr comments. *Journal of Systems and Software* 175 (2021), 110911.

[197] Golzadeh, M., Legay, D., Decan, A., and Mens, T. Bot or not? detecting bots in github pull request activity based on comment similarity. In: *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*. 2020, 31–35.

[198] Gong, L. *Dynamic analysis for JavaScript code*. University of California, Berkeley, 2018.

[199] Google. *AMP is a web component framework to easily create user-first web experiences - amp.dev*. https://amp.dev. 2019.

[200] Google. *Core Web Vitals report - Search Console Help*. URL: https://support.google.com/webmasters/answer/9205520?hl=en.

[201] Google. *Load mobile pages faster with Web Light - Google Search Help*. URL: https://support.google.com/websearch/answer/9836344?hl=en.

[202] Google Chrome Developers. *Avoid an Excessive DOM Size*. https://developer.chrome.com/docs/lighthouse/performance/dom-size/. 2024.

[203] Google Developers. *Lighthouse*. en. Chrome for Developers. URL: https://developer.chrome.com/docs/lighthouse.

[204] Google Developers. *Removing cookie consent banners from your Lighthouse test*. URL: https://pagevitals.com/docs/guides/removing-cookie-consent-banners-from-your-lighthouse-tests/.

[205] Google Developers. *Serve images in modern formats | Lighthouse*. en. URL: https://developer.chrome.com/docs/lighthouse/performance/uses-webp-images.

[206] Gotze, M., Matic, S., Iordanou, C., Smaragdakis, G., and Laoutaris, N. Measuring web cookies in governmental websites. In: *Proceedings of the 14th ACM Web Science Conference 2022*. 2022, 44–54.

[207] Gousios, G., Pinzger, M., and Deursen, A. v. An exploratory study of the pull-based software development model. In: *Proceedings of the 36th international conference on software engineering*. 2014, 345–355.

[208] Goyvaerts, J. *Preventing Regular Expression Denial of Service (ReDoS)*. 2019. URL: https://www.regular-expressions.info/redos.html.

[209] Goyvaerts, J. *Runaway Regular Expressions: Catastrophic Backtracking*. 2021. URL: http://www.regular-expressions.info/catastrophic.html.

[210] Graham, S. L., Kessler, P. B., and McKusick, M. K. Gprof: a call graph execution profiler. *ACM Sigplan Notices* 17, 6 (1982), 120–126.

[211] Graham-Cumming, J. *Details of the Cloudflare outage on July 2, 2019*. July 2019. URL: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019.

[212] Group, P. *Php/Php-Src*. Aug. 2024. URL: https://github.com/php/php-src.

[213] Group, P. D. *PHP: General Information*. 2024. URL: https://www.php.net/manual/en/faq.general.php.

[214] Group, P. D. *PHP: History of PHP and Related Projects*. https://www.php.net/manual/en/history.php. 2024.

[215] Gruber, J. *Speeding up V8 Regular Expressions \cdot V8*. Jan. 2017. URL: https://v8.dev/blog/speeding-up-regular-expressions.

[216] GSMA. *The State of Mobile Internet Connectivity Report 2020*. Tech. rep. GSMA, 2020. URL: https://www.gsma.com/r/somic/.

[217] GSMA. *The Mobile Economy*. en-US. URL: https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/.

[218] GSMA. *The State of Mobile Internet Connectivity Report 2023 - Mobile for Development*. en-US. URL: https://www.gsma.com/r/somic/.

[219] Guinness, D., Cutrell, E., and Morris, M. R. Caption crawler: Enabling reusable alternative text descriptions using reverse image search. en. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18*. New York, NY, USA, 2018, 1–11.

[220] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. *Graphcodebert: Pre-training code representations with data flow*. 2020.

[221] Guo, P. *Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities — Communications of the ACM*. https://cacm.acm.org/blogcacm/python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/. 2014.

[222] Guo, P. J. Non-native english speakers learning computer programming: barriers, desires, and design opportunities. In: *Proceedings of the 2018 CHI conference on human factors in computing systems*. 2018, 1–14.

[223] Gutierrez, D. *Regular Expression Performance*. Nov. 2004. URL: https://learn.microsoft.com/en-us/archive/blogs/bclteam/regular-expression-performance-david-gutierrez.

[224] Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., and Mesbah, A. Bugsjs: a benchmark of javascript bugs. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, 90–101.

[225] Habib, R., Inam, A., Ali, A., Qazi, I. A., and Qazi, Z. A. A First Look at Public Service Websites from the Affordability Lens. en. In: *Proceedings of the ACM Web Conference 2023*. ACM, Austin TX USA, Apr. 2023, 2731–2741.

[226] Habib, R., Tanveer, S., Inam, A., Ahmed, H., Ali, A., Uzmi, Z. A., Qazi, Z. A., and Qazi, I. A. A framework for improving web affordability and inclusiveness. In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. Association for Computing Machinery, New York, NY, USA, 2023, 592–607.

[227] Hanley, M., Barocas, S., Levy, K., Azenkot, S., and Nissenbaum, H. Computer vision and conflicting values: describing people with automated alt text. In: *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*. 2021, 543–554.

[228] Haroon, S., Khan, A. F., Humayun, A., Gill, W., Amjad, A. H., Butt, A. R., Khan, M. T., and Gulzar, M. A. How accurately do large language models understand code? *arXiv preprint arXiv:2504.04372* (2025).

[229] Hasan, K. A., Macedo, M., Tian, Y., Adams, B., and Ding, S. Understanding the time to first response in github pull requests. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE. 2023, 1–11.

[230] Hasan, S., Barela, M. C., Johnson, M., Brewer, E., and Heimerl, K. Scaling community cellular networks with community cellular manager. In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI'19. USENIX Association, Boston, MA, USA, 2019, 735–750.

[231] Hasan, S., Ben-David, Y., Scott, C., Brewer, E., and Shenker, S. Enhancing rural connectivity with software defined networks. In: *Proceedings of the 3rd ACM Symposium on Computing for Development*. ACM DEV '13. Association for Computing Machinery, Bangalore, India, 2013.

[232] Hasan, S., Heimerl, K., Harrison, K., Ali, K., Roberts, S., Sahai, A., and Brewer, E. Gsm whitespaces: an opportunity for rural cellular service. In: *2014 IEEE International Symposium on Dynamic Spectrum Access Networks (DYSPAN)*. 2014, 271–282.

[233] Hasegawa, A. A., Yamashita, N., Akiyama, M., and Mori, T. Why they ignore english emails: the challenges of {non-native} speakers in identifying phishing emails. In: *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. 2021, 319–338.

[234] Hassan, S. A., Aamir, Z., Lee, D., Davis, J. C., and Servant, F. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2023, 1238–1255.

[235] Hazel, P. *PCRE - Perl Compatible Regular Expressions*. 1997. URL: `https://www.pcre.org/`.

[236] Hazimeh, A., Herrera, A., and Payer, M. Magma: a ground-truth fuzzing benchmark. In: *Abstract Proceedings of the 2021 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*. 2021, 81–82.

[237] He, H., Yang, H., Burckhardt, P., Kapravelos, A., Vasilescu, B., and Kästner, C. 4.5 million (suspected) fake stars in github: a growing spiral of popularity contests, scams, and malware. *arXiv preprint arXiv:2412.13459* (2024).

[238] Heeks, R. Digital inequality beyond the digital divide: conceptualizing adverse digital incorporation in the global south. *Information Technology for Development* 28, 4 (2022), 688–704.

[239] Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. Deep learning type inference. In: *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2018, 152–162.

[240] Hellman, J., Chen, J., Uddin, M. S., Cheng, J., and Guo, J. L. Characterizing user behaviors in open-source software user forums: an empirical study. In: *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*. 2022, 46–55.

[241] Helsper, E. The digital disconnect: the social causes and consequences of digital inequalities (2021).

[242] Henning, J. L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[243] Herzig, K. and Zeller, A. The impact of tangled code changes. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, 121–130.

[244] Hilbert, M. The bad news is that the digital access divide is here to stay: domestically installed bandwidths among 172 countries for 1986–2014. *Telecommunications Policy* 40, 6 (2016), 567–581.

[245] Hindle, A., German, D. M., and Holt, R. What do large commits tell us? a taxonomical study of large commits. In: *Proceedings of the 2008 international working conference on Mining software repositories*. 2008, 99–108.

[246] Holík, L., Síč, J., Turoňová, L., and Vojnar, T. Fast Matching of Regular Patterns with Synchronizing Counting. In: *Foundations of Software Science and Computation Structures*. Ed. by Kupferman, O. and Sobocinski, P. Springer Nature Switzerland, Cham, 2023, 392–412.

[247] Hollington, A., Salverda, T., Schwarz, T., and Tappe, O. Concepts of the global south (2015).

[248] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson/Addison Wesley, Boston, 2007.

[249] Hotspot Shield. *Hotspot Shield: Fastest VPN for Streaming, Gaming & More — hotspotshield.com*. https://www.hotspotshield.com/.

[250] Hyrynsalmi, S. M., Baltes, S., Brown, C., Prikladnicki, R., Rodriguez-Perez, G., Serebrenik, A., Simmonds, J., Trinkenreich, B., Wang, Y., and Liebel, G. Bridging gaps, building futures: advancing software developer diversity and inclusion through future-oriented research. *arXiv preprint arXiv:2404.07142* (2024).

[251] Imambi, S., Prakash, K. B., and Kanagachidambaresan, G. Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications* (2021), 87–104.

[252] IMF. *World Economic Outlook Database - Groups and Aggregates*. ENG. URL: https://www.imf.org/en/Publications/WEO/weo-database/2023/April/groups-and-aggregates.

[253] Index, S. G. *Speedtest Global Index – Internet Speed around the world*. en. URL: https://www.speedtest.net/global-index%5C#mobile.

[254] India, T. T. of. *Bengalureans Face Name Mismatch Issues in New E-Khata System | Bengaluru News - The Times of India — timesofindia.indiatimes.com*. https://timesofindia.indiatimes.com/city/bengaluru/bengalureans-face-name-mismatch-issues-in-new-e-khata-system/articleshow/115308462.cms.

[255] Initiative (WAI), W. W. A. *Web Content Accessibility Guidelines (WCAG) 2.1*. URL: https://www.w3.org/TR/WCAG21/.

[256] Initiative (WAI), W. W. A. *What's New in WCAG 2.2*. en. URL: https://www.w3.org/WAI/standards-guidelines/wcag/new-in-22/.

[257] International Telecommunication Union. *Individuals using the Internet - ITU DataHub*. URL: https://www.itu.int/itu-d/reports/statistics/2025/10/15/ff25-internet-use.

[258] Interoperable Europe. *Open Source Software Country Intelligence Report South Korea*. Tech. rep. European Commission, 2022. URL: https://interoperable-europe.ec.europa.eu/sites/default/files/inline-files/OSS%20Country%20Intelligence%20report_KR.pdf.

[259] Ismailova, R. Web site accessibility, usability and security: a survey of government web sites in kyrgyz republic. *Universal Access in the Information Society* 16, 1 (2017), 257–264.

[260] Ismailova, R. and Kimsanova, G. Universities of the kyrgyz republic on the web: accessibility and usability. *Universal Access in the Information Society* 16, 4 (2017), 1017–1025.

[261] Jenkinson, T. *Tjenkinson/redos-detector*. May 2024. URL: https://github.com/tjenkinson/redos-detector.

[262] JetBrains. *IntelliJ File Encoding Issue — intellij-support.jetbrains.com*. https://intellij-support.jetbrains.com/hc/en-us/community/posts/360003953579-IntelliJ-file-encoding-issue. 2018.

[263] JetBrains. *Java Programming — The State of Developer Ecosystem in 2021 Infographic*. https://www.jetbrains.com/lp/devecosystem-2021/java/. 2021.

[264] Jose, A. and Aishwarya. *Aadhaar fails the women who need it most - The Tribune — tribuneindia.com*. https://www.tribuneindia.com/news/comment/aadhaar-fails-the-women-who-need-it-most/.

[265] Jueckstock, J., Sarker, S., Snyder, P., Beggs, A., Papadopoulos, P., Varvello, M., Livshits, B., and Kapravelos, A. Towards realistic and reproducible web crawl measurements. In: *Proceedings of the Web Conference 2021*. 2021, 80–91.

[266] Just, R., Jalali, D., and Ernst, M. D. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 international symposium on software testing and analysis*. 2014, 437–440.

[267] Karampatsis, R.-M. and Sutton, C. Scelmo: source code embeddings from language models. *arXiv preprint arXiv:2004.13214* (2020).

[268] Karim, R., Tip, F., Sochůrková, A., and Sen, K. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1364–1379.

[269] Katzy, J., Huang, Y., Panchu, G.-R., Ziemlewski, M., Loizides, P., Vermeulen, S., Deursen, A. van, and Izadi, M. A qualitative investigation into llm-generated multilingual code comments and automatic evaluation metrics (2025).

[270] Kavaler, D., Sirovica, S., Hellendoorn, V., Aranovich, R., and Filkov, V. Perceived language complexity in github issue discussions and their effect on issue resolution. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, 72–83.

[271] Kemp, S. *Digital 2026: Six Billion Internet Users*. https://datareportal.com/reports/digital-2026-six-billion-internet-users. 2026.

[272] Khodayari, S. and Pellegrino, G. {Jaw}: studying client-side {csrf} with hybrid property graphs and declarative traversals. In: *30th usenix security symposium (usenix security 21)*. 2021, 2525–2542.

[273] Kirrage, J., Rathnayake, A., and Thielecke, H. Static Analysis for Regular Expression Denial-of-Service Attacks. In: *Network and System Security*. Ed. by Lopez, J., Huang, X., and Sandhu, R. Springer, Berlin, Heidelberg, 2013, 135–148.

[274] Kleene, S. C. Representation of Events in Nerve Nets and Finite Automata. In: *Automata Studies*. Ed. by Shannon, C. E. and McCarthy, J. Princeton University Press, Princeton, 1956, 3–42.

[275] Kluban, M., Mannan, M., and Youssef, A. On measuring vulnerable javascript functions in the wild. In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '22. Association for Computing Machinery, Nagasaki, Japan, 2022, 917–930.

[276] Kluban, M., Mannan, M., and Youssef, A. On detecting and measuring exploitable javascript functions in real-world applications. *ACM Transactions on Privacy and Security* 27, 1 (Feb. 2024).

[277] Koch, S., Klein, D., and Johns, M. The fault in our stars: an analysis of github stars as an importance metric for web source code. In: *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. Vol. 2024. 2024.

[278] Koehler, W. Web page change and persistence - A four-year longitudinal study. *J. Assoc. Inf. Sci. Technol.* 53, 2 (2002), 162–171.

[279] Koishybayev, I. and Kapravelos, A. Mininode: reducing the attack surface of node. js applications. In: *23rd international symposium on research in attacks, intrusions and defenses (RAID 2020)*. 2020, 121–134.

[280] Kondo, M., German, D. M., Kamei, Y., Ubayashi, N., and Mizuno, O. An empirical study of token-based micro commits. *Empirical Software Engineering* 29, 6 (2024), 148.

[281] Koradia, Z., Mannava, G., Raman, A., Aggarwal, G., Ribeiro, V., Seth, A., Ardon, S., Mahanti, A., and Triukose, S. First impressions on the state of cellular data connectivity in India. en. In: *Proceedings of the 4th Annual Symposium on Computing for Development*. ACM, Cape Town South Africa, Dec. 2013, 1–10.

[282] Kosako, K. *Kkos/Oniguruma*. June 2024. URL: `https://github.com/kkos/oniguruma`.

[283] Kostić, M., Batanović, V., and Nikolić, B. Monolingual, multilingual and cross-lingual code comment classification. *Engineering Applications of Artificial Intelligence* 124 (2023), 106485.

[284] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.

[285] Kumar, S. *Understanding WCAG SC 1.1.1 Non-text Content • DigitalA11Y — digitala11y.com*. https://www.digitala11y.com/understanding-sc-1-1-1-non-text-content/.

[286] Kupoluyi, J., Chaqfeh, M., Varvello, M., Coke, R., Hashmi, W., Subramanian, L., and Zaki, Y. Muzeel: assessing the impact of javascript dead code elimination on mobile web performance. In: *Proceedings of the 22nd ACM Internet Measurement Conference*. IMC '22. Association for Computing Machinery, Nice, France, 2022, 335–348.

[287] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou shalt not depend on me: analysing the use of outdated javascript libraries on the web. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[288] Lazar, J., Goldstein, D. F., and Taylor, A. *Ensuring Digital Accessibility Through Process and Policy*. Morgan Kaufmann, 2015.

[289] LeClair, A., Jiang, S., and McMillan, C. A neural model for generating natural language summaries of program subroutines. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, 795–806.

[290] Lehman, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* 68, 9 (2005), 1060–1076.

[291] Lewis, P. *Optimize JavaScript Execution*. https://web.dev/articles/optimize-javascript-execution. 2015.

[292] Lewthwaite, S. Web accessibility standards and disability: developing critical perspectives on accessibility. *Disability and Rehabilitation* 36, 16 (2014), 1375–1383.

[293] Li, J., Wang, Y., Lyu, M. R., and King, I. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[294] Li, S., Kang, M., Hou, J., and Cao, Y. Detecting node. js prototype pollution vulnerabilities via object lookup analysis. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, 268–279.

[295] Li, S., Kang, M., Hou, J., and Cao, Y. Mining node. js vulnerabilities via object dependence graph and query. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, 143–160.

[296] Li, Y., Chen, Z., Cao, J., Xu, Z., Peng, Q., Chen, H., Chen, L., and Cheung, S.-C. ReDoSHunter: a combined static and dynamic approach for regular expression DoS detection. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, 3847–3864.

[297] Li, Y., Sun, Y., Xu, Z., Cao, J., Li, Y., Li, R., Chen, H., Cheung, S.-C., Liu, Y., and Xiao, Y. RegexScalpel: regular expression denial of service (ReDoS) defense by Localize-and-Fix. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, Aug. 2022, 4183–4200.

[298] Li, Y., Xu, Z., Cao, J., Chen, H., Ge, T., Cheung, S.-C., and Zhao, H. Flashregex: deducing anti-redos regexes from examples. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Association for Computing Machinery, Virtual Event, Australia, 2021, 659–671.

[299] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker: a deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[300] Lim, K., Kwon, Y., and Kim, D. A longitudinal study of vulnerable client-side resources and web developers' updating behaviors. In: *Proceedings of the 2023 ACM on Internet Measurement Conference, IMC 2023, Montreal, QC, Canada, October 24-26, 2023*. Ed. by Montpetit, M., Leivadeas, A., Uhlig, S., and Javed, M. ACM, 2023, 162–180.

[301] Limited, S. *ReDoS vulnerabilities in npm spikes by 143% and XSS continues to grow*. Feb. 2019. URL: https://snyk.io/blog/redos-vulnerabilities-in-npm-spikes-by-143-and-xss-continues-to-grow/.

[302] Limited, S. *State of Open Source Security*. 2023. URL: https://snyk.io/reports/open-source-security/.

[303] Limited, S. *ReDoS Tutorials & Examples*. 2024. URL: https://learn.snyk.io/lesson/redos/.

[304] Lin, Z., Marinov, D., Zhong, H., Chen, Y., and Zhao, J. Jacontebe: a benchmark suite of real-world java concurrency bugs (t). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, 178–189.

[305] Liu, B., Zhao, Y., Xu, G., and Wang, H. Llm agents for automated web vulnerability reproduction: are we there yet? *arXiv preprint arXiv:2510.14700* (2025).

[306] Liu, J., Zeng, J., Wang, X., and Liang, Z. Learning graph-based code representations for source-level functional similarity detection. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, 345–357.

[307] Liu, Y., Zhang, M., and Meng, W. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021, 1468–1484.

[308] LLC., D. *Doyensec/regexploit*. Aug. 2024. URL: https://github.com/doyensec/regexploit.

[309] Lou, Y., Zhu, Q., Dong, J., Li, X., Sun, Z., Hao, D., Zhang, L., and Zhang, L. Boosting coverage-based fault localization via graph-based representation learning. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, 664–676.

[310] Ludwig, J. *Complex regular subexpression recursion limit*. Dec. 2009. URL: https://www.perlmonks.org/?node_id=810857.

[311] Lythreatis, S., Singh, S. K., and El-Kassar, A.-N. The digital divide: a review and future research agenda. *Technological Forecasting and Social Change* 175 (2022), 121359.

[312] Maddison, C. and Tarlow, D. Structured generative models of natural source code. In: *International Conference on Machine Learning*. PMLR. 2014, 649–657.

[313] Mahoney, M. S. The histories of computing (s). *Interdisciplinary science reviews* 30, 2 (2005), 119–135.

[314] Mamouras, K. and Chattopadhyay, A. Efficient matching of regular expressions with lookaround assertions. *Proc. ACM Program. Lang.* 8, POPL (Jan. 2024).

[315] Mangal, R., Zhang, X., Nori, A. V., and Naik, M. A user-guided approach to program analysis. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, 462–473.

[316] Mantas, G., Stakhanova, N., Gonzalez, H., Jazi, H. H., and Ghorbani, A. A. Application-layer denial of service attacks: taxonomy and survey. *International Journal of Information and Computer Security* 7, 2/3/4 (Nov. 2015), 216–239.

[317] Mashau, P. and Farisani, T. R. *Accessibility of Digital Higher Education in the Global South*. IGI Global, 2023.

[318] Mashhadi, E. and Hemmati, H. Applying codebert for automated program repair of java simple bugs. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, 505–509.

[319] Mayer, J. R. and Mitchell, J. C. Third-party web tracking: policy and technology. In: *2012 IEEE symposium on security and privacy*. IEEE. 2012, 413–427.

[320] McCarthy, T., Pal, J., Marballi, T., and Cutrell, E. An analysis of screen reader use in India. In: *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development*. 2012, 149–158.

[321] McLaughlin, R., Pagani, F., Spahn, N., Kruegel, C., and Vigna, G. Regulator: dynamic analysis to detect ReDoS. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, Aug. 2022, 4219–4235.

[322] McNaughton, R. and Yamada, H. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers* EC-9, 1 (Mar. 1960), 39–47.

[323] Meng, W., Qian, C., Hao, S., Borgolte, K., Vigna, G., Kruegel, C., and Lee, W. Rampart: protecting web applications from CPU-Exhaustion Denial-of-Service attacks. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, Aug. 2018, 393–410.

[324] Meta Platforms, Inc. *Facebook Lite APK for Android*. URL: `https://www.facebook.com/lite/`.

[325] Microsoft. *C# Dev Kit FAQ — code.visualstudio.com*. `https://code.visualstudio.com/docs/csharp/cs-dev-kit-faq#_who-can-use-chash-dev-kit`. 2023.

[326] Microsoft. *What Is .NET?* 2024. URL: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet.

[327] Miller, C., Cohen, S., Klug, D., Vasilescu, B., and KaUstner, C. " did you miss my comment or what?" understanding toxicity in open source discussions. In: *Proceedings of the 44th international conference on software engineering*. 2022, 710–722.

[328] Ming'ate, F. L. M. The global south: what does it mean to kenya. *Concepts of the Global South: Voices from around the world* (2015), 8.

[329] Ministry of Human Rights, Pakistan. *Ministry of Human Rights*. URL: https://web.archive.org/web/20241024185918/https://mohr.gov.pk/NewsDetail/NDA2NDc4OWUtYjY2NC00M2MzLTg0MTktZjY1YzZiNzc0NWQw.

[330] Mir, A. M., Keshani, M., and Proksch, S. On the effectiveness of machine learning-based call graph pruning: an empirical study. In: *Proceedings of the 21st International Conference on Mining Software Repositories*. 2024, 457–468.

[331] Mitra, J. and Ranganath, V.-P. Ghera: a repository of android app vulnerability benchmarks. In: *Proceedings of the 13th international conference on predictive models and data analytics in software engineering*. 2017, 43–52.

[332] MITRE. *CWE-1333: Inefficient Regular Expression Complexity*. 2021. URL: https://cwe.mitre.org/data/definitions/1333.html.

[333] Mockus, A., Fielding, R. T., and Herbsleb, J. D. Two case studies of open source software development: apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (2002), 309–346.

[334] Moon, Y., Lee, H., Oh, S., and Jung, H. Sagol: using minigpt-4 to generate alt text for improving image accessibility. In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*. 2024, 8745–8748.

[335] Moseley, D., Nishio, M., Perez Rodriguez, J., Saarikivi, O., Toub, S., Veanes, M., Wan, T., and Xu, E. Derivative based nonbacktracking real-world regex matching with backtracking semantics. *Proc. ACM Program. Lang.* 7, PLDI (June 2023).

[336] Mozilla. *Mobile accessibility - Learn web development | MDN*. en-US. Aug. 2024. URL: https://developer.mozilla.org/en-US/docs/Learn/Accessibility/Mobile.

[337] Mozilla. *Accessibility tree*. https://developer.mozilla.org/en-US/docs/Glossary/Accessibility_tree.

[338] Mozilla. *alt*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img#alt.

[339] Mozilla. *ARIA - Accessibility | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA.

[340] Mozilla. *Screen reader*. https://developer.mozilla.org/en-US/docs/Glossary/Screen_reader.

[341] Mozilla. *Third-party cookies*. https://developer.mozilla.org/en-US/docs/Web/Privacy/Guides/Third-party_cookies.

[342] Mtebe, J. S. and Kondoro, A. Accessibility and usability of government websites in tanzania. *African Journal of Information Systems* 9, 4 (2017), 3.

[343] Muff, A. M. From Desktop to Mobile Device Who Is Left Behind? Moving Toward Global Accessibility Standards. en (2017).

[344] Murgia, A., Ortu, M., Tourani, P., Adams, B., and Demeyer, S. An exploratory qualitative and quantitative analysis of emotions in issue report comments of open source systems. *Empirical Software Engineering* 23, 1 (2018), 521–564.

[345] Murphy, E., Crick, T., and Davenport, J. H. An analysis of introductory programming courses at uk universities. *arXiv preprint arXiv:1609.06622* (2016).

[346] N-iX. *Where to Hire Java Developers: UK vs Ukraine vs India — N-iX*. `https://www.n-ix.com/where-hire-java-developers-uk-ukraine-india/`. 2023.

[347] Naruse, Y. *Ruby 3.2.0 Released*. `https://www.ruby-lang.org/en/news/2022/12/25/ruby-3-2-0-released/`. 2022.

[348] Naseer, U. and Benson, T. A. Js capsules: a framework for capturing fine-grained javascript memory measurements for the mobile web. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1 (Mar. 2023).

[349] Naseer, U., Benson, T. A., and Netravali, R. Webmedic: disentangling the memory-functionality tension for the next billion mobile web users. In: *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. HotMobile '21. Association for Computing Machinery, Virtual, United Kingdom, 2021, 71–77.

[350] New Relic, I. *State of the Java Ecosystem Report*. 2024. URL: `https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem`.

[351] Ngowi, L. and Mvungi, N. H. Socio-technical systems: transforming theory into practice. *International Journal of Industrial and Systems Engineering* 12, 2 (2018), 310–316.

[352] Nguyen, T.-D., Le-Cong, T., Luong, D.-M., Duong, V.-H., Le, X.-B. D., Lo, D., and Huynh, Q.-T. Ffl: fine-grained fault localization for student programs via syntactic and semantic reasoning. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2022, 151–162.

[353] Nguyen, T. G., Le-Cong, T., Kang, H. J., Le, X.-B. D., and Lo, D. Vulcurator: a vulnerability-fixing commit detector. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, 1726–1730.

[354] Nielsen, B. B., Hassanshahi, B., and Gauthier, F. Nodest: feedback-driven static analysis of node. js applications. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, 455–465.

[355] Nielsen, B. B., Torp, M. T., and Møller, A. Modular call graph construction for security scanning of node. js applications. In: *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 2021, 29–41.

[356] Noll, J., Beecham, S., and Richardson, I. Global software development and collaboration: barriers and solutions. *ACM inroads* 1, 3 (2011), 66–78.

[357] Noller, Y., Kersten, R., and Păsăreanu, C. S. Badger: complexity analysis with fuzzing and symbolic execution. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Association for Computing Machinery, Amsterdam, Netherlands, 2018, 322–332.

[358] npm, Inc. *npm rank*. 2015. URL: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491.

[359] Nso-Mangue, P., Acosta, T., and Luján-Mora, S. Web accessibility of top-10 universities of africa, america, asia, europe, and oceania: 2023 snapshot. In: *INTED2024 Proceedings*. IATED. 2024, 5059–5068.

[360] Nsrav. *Denial of Service*. 2024. URL: https://owasp.org/www-community/attacks/Denial_of_Service.

[361] NV Access. *NV Access — nvaccess.org*. https://www.nvaccess.org/.

[362] NV Access. *NVDA User Guide: Browse Mode*. https://download.nvaccess.org/documentation/userGuide.html#BrowseMode.

[363] OECD. *Bridging the Digital Gender Divide*. Tech. rep. Organisation for Economic Co-operation and Development, 2018. URL: https://www.oecd.org/digital/bridging-the-digital-gender-divide.pdf.

[364] Ohira, M., Kashiwa, Y., Yamatani, Y., Yoshiyuki, H., Maeda, Y., Limsettho, N., Fujino, K., Hata, H., Ihara, A., and Matsumoto, K. A dataset of high impact bugs: manually-classified issue reports. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, 518–521.

[365] Olivo, O., Dillig, I., and Lin, C. Static detection of asymptotic performance bugs in collection traversals. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Association for Computing Machinery, Portland, OR, USA, 2015, 369–378.

[366] OpenAI. *ChatGPT — chatgpt.com*. https://chatgpt.com/. 2022.

[367] OpenJDK. *JDK-8242257: Encoding Issue in Windows Terminal — bugs.openjdk.org*. https://bugs.openjdk.org/browse/JDK-8242257?page=com.atlassian.jira.plugin.system.issuetabpanels%3Aall-tabpanel. 2020.

[368] Oracle. *javadoc — docs.oracle.com*. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html. 2015.

[369] Oracle Corporation. *OpenJDK*. 2024. URL: https://openjdk.org/.

[370] Oracle Corporation. *Pattern (Java Platform SE 8)*. 2024. URL: https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

[371] Ortu, M., Destefanis, G., Kassab, M., Counsell, S., Marchesi, M., and Tonelli, R. Would you mind fixing this issue? an empirical analysis of politeness and attractiveness in software developed using agile boards. In: *International conference on Agile software development*. Springer. 2015, 129–140.

[372] Osmani, A. and Bynens, M. *The cost of JavaScript in 2019 · V8*. URL: https://v8.dev/blog/cost-of-javascript-2019.

[373] Pal, J., Chandra, P., O'neill, T., Youngman, M., Jones, J., Song, J. H., Strayer, W., and Ferrari, L. An accessibility infrastructure for the global south. In: *Proceedings of the Eighth International Conference on Information and Communication Technologies and Development*. 2016, 1–11.

[374] Pareddy, S., Guo, A., and Bigham, J. P. X-ray: Screenshot accessibility via embedded metadata. en. In: *Proceedings of the 21st International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS '19*. New York, NY, USA, 2019, 389–395.

[375] Park, S., Jin, H., Cha, J.-w., and Han, Y.-S. Detection of llm-paraphrased code and identification of the responsible llm using coding style features. *arXiv preprint arXiv:2502.17749* (2025).

[376] Park, S.-Y., Glidden, L. M., and Shin, J. Y. Structural and Functional Aspects of Social Support for Mothers of Children with and without Cognitive Delays in Vietnam. en. *Journal of Applied Research in Intellectual Disabilities* 23, 1 (2010), 38–51.

[377] Parolini, F. and Miné, A. Sound static analysis of regular expressions for vulnerabilities to denial of service attacks. *Science of Computer Programming* 229 (July 2023).

[378] Pashchenko, I., Vu, D.-L., and Massacci, F. A qualitative study of dependency management and its security implications. In: *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2020, 1513–1531.

[379] Paul, S. Accessibility analysis using wcag 2.1: evidence from indian e-government websites. *Universal Access in the Information Society* 22, 3 (2023), 663–669.

[380] Pawelka, T. and Juergens, E. Is this code written in english? a study of the natural language of comments and identifiers in practice. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, 401–410.

[381] Peng, Y., Gao, C., Li, Z., Gao, B., Lo, D., Zhang, Q., and Lyu, M. Static inference meets deep learning: a hybrid type inference approach for python. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, 2019–2030.

[382] Penman, R. *python-whois: Whois querying and parsing of domain registration information.* URL: https://github.com/richardpenman/whois.

[383] Perl.org. *About Perl*. 2024. URL: https://www.perl.org/about.html.

[384] Perl.org. *Perl/Perl5*. Aug. 2024. URL: https://github.com/perl/perl5.

[385] Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. Slowfuzz: automated domain-independent detection of algorithmic complexity vulnerabilities. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Association for Computing Machinery, Dallas, Texas, USA, 2017, 2155–2168.

[386] PHP Documentation Group. *PHP: Runtime Configuration - Manual*. https://www.php.net/manual/en/pcre.configuration.php. 2024.

[387] Pimienta, D. Resource: indicators on the presence of languages in internet. In: *Proceedings of the 1st Annual Meeting of the ELRA/ISCA Special Interest Group on Under-Resourced Languages*. 2022, 83–91.

[388] Pimienta, D. Reliably exploring the presence of languages on the internet. *Research Outreach* (2024).

[389] Pimienta, D., Blanco, Á., and Oliveira, G. M. de. The method behind the unprecedented production of indicators of the presence of languages in the internet. *Frontiers in Research Metrics and Analytics* 8 (2023), 1149347.

[390] Pochat, V. L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: a research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156* (2018).

[391] Poess, M. and Floyd, C. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.

[392] Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., and Dangremont, C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, 383–387.

[393] Pradel, M., Gousios, G., Liu, J., and Chandra, S. Typewriter: neural type prediction with search-based validation. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, 209–220.

[394] Pressman, R. S. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[395] Prikladnicki, R., Nicolas Audy, J. L., and Evaristo, R. Global software development in practice lessons learned. *Software Process: Improvement and Practice* 8, 4 (2003), 267–281.

[396] Project, V. *V8 JavaScript Engine*. 2024. URL: https://v8.dev/.

[397] Project, V. *V8/v8*. Aug. 2024. URL: https://chromium.googlesource.com/v8/v8.git.

[398] Proton AG. *The best VPN for speed and security — protonvpn.com*. https://protonvpn.com/.

[399] Puppeteer. *Puppeteer | Puppeteer*. en. URL: https://pptr.dev/.

[400] Python Software Foundation. *PEP 257 – Docstring Conventions | peps.python.org*. https://peps.python.org/pep-0257/. 2001.

[401] Q-Success. *Distribution of Web Servers among Websites That Use JavaScript*. 2024. URL: https://w3techs.com/technologies/segmentation/pl-js/web_server.

[402] Qazi, I. A., Qazi, Z. A., Ali, A., Abdullah, M., and Habib, R. Rethinking Web for Affordability and Inclusion. en. In: *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. ACM, Virtual Event United Kingdom, Nov. 2021, 9–15.

[403] Rabin, M. O. and Scott, D. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, 2 (Apr. 1959), 114–125.

[404] Rack, J. and Staicu, C.-A. Jack-in-the-box: an empirical study of javascript bundling on the web and its security implications. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, 3198–3212.

[405] Radu, A. and Nadi, S. A dataset of non-functional bugs. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, 399–403.

[406] Raghavan, T. S. *Govt offices just can't handle my South Indian name. A saga of dots and initials — theprint.in*. https://theprint.in/opinion/pov/govt-offices-just-cant-handle-my-south-indian-name-a-saga-of-dots-and-initials/2331597/.

[407] Raghavendra, E. V. and Prahallad, K. A multilingual screen reader in indian languages. In: *2010 National Conference On Communications (NCC)*. IEEE. 2010, 1–5.

[408] Rathnayake, A. and Thielecke, H. Static analysis for regular expression exponential runtime via substructural logics (extended). *arXiv preprint arXiv:1405.7058* (2017).

[409] Rauschmayer, A. *26. Unicode in ES6 — exploringjs.com*. https://exploringjs.com/es6/ch_unicode.html. 2016.

[410] Rautenstrauch, J., Mitkov, M., Helbrecht, T., Hetterich, L., and Stock, B. To auth or not to auth? a comparative analysis of the pre-and post-login security landscape. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, 1500–1516.

[411] Ray, B., Posnett, D., Filkov, V., and Devanbu, P. A large scale study of programming languages and code quality in github. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014, 155–165.

[412] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In: *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 2014, 419–428.

[413] Raza, A., Zaki, Y., Pötsch, T., Chen, J., and Subramanian, L. Xcache: rethinking edge caching for developing regions. In: *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*. ICTD '17. Association for Computing Machinery, Lahore, Pakistan, 2017.

[414] Reestman, K. and Dorn, B. Native language's effect on java compiler errors. In: *Proceedings of the 2019 ACM conference on international computing education research*. 2019, 249–257.

[415] Rescorla, E. *HTTP Over TLS*. 2000. URL: https://www.rfc-editor.org/rfc/rfc2818.html.

[416] Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3*. https://www.rfc-editor.org/rfc/rfc8446. 2018.

[417] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[418] Robie, J. *What is the Document Object Model?* https://www.w3.org/TR/WD-DOM/introduction.html.

[419] Rodríguez Vázquez, S. Introducing web accessibility to localization students: implications for a universal web. In: *Proceedings of the 16th international ACM SIGACCESS conference on Computers & accessibility*. 2014, 333–334.

[420] Rodríguez Vázquez, S. Exploring Current Accessibility Challenges in the Multilingual Web for Visually-Impaired Users. en. In: *Proceedings of the 24th International Conference on World Wide Web*. ACM, Florence Italy, May 2015, 871–873.

[421] Rodríguez Vázquez, S. Assuring accessibility during web localisation: an empirical investigation on the achievement of appropriate text alternatives for images. en. PhD thesis. Universidad de Salamanca, 2016.

[422] Rodríguez Vázquez, S. Measuring the impact of automated evaluation tools on alternative text quality: a web translation study. en. In: *Proceedings of the 13th International Web for All Conference*. ACM, Montreal Canada, Apr. 2016, 1–10.

[423] Rodríguez Vázquez, S. and O'Brien, S. Bringing accessibility into the multilingual web production chain: perceptions from the localization industry. In: *International Conference on Universal Access in Human-Computer Interaction*. Springer. 2017, 238–257.

[424] Roesch, M. Snort - lightweight intrusion detection for networks. In: *Proceedings of the 13th USENIX Conference on System Administration*. LISA '99. USENIX Association, Seattle, Washington, 1999, 229–238.

[425] Roesner, F., Kohno, T., and Wetherall, D. Detecting and defending against {third-party} tracking on the web. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, 155–168.

[426] Roichman, A. and Weidman, A. VAC-ReDoS: Regular Expression Denial Of Service. *Open Web Application Security Project (OWASP)* (2009).

[427] Roppelt, T. *Der Accessibility Tree einfach erklärt*. https://gehirngerecht.digital/accessibility-tree/. 2024.

[428] Roth, S., Gröber, L., Backes, M., Krombholz, K., and Stock, B. 12 angry developers - A qualitative study on developers' struggles with CSP. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 2021, 3085–3103.

[429] Royce, W. W. Managing the development of large software systems: concepts and techniques. In: *Proceedings of the 9th international conference on Software Engineering*. 1987, 328–338.

[430] Ruth, K., Fass, A., Azose, J., Pearson, M., Thomas, E., Sadowski, C., and Durumeric, Z. A world wide view of browsing the world wide web. In: *Proceedings of the 22nd ACM Internet Measurement Conference, IMC 2022, Nice, France, October 25-27, 2022*. Ed. by Barakat, C., Pelsser, C., Benson, T. A., and Choffnes, D. R. ACM, 2022, 317–336.

[431] Saarikivi, O., Veanes, M., Wan, T., and Xu, E. Symbolic Regex Matcher. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Vojnar, T. and Zhang, L. Springer International Publishing, Cham, 2019, 372–378.

[432] Saha, R. K., Lyu, Y., Lam, W., Yoshida, H., and Prasad, M. R. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In: *Proceedings of the 15th international conference on mining software repositories*. 2018, 10–13.

[433] Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., and Lopes, C. V. Oreo: detection of clones in the twilight zone. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, 354–365.

[434] Salesforce, I. *HTTP Routing*. en. 2024. URL: `https://devcenter.heroku.com/articles/http-routing`.

[435] Sankhi, P. and Sandnes, F. E. A glimpse into smartphone screen reader use among blind teenagers in rural nepal. *Disability and Rehabilitation: Assistive Technology* 17, 8 (2022), 875–881.

[436] Santhanam, S., Hecking, T., Schreiber, A., and Wagner, S. Bots in software engineering: a systematic mapping study. *PeerJ Computer Science* 8 (2022), e866.

[437] Santos, R., Prado, R., Holanda Silva, A. P. de, Gama, K., Castor, F., and Souza Santos, R. de. Understanding underrepresented groups in open source software. In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 2025, 919–928.

[438] Sarita, K., Kaur, P., and Kaur, S. Accessibility of healthcare sites: evaluation by automated tools. In: *Proceedings of International Conference on Data Science and Applications: ICDSA 2021, Volume 2*. Springer. 2022, 625–636.

[439] Sarker, J., Sultana, S., Wilson, S. R., and Bosu, A. Toxispanse: an explainable toxicity detection in code review comments. In: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2023, 1–12.

[440] Sarker, J., Turzo, A. K., and Bosu, A. The landscape of toxicity: an empirical investigation of toxicity on github. *Proc. ACM Softw. Eng.* 2, FSE (June 2025).

[441] Schloegel, M., Bars, N., Schiller, N., Bernhard, L., Scharnowski, T., Crump, A., Ale-Ebrahim, A., Bissantz, N., Muench, M., and Holz, T. Sok: prudent evaluation practices for fuzzing. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, 1974–1993.

[442] Selakovic, M. and Pradel, M. Performance issues and optimizations in javascript: an empirical study. In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, 61–72.

[443] Shan, H., Wang, Q., and Pu, C. Tail attacks on web applications. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Association for Computing Machinery, Dallas, Texas, USA, 2017, 1725–1739.

[444] Sharma, A., Kaur, M., Koradia, Z., Nishant, R., Pandit, S., Raman, A., and Seth, A. Revisiting the state of cellular data connectivity in india. In: *Proceedings of the 2015 Annual Symposium on Computing for Development*. DEV '15. Association for Computing Machinery, London, United Kingdom, 2015, 149–157.

[445] Shcherbakov, M., Balliu, M., and Staicu, C.-A. Silent spring: prototype pollution leads to remote code execution in node. js. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, 5521–5538.

[446] Shen, Y., Zhang, H., Shen, Y., Wang, L., Shi, C., Du, S., and Tao, Y. Altgen: ai-driven alt text generation for enhancing epub accessibility. *arXiv preprint arXiv:2501.00113* (2024).

[447] Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., and Lu, J. ReScue: crafting regular expression DoS attacks. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Sept. 2018, 225–235.

[448] Shepelev, V. *Ruby 3.2 Changes*. `https://rubyreferences.github.io/rubychanges/3.2.html#regexp-redos-vulnerability-prevention`. 2023.

[449] Siddiq, M. L., Zhang, J., Roney, L., and Santos, J. C. Re (gex| dos) eval: evaluating generated regular expressions and their proneness to dos attacks. In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 2024, 52–56.

[450] Siddiq, M. L., Zhang, J., and Santos, J. C. D. S. Understanding regular expression denial of service (redos): insights from llm-generated regexes and developer forums. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC '24. Association for Computing Machinery, Lisbon, Portugal, 2024, 190–201.

[451] Smith, R., Estan, C., and Jha, S. XFA: Faster Signature Matching with Extended Automata. In: *2008 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2008, 187–201.

[452] Snyder, P., Taylor, C., and Kanich, C. Most websites don't need to vibrate: a cost-benefit approach to improving browser security. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, 179–194.

[453] Sommerville, I. Software engineering (ed.) *America: Pearson Education Inc* (2011).

[454] Source, G. O. *Google/Re2*. Sept. 2024. URL: https://github.com/google/re2.

[455] Spencer, H. A Regular-Expression Matcher. In: *Software Solutions in C*. Academic Press Professional, Inc., USA, Apr. 1994, 35–71.

[456] Stack Overflow. *Can I Ask a Question in a Language Other Than English? — Help Center*. https://stackoverflow.com/help/non-english-questions. 2024.

[457] Stack Overflow. *Stack Overflow — survey.stackoverflow.co*. https://survey.stackoverflow.co/.

[458] Staff, G. *Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1 — github.blog*. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/#h-the-state-of-github-in-2025-a-year-of-record-growth.

[459] Staicu, C.-A. and Pradel, M. Freezing the web: a study of ReDoS vulnerabilities in JavaScript-based web servers. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, Aug. 2018, 361–376.

[460] Staicu, C.-A., Pradel, M., and Livshits, B. Synode: understanding and automatically preventing injection attacks on node. js. In: *NDSS*. Vol. 1. 2018, 12–52.

[461] Staicu, C.-A., Rahaman, S., Kiss, Á., and Backes, M. Bilingual problems: studying the security risks incurred by native extensions in scripting languages. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, 6133–6150.

[462] Staicu, C.-A., Torp, M. T., Schäfer, M., Møller, A., and Pradel, M. Extracting taint specifications for javascript libraries. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, 198–209.

[463] Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. The german traffic sign recognition benchmark: a multi-class classification competition. In: *The 2011 international joint conference on neural networks*. IEEE. 2011, 1453–1460.

[464] StatCounter. *Desktop vs Mobile vs Tablet Market Share in Asia*. en. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/asia.

[465] Status, S. E. N. *Outage Postmortem - July 20, 2016*. July 2016. URL: https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016.

[466] Steinmacher, I., Conte, T., Gerosa, M. A., and Redmiles, D. Social barriers faced by newcomers placing their first contribution in open source software projects. In: *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*. 2015, 1379–1392.

[467] Steinmacher, I., Gerosa, M., Conte, T. U., and Redmiles, D. F. Overcoming social barriers when contributing to open source software projects. *Computer Supported Cooperative Work (CSCW)* 28, 1 (2019), 247–290.

[468] Steinmacher, I., Silva, M. A. G., Gerosa, M. A., and Redmiles, D. F. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology* 59 (2015), 67–85.

[469] Su, W., Huang, H., Li, R., Chen, H., and Ge, T. Towards an effective method of ReDoS detection for non-backtracking engines. In: *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, Aug. 2024, 271–288.

[470] Sui, Y., Cheng, X., Zhang, G., and Wang, H. Flow2vec: value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[471] Sultana, S., Uddin, G., and Bosu, A. Assessing the influence of toxic and gender discriminatory communication on perceptible diversity in oss projects. *arXiv preprint arXiv:2403.08113* (2024).

[472] Sung, S., Cheon, H., and Han, Y.-S. How to Settle the ReDoS Problem: Back to the Classical Automata Theory. In: *Implementation and Application of Automata*. Ed. by Caron, P. and Mignot, L. Vol. 13266. Springer International Publishing, 2022, 34–49.

[473] Svyatkovskiy, A., Zhao, Y., Fu, S., and Sundaresan, N. Pythia: ai-assisted code completion system. In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, 2727–2735.

[474] Takata, K. *K-Takata/onigmo*. May 2024. URL: `https://github.com/k-takata/onigmo`.

[475] Tan, T., Li, Y., and Xue, J. Making k-object-sensitive pointer analysis more precise with still k-limiting. In: *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Springer. 2016, 489–510.

[476] Tandon, R., Wang, H., Weideman, N., Arakelyan, S., Bartlett, G., Hauser, C., and Mirkovic, J. Leader: defense against exploit-based denial-of-service attacks on web applications. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '23. Association for Computing Machinery, Hong Kong, China, 2023, 744–758.

[477] Taylor, M., Vaidya, R., Davidson, D., De Carli, L., and Rastogi, V. Defending against package typosquatting. In: *International conference on network and system security*. Springer. 2020, 112–131.

[478] Team, C. T. D. R. *Regex - Snort 3 Rule Writing Guide*. 2024. URL: https://docs.snort.org/rules/options/payload/regex.

[479] Team, G. *Golang/Go*. Aug. 2024. URL: https://github.com/golang/go.

[480] Team, G. *Introduction to the Go Compiler*. 2024. URL: https://go.dev/src/cmd/compile/README.

[481] Team, G. *Regexp Package - Go*. https://pkg.go.dev/regexp. 2024.

[482] Terrell, J., Kofink, A., Middleton, J., Rainear, C., Murphy-Hill, E., and Parnin, C. Gender bias in open source: pull request acceptance of women versus men.(jan 2016). *PeerJ Computer Science* (2016).

[483] Tewary, A. *Jharkhand, where not having Aadhaar could starve you to death — thehindu.com*. https://www.thehindu.com/news/national/other-states/jharkhand-where-not-having-aadhaar-could-starve-you-to-death/article61827611.ece.

[484] Thompson, K. Programming techniques: regular expression search algorithm. *Communications of the ACM* 11, 6 (June 1968), 419–422.

[485] Tomassi, D. A., Dmeiri, N., Wang, Y., Bhowmick, A., Liu, Y.-C., Devanbu, P. T., Vasilescu, B., and Rubio-González, C. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, 339–349.

[486] Toub, S. *Regular Expression Improvements in .NET 7*. May 2022. URL: http://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7.

[487] Toub, S. *Performance Improvements in .NET 8*. Sept. 2023. URL: https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/%5C#regex.

[488] Trade, U. N. C. on and Development. *Data and privacy unprotected in one third of countries, despite progress | UNCTAD*. en. Apr. 2020. URL: https://unctad.org/news/data-and-privacy-unprotected-one-third-countries-despite-progress.

[489] Tree-sitter Project. *Introduction — Tree-sitter*. https://tree-sitter.github.io/tree-sitter/index.html. 2024.

[490] Tripp, O., Guarnieri, S., Pistoia, M., and Aravkin, A. Aletheia: improving the usability of static security analysis. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, 762–774.

[491] Turoňová, L., Holík, L., Homoliak, I., Lengál, O., Veanes, M., and Vojnar, T. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, Aug. 2022, 4165–4182.

[492] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M., and Vojnar, T. Regex matching with counting-set automata. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020).

[493] Twist, L., Zhang, J. M., Harman, M., Syme, D., Noppen, J., and Nauck, D. Llms love python: a study of llms' bias for programming languages and libraries. *arXiv preprint arXiv:2503.17181* (2025).

[494] Unicode Consortium. *Unicode 5.1.0 — unicode.org*. https://www.unicode.org/versions/Unicode5.1.0/.

[495] Union, I. T. *Measuring digital development: ICT Price Trends 2020*. Tech. rep. ITU, 2020. URL: https://www.itu.int/en/ITU-D/Statistics/Pages/default.aspx.

[496] Unique Identification Authority of India. *Language & Translitration - Unique Identification Authority of India | Government of India — uidai.gov.in*. https://uidai.gov.in/en/contact-support/have-any-question/301-english-uk/faqs/enrolment-update/language-translitration.html.

[497] Unique Identification Authority of India. *Language Selection - Unique Identification Authority of India | Government of India — uidai.gov.in*. https://uidai.gov.in.

[498] United Nations. *Human Development Report 2021-22*. en. Tech. rep. United Nations, Sept. 2022. URL: https://hdr.undp.org/content/human-development-report-2021-22.

[499] United Nations Department of Economic and Social Affairs. *World Economic Situation and Prospects: Country Classification*. https://www.un.org/en/development/desa/policy/wesp/wesp_current/2014wesp_country_classification.pdf. 2014.

[500] United Nations Department of Economic and Social Affairs. *World Population Prospects - Population Division - United Nations*. URL: https://population.un.org/wpp/.

[501] Utture, A., Liu, S., Kalhauge, C. G., and Palsberg, J. Striking a balance: pruning false-positives from static call graphs. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, 2043–2055.

[502] Varatalu, I. E., Veanes, M., and Ernits, J. Re#: high performance derivative-based regex matching with intersection, complement, and restricted lookarounds. *Proc. ACM Program. Lang.* 9, POPL (Jan. 2025).

[503] Vashistha, A. and Anderson, R. Technology use and non-use by low-income blind people in india. *ACM SIGACCESS Accessibility and Computing* 116 (2016), 10–21.

[504] Vashistha, A., Brady, E., Thies, W., and Cutrell, E. Educational content creation and sharing by low-income visually impaired people in india. In: *Proceedings of the Fifth ACM Symposium on Computing for Development*. 2014, 63–72.

[505] Vasilakis, N., Karel, B., Roessler, N., Dautenhahn, N., DeHon, A., and Smith, J. M. Breakapp: automated, flexible application compartmentalization. In: *NDSS*. 2018.

[506] Vasilakis, N., Staicu, C.-A., Ntousakis, G., Kallas, K., Karel, B., DeHon, A., and Pradel, M. Preventing dynamic library compromise on node. js via rwx-based privilege reduction. In: *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*. 2021, 1821–1838.

[507] Vasilescu, B., Capiluppi, A., and Serebrenik, A. Gender, representation and online participation: a quantitative study of stackoverflow. In: *2012 International Conference on Social Informatics*. 2012, 332–338.

[508] Vasilescu, B., Posnett, D., Ray, B., Brand, M. G. van den, Serebrenik, A., Devanbu, P., and Filkov, V. Gender and tenure diversity in github teams. In: *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 2015, 3789–3798.

[509] Vázquez, S. R. Unlocking the potential of web localizers as contributors to image accessibility: what do evaluation tools have to offer? In: *Proceedings of the 12th International Web for All Conference*. 2015, 1–4.

[510] Vinadé, R. and Marczak, S. Unveiling developer perspectives: a survey on accessibility practices and requirements in software development. *WER 2024* (2024).

[511] Vu, D. L., Pashchenko, I., Massacci, F., Plate, H., and Sabetta, A. Towards using source code repositories to identify software supply chain attacks. In: *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2020, 2093–2095.

[512] W3C Web Accessibility Initiative (WAI). *The Business Case for Digital Accessibility — w3.org*. `https://www.w3.org/WAI/business-case/`.

[513] W3C Web Accessibility Initiative (WAI). *Web Content Accessibility and Mobile Web: Making a Website Accessible Both for People with Disabilities and for Mobile Devices*. en. URL: `https://www.w3.org/WAI/standards-guidelines/wcag-mobile-overlap/`.

[514] Wachs, J., Nitecki, M., Schueller, W., and Polleres, A. The geography of open source software: evidence from github. *Technological Forecasting and Social Change* 176 (2022), 121478.

[515] Walton, P. *Web Vitals*. `https://web.dev/articles/vitals`. 2020.

[516] Walton, P. and Pollard, B. *Largest Contentful Paint (LCP) | Articles*. en. URL: `https://web.dev/articles/lcp`.

[517] Wang, C., Zhang, W., Su, Z., Xu, X., Xie, X., and Zhang, X. Llmdfa: analyzing dataflow in code with large language models. *Advances in Neural Information Processing Systems* 37 (2024), 131545–131574.

[518] Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: a graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[519] Wang, X. *Introduction to Hyperscan*. 2017. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html`.

[520] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., and Zhu, H. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, Feb. 2019, 631–648.

[521] Wang, X., Zhang, C., Li, Y., Xu, Z., Huang, S., Liu, Y., Yao, Y., Xiao, Y., Zou, Y., Liu, Y., and Huo, W. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2023, 2427–2443.

[522] Wang, Y., Yue, Y., Wang, W., and Zhang, G. Uncovering non-native speakers' experiences in global software development teams——a bourdieusian perspective. *Computer Supported Cooperative Work (CSCW)* (2024), 1–36.

[523] Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[524] Wang, Z., Shi, T., He, J., Cai, M., Zhang, J., and Song, D. Cybergym: evaluating ai agents' cybersecurity capabilities with real-world vulnerabilities at scale. *arXiv preprint arXiv:2506.02548* (2025).

[525] Weideman, N., Merwe, B. van der, Berglund, M., and Watson, B. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In: *Implementation and Application of Automata*. Ed. by Han, Y.-S. and Salomaa, K. Lecture Notes in Computer Science. Springer International Publishing, 2016, 322–334.

[526] Weissberg, F., Pirch, L., Imgrund, E., Möller, J., Eisenhofer, T., and Rieck, K. Llm-based vulnerability discovery through the lens of code metrics. *arXiv preprint arXiv:2509.19117* (2025).

[527] Wessel, M., De Souza, B. M., Steinmacher, I., Wiese, I. S., Polato, I., Chaves, A. P., and Gerosa, M. A. The power of bots: characterizing and understanding bots in oss projects. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–19.

[528] West, M. and Medley, J. *Content Security Policy (CSP)*. `https://developer.chrome.com/docs/privacy-security/csp/`. 2012.

[529] West, M. and Sartori, A. *Content Security Policy Level 3*. `https://www.w3.org/TR/CSP3/`. 2025.

[530] Who.is. *WHOIS Search, Domain Name, Website, and IP Tools - Who.is*. URL: `https://who.is/`.

[531] Wikipedia. *C++23 — Wikipedia*. `https://en.wikipedia.org/wiki/C%2B%2B23`. 2024.

[532] Wikipedia. *TypeScript — Wikipedia*. `https://en.wikipedia.org/wiki/TypeScript`. 2024.

[533] Woodruff, W. *ReDoS "Vulnerabilities" and Misaligned Incentives*. Dec. 2022. URL: `https://news.ycombinator.com/item?id=34161221`.

[534] World Bank. *World Bank Country and Lending Groups*. https://datahelpdesk.worldbank.org/knowledgebase/articles/906519. 2024.

[535] World Bank. *Individuals using the Internet (% of population) | Data Catalog*. URL: https://data.worldbank.org/indicator/IT.NET.USER.ZS.

[536] World Economics. *World Economics | Economic data for the benefit of investors*. en-us. URL: https://www.worldeconomics.com/Regions/Global-South/default.aspx.

[537] World Health Organization. *World Report on Vision*. WHO, 2021. URL: https://www.who.int.

[538] World Wide Web Consortium (W3C). *Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile*. en. Feb. 2015. URL: https://www.w3.org/TR/mobile-accessibility-mapping/#wcag-2.0-and-mobile-content-applications.

[539] World Wide Web Consortium (W3C). *Success Criterion 3.1.2: Language of Parts*. 2023. URL: https://www.w3.org/WAI/WCAG21/Understanding/language-of-parts.html.

[540] World Wide Web Consortium (W3C). *Mobile Accessibility Challenges - Mobile Accessibility Task Force*. URL: https://www.w3.org/WAI/GL/mobile-a11y-tf/wiki/Mobile_Accessibility_Challenges.

[541] Worlddata.info. *List of 152 Developing Countries*. en. URL: https://www.worlddata.info/developing-countries.php.

[542] Wu, S., Wieland, J., Farivar, O., and Schiller, J. Automatic Alt-text: Computer-generated Image Descriptions for Blind Users on a Social Network Service. In: *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. CSCW '17. Association for Computing Machinery, New York, NY, USA, Feb. 2017, 1180–1192.

[543] Wüstholz, V., Olivo, O., Heule, M. J. H., and Dillig, I. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Legay, A. and Margaria, T. Vol. 10206. Springer Berlin Heidelberg, 2017, 3–20.

[544] Xiao, F., Huang, J., Xiong, Y., Yang, G., Hu, H., Gu, G., and Lee, W. Abusing hidden properties to attack the node. js ecosystem. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, 2951–2968.

[545] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[546] Xurxe Toivo García. *The troubled state of screen readers in multilingual situations — uxdesign.cc*. https://uxdesign.cc/the-troubled-state-of-screen-readers-in-multilingual-situations-f6a9da4ecdf3.

[547] Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., and Gaunt, A. L. Learning to represent edits. *arXiv preprint arXiv:1810.13337* (2018).

[548] Yuan, T., Li, G., Lu, J., Liu, C., Li, L., and Xue, J. Gobench: a benchmark suite of real-world go concurrency bugs. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, 187–199.

[549] Zaki, Y., Chen, J., Pötsch, T., Ahmad, T., and Subramanian, L. Dissecting Web Latency in Ghana. en. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, Vancouver BC Canada, Nov. 2014, 241–248.

[550] Zhang, H., Gong, L., and Versteeg, S. Predicting bug-fixing time: an empirical study of commercial software projects. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, 1042–1051.

[551] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A novel neural source code representation based on abstract syntax tree. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 783–794.

[552] Zhang, M. and Chen, Y. Link prediction based on graph neural networks. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 2018, 5171–5181.

[553] Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., and Sun, J. Detecting similar repositories on github. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2017, 13–23.

[554] Zhang, Zhang, M. R., Zhong, M., and Wobbrock, J. O. Ga11y: An automated gif annotation system for visually impaired users. en. In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, CHI '22*. New York, NY, USA, 2022.

[555] Zhao, G. and Huang, J. Deepsim: deep learning code functional similarity. In: *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2018, 141–151.

[556] Zhao, L. and Deek, F. P. User collaboration in open source software development. *Electronic Markets* 14, 2 (2004), 89–103.

[557] Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., and Su, Z. D2a: a dataset built for ai-based vulnerability detection methods using differential analysis. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2021, 111–120.

[558] Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[559] Zigisova, E. *The 2022 Web Almanac: Third Parties*. English. Tech. rep. HTTP Archive, Sept. 2022. URL: https://almanac.httparchive.org/en/2022/third-parties.

[560] Zimmermann, M., Staicu, C.-A., Tenny, C., and Pradel, M. Small world with high risks: a study of security threats in the npm ecosystem. In: *28th USENIX Security symposium (USENIX security 19)*. 2019, 995–1010.