A Binary Analysis Platform for Cryptographic Protocols

A dissertation submitted towards the degree *Doctor of Engineering (Dr.-Ing.)* of the Faculty of Mathematics and Computer Science of Saarland University

Faezeh Nasrabadi

Saarbrücken, 2025

Day of Colloquium:	18 March 2026
Dean of Faculty:	Prof. Dr. Roland Speicher
Chair of the Committee:	Prof. Dr. Sebastian Hack
Examination Board:	Prof. Dr. Cas Cremers
	Prof. Dr. Hamed Nemati
	Prof. Dr. Andreas Zeller
	Prof. Dr. Joshua Guttman
Academic assistant:	Dr. Robert Künnemann

Abstract

This thesis advances the formal verification of cryptographic protocol implementations at the binary level by tackling issues related to low-level programming languages, multi-language interactions, and microarchitectural side-channel vulnerabilities. We present CryptoBap, a framework for verifying security properties directly on the machine code of protocol implementations, supporting both ARMv8 and RISC-V architectures. CryptoBap performs crypto-aware symbolic execution on binary code to extract high-level models that represent all possible execution paths, handling complex control-flow constructs like indirect jumps and bounded loops through fully automated loop summarization. These models are then verified using protocol verification tools such as ProVerif, Tamarin, CryptoVerif, and DeepSec.

To support verification across systems built from multiple programming languages, we introduce a symbolic semantics that unifies languages operating on different atomic types. This semantics enables the modular composition of labeled transition systems and uses Dolev-Yao terms as symbolic abstractions of base-level types like bitstrings, allowing components to interoperate without the need for explicit translation layers. This approach makes it possible to analyze and verify security properties in complex, heterogeneous software systems.

We also extend CryptoBap to assess resilience against side-channel attacks by incorporating leakage models into the symbolic analysis. This allows us to reason not only about functional correctness but also about side-channel resistance. Our methodology has been applied to a range of case studies, including the Basic Access Control protocol utilized in e-passports, TinySSH, an implementation of SSH, WireGuard, a modern VPN protocol, and WhatsApp, the most popular messaging app. Using this methodology, on WhatsApp Desktop, we identified a privacy attack that allows a side-channel attacker to learn the victim's contacts, a post-compromise security violation by a clone attacker, and functional gaps between its implementation and specification.

Zusammenfassung

In dieser Arbeit widmen wir uns der formalen Verifizierung von Implementierungen kryptografischer Protokolle auf Binärcode-Ebene. Dabei adressieren wir Herausforderungen im Zusammenhang mit Low-Level-Programmiersprachen, der Interaktion zwischen Sprachen sowie mikroarchitekturellen Seitenkanalangriffen.

Wir stellen CryptoBap vor – ein Framework zur Verifizierung von Sicherheitseigenschaften direkt auf dem Maschinencode. CryptoBap unterstützt ARMv8 und RISC-V. Es führt eine kryptobewusste symbolische Ausführung auf Binärcode-Ebene durch, um Modelle zu extrahieren, die alle Ausführungspfade abbilden. Dabei werden auch komplexe Kontrollflusskonstrukte wie indirekte Sprünge und begrenzte Schleifen durch eine vollständig automatisierte Schleifenzusammenfassung behandelt. Diese Modelle werden anschließend mit ProVerif, Tamarin, CryptoVerif und DeepSec überprüft.

Um die Verifizierung über Systeme hinweg zu ermöglichen, die aus mehreren Sprachen bestehen, führen wir eine symbolische Semantik ein, die Sprachen vereinheitlicht, die auf unterschiedlichen atomaren Typen operieren. Sie ermöglicht die modulare Zusammensetzung von beschrifteten Übergangssystemen und verwendet Dolev-Yao-Terme als symbolische Abstraktionen von Basistypen wie Bitstrings. So können Komponenten ohne Übersetzungsschichten zusammenarbeiten, was die Analyse und Verifizierung in heterogene Softwaresysteme erlaubt.

Zur Bewertung der Widerstandsfähigkeit gegen Seitenkanalangriffe erweitern wir CryptoBap um Leckagemodelle, die in die symbolische Analyse integriert werden. Damit können wir sowohl funktionale Korrektheit als auch Resistenz gegenüber Seitenkanalangriffen bewerten.

Unsere Methodik haben wir auf mehrere Fallstudien angewandt: das Basic Access Control-Protokoll in elektronischen Reisepässen, TinySSH (eine SSH-Implementierung), WireGuard (ein modernes VPN-Protokoll) und WhatsApp. Dabei haben wir bei WhatsApp Desktop einen Datenschutzangriff identifiziert, bei dem ein Seitenkanalangreifer die Kontakte des Opfers ausspähen kann. Außerdem fanden wir eine Sicherheitsverletzung nach einer Kompromittierung durch einen Klonangriff sowie funktionale Diskrepanzen zwischen Implementierung und Spezifikation.

Acknowledgements

This thesis has been a meaningful journey, made all the more rewarding by the wonderful people I have had the pleasure of meeting. The support I received has transformed this academic pursuit into a truly memorable experience.

First and foremost, I sincerely thank my outstanding supervisors, Cas Cremers, for being an exceptional mentor and for his ongoing support and patience, and Hamed Nemati, for his guidance throughout the process and his openness to discussion and the absence of any pressure. Thank you both for your trust and appreciation of my ideas, which mean a great deal to me. My gratitude also extends to Robert Künnemann, whose continued support, expertise, and advice have significantly contributed to my growth.

I would also like to express my appreciation to the reviewers of this thesis and my defense committee, including Cas Cremers, Hamed Nemati, Andreas Zeller, Joshua Guttman, Sebastian Hack, and Robert Künnemann, for their time and thoughtful feedback during this final stage. Furthermore, my special thanks go to Andreas Lindner for sparking my interest in program verification and for helping me develop the necessary technical skills during the early stages of my research. I am grateful to the CISPA faculty members—Andreas Zeller, Ali Abbasi, Sven Bugiel, and especially Michael Schwarz and his student Daniel Weber—for their kind support and for the many valuable, in-depth discussions we shared.

I would like to thank Marco Patrignani for our scientific discussions and for suggesting the use of syntax highlighting, which has significantly improved my ability to communicate my research ideas more clearly and effectively. My gratitude also goes to Jana Hofmann for her kindness in taking the time to prepare and share her thesis template with me.

Many thanks to my wonderful lab mates—Kevin Morio, Tiziano Marinaro, Patrick Speicher, Ilkan Esiyok, Matthis Kruse, and Xaver Fabian—who made my academic journey memorable through insightful discussions, shared successes, and daily camaraderie. I am fortunate to have had an amazing group of friends who made me laugh, listened to my daily concerns, and shared many joyful moments. I am grateful to Sanam, Keno, Soheil, Azadeh, Carolyn, Hamed, Kimia, Saleh, Xinyi, Pouya, Elham, Hossein, Esra, Daniel, Ensiye, Hamidreza, CheolJun, Changhun, Addison, Maede, Au-

rora, Mandana, Matteo, Arash, Nilofar, Omid, Hossein, Arthur, and Eva for making this city feel like home.

When I first moved to Saarbrücken during the COVID-19 pandemic, I did not know anyone, and I was fortunate to meet Silvia and Mohammad. I am deeply grateful for their unconditional support in helping me find a home, furnish it, and feel part of a family while living abroad—something I will never forget.

My deepest gratitude goes to my family, including my grandparents, uncles, aunts, and cousins, whose thoughtfulness and kindness from afar warm my heart and give me strength to continue this journey. Mom and Dad, words cannot express how profoundly thankful I am for your endless, unwavering love and support, which have continuously empowered me to persevere and maintain hope. Farnaz, thank you for always being by my side in both good and difficult times. You have been strong and supportive, like a caring big sister, even though you are much younger. I truly appreciate you. My heartfelt thanks to my beloved partner, Mostafa, for his bravery, patience, insightful advice, and steadfast support. His belief in me strengthened my courage and self-confidence during difficult times, and I could not have completed this journey without him.

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, 11/2025

gez. / signed

Faezeh Nasrabadi

Contents

Chapter 1

Introduction

Cryptographic protocols form the backbone of modern digital security, protecting sensitive data across online interactions. They are a vital part of end-user security on the Internet. Yet their design and implementation remain prone to subtle and sometimes catastrophic errors—both at the specification level (e.g., the POODLE attack on SSL version 3.0, which exploited predictable padding in CBC-mode encryption [49]) and at the implementation level (e.g., Heartbleed, `CVE-2014-0160`).

Formal methods provide a systematic and rigorous foundation for detecting such vulnerabilities and, in some cases, proving their absence entirely. Protocol verification provides guarantees about privacy and authenticity by analyzing abstract models of communication between agents, typically under symbolic attacker models such as Dolev-Yao [74]. Program verification, in contrast, reasons about concrete implementations, providing functional and confidentiality guarantees under a more powerful attacker model unconstrained by symbolic assumptions. Several works have adopted features from one domain in the other but this comes at the cost of complex, inflexible execution models and high development cost. Composing these techniques bridges the gap between tools and verification technologies, allowing them to continue evolving independently.

Recent efforts have made substantial progress in verifying both abstract protocol specifications and high-level implementations written in languages like *C* or *F#* [9, 8, 59, 85, 18, 40]. Still, a significant *verification gap* remains between the correctness proofs developed at these higher levels and the object code that ultimately runs on hardware. Program behavior at the source level can diverge from its compiled behavior, particularly when aggressive compiler optimizations introduce subtle transformations [177]. These may eliminate critical checks—such as those for integer overflow [171] or null pointers [177]—or even remove security-critical operations like secret scrubbing [156]. In some cases, constant-time code can be transformed into non-constant-time binaries, introducing timing side channels [158]. Fig. 1.1 illustrates one such instance, where `GCC ARM V11.2.1` removes a memset intended to erase a secret

key from memory. Even formally verified compilers like CompCert [112] do not fully address these issues, as verifying all the optimization passes typically used in practice remains out of reach.

Beyond compiler-induced vulnerabilities, equally critical risks arise in protocol session management, where stateful behaviors at the binary implementation level can leak sensitive information. Cryptographic protocols depend on cryptographic primitives that are carefully designed to withstand side channel attacks, and verification efforts have traditionally concentrated on these core components. However, protocols encompass more than just cryptographic operations. Keys must be stored, retrieved, and managed across different layers of the protocol, where processes like session initiation, device synchronization, and periodic rekeying occur. These higher-level operations can unintentionally leak sensitive information and introduce vulnerabilities that current protocol analysis cannot capture.

Considering low-level implementation for protocol analysis also offers an opportunity to go beyond functional correctness and detect attacks that result from interactions with the hardware. Notably, side-channel attacks can exploit hardware-induced behaviors—such as timing variations or memory access patterns—to extract sensitive information. While traditional side-channel detection tools exist, they primarily target cryptographic primitives or isolated library code, often missing protocol-level patterns that amplify leakage. For example, error handling, session establishment order, or memory access during key derivation can all introduce exploitable side-channel signals. Yet no existing work integrates this level of analysis into complete protocol verification.

Session management highlights this gap particularly well. Secure messaging protocols such as Signal's Sesame [68] or the Messaging Layer Security (MLS) protocol [24] manage complex state across multiple devices. Operations like session key initialization, synchronization, and handling multi-device states are critical for security, but they can reveal exploitable patterns through their side effects. Post-compromise security mechanisms—such as periodic key updates and healing processes—are similarly vulnerable. Although these mechanisms are designed to restore confidentiality after compromise, their real-world implementations may still expose subtle leaks during rekeying or state transitions.

In a nutshell, limiting security analysis to cryptographic libraries overlooks the complex interactions that occur in protocol and session management. To fully understand the risks facing modern secure messaging systems, side-channel analysis must extend beyond primitives to the full protocol flow, including binary-level implementations. Only through this approach can we effectively address the overlooked yet critical vulnerabilities introduced during real-world protocol execution.

Adding to this complexity is the fact that real-world systems rarely consist of a single programming language. Especially in complex systems like network protocols and

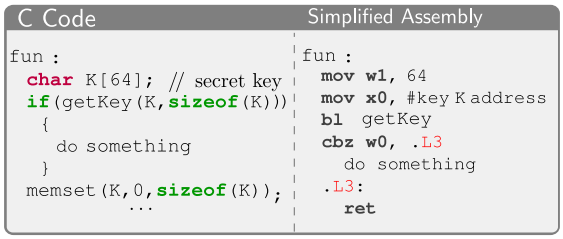


Figure 1.1: An example of how compilers can invalidate code for secret scrubbing

operating systems, where components written in different languages must communicate seamlessly. At a minimum, the analysis of network protocol implementations consist of (a) a party written in a real programming language, (b) their communication partner(s) operating according to a specification and modeled abstractly, such as in the applied pi calculus, and (c) an attacker, which is underspecified but typically limited by some threat model, like the Dolev-Yao (DY) model or the cryptographic model of a time-bounded probabilistic Turing machine. As protocol properties extend over multiple parties in the presence of an attacker, an implementation-level analysis needs to reason about these types of components and their interactions.

Traditional approaches in program verification and system analysis often fall short in these multi-language environments, as they typically assume a homogeneous language framework. This assumption overlooks the challenges presented by the interplay of different programming languages, each with its unique syntax, semantics, and operational paradigms. The state of art [17, 161, 13, 12, 150] describes the heterogeneous system as a composition of labeled transition systems (LTS). LTS are very flexible; they can abstract any programming language. Hence, the composition of LTS is the key to capturing cross-language communication, be it at runtime [161] or compile time [150]. Current composition approaches insist on *translating* base values that are truly incompatible, e.g., bitstrings and abstract DY terms. This leads to shortcomings that we describe in detail in Sec. 3.1 and solve in Chapter 3.

In short, the translation approach is:

- **Hard to apply due to strong parsing assumptions:** For instance, keys must always be syntactically distinguishable from bitstrings used elsewhere and network messages must use known encodings [13, 12]. We can avoid this assumption by not requiring a 'universal' translation a priori, but instead by tracking what the application actually does. We elaborate on this in Sec. 3.1.1 and use example 3.1 to show how we solve this problem.

- **Limited in the ability to capture adversarial bit-level reasoning:** The translation approach notoriously struggles with mixed values, for instance, abstract encryption terms or keys that the implementation manipulates on the bit level.

In Sec. 3.1.2, we further explain this issue and discuss our solutions using examples 3.2 and 3.3.

- **Not truly versatile:** The complexity-theoretic computational attacker, e.g., is not compatible with standard language semantics. We argue the compatibility of our framework with a computational attacker in Sec. 3.1.3.

We solve these shortcomings by forgoing the translation step between different base types as, e.g., bitstrings and DY terms. The crux is, in our view, that the DY model is a symbolic abstraction (it is sometimes called the 'symbolic model of cryptography' [1]), whereas the translation approach and the other method of composition treats DY terms as if they were concrete values (e.g., see [13, Sec. 4.2]).

In summary, these observations emphasize a hierarchy of challenges in verifying cryptographic protocols. The primary challenge lies in the absence of a unified verification framework that bridges protocol and program-level verification at the binary implementation level, accounting for compiler optimizations and hardware interactions. Additionally, there are secondary but critical challenges to reason about heterogeneous multi-language implementations and overcoming the limitations of translation-based approaches between symbolic and computational models. Together, these challenges highlight the need for a compositional verification methodology and a binary verification framework, which this thesis develops in the following chapters.

The remainder of this introduction outlines our key contributions (Sec. 1.1) and supporting publications (Sec.1.2), before presenting the structure of this thesis (Sec.1.3).

1.1 Contributions

To address these challenges, our work develops a compositional verification framework that unifies protocol and program reasoning, supports multi-language implementations, and models low-level execution behaviors, including side channels. This eliminates the need for trusting compilers and provides a higher assurance about the security and correctness of protocols.

1.1.1 Parallel Composition for Multi-language Verification

We address the challenge of combining different semantic models by extending *parallel asynchronous composition*, which merges two independently defined systems—each communicating with an unspecified external environment—into a single, interacting system. This technique enables us to reason about multi-component systems, such as protocol participants and adversaries, in a unified way.

To enable such composition across heterogeneous systems, we establish general principles for combining languages that operate on different atomic types. In particular, we show that the DY model should be composed with a labeled transition system (LTS) that describes the other components at a compatible level of abstraction—namely, using *symbolic execution semantics*.

Symbolic execution models a program's behavior by executing it with symbolic inputs, rather than concrete values. These symbolic values—abstract representations of inputs at the object level—are neither program variables nor meta-variables, but instead serve as placeholders for unknown but consistent inputs. They allow us to track how outputs depend on inputs across execution paths, without fixing specific values.

Crucially, these symbolic values act as a 'bridge' between the two semantics, enabling communication without the need to translate values from one representation to another. This provides a clean and uniform way to describe message passing between the DY model and the symbolic execution model.

Assuming we are given a symbolic execution semantics (which can be derived in standard ways), we define a structured class of LTS, which we call *symbolic LTS*, and a corresponding parallel composition operator. This operator exploits symbolic values to mediate communication and supports the transfer of logical statements about symbols between the two systems. It thus forms the semantic foundation for modular, cross-language verification of protocol implementations.

To summarize, we make the following contributions:

- We propose a framework (Part I) for the parallel asynchronous composition of components written in different languages with applications for various methods of analysis, e.g., secure compilation, code-level verification, model extraction (such as ours), or monitoring.
- Using this framework and additional theorems, we support integrating DY attackers into arbitrary languages. This is necessary to make the end-to-end proof feasible.
- We discuss three methods of improving symbolic execution engines with DY support using combined deduction relations (i.e., $\vdash_{12}$) in Sec. 3.3.
- We formalized our framework and proved its soundness and the DY attacker and library properties in HOL4 [165].

1.1.2 A Binary Analysis Platform for Cryptographic Protocols

We extend HolBA [116] to enable symbolic execution of RISC-V and ARM binaries in the presence of an arbitrary attacker and trusted cryptographic code. Our symbolic

execution engine resolves indirect jumps and handles bounded (compile-time) loops using the summarization technique [163], which we fully automate.

To support formal security verification, we define a sound translation from symbolic execution traces to intermediate models suitable for off-the-shelf protocol verifiers such as ProVerif [45] and CryptoVerif [46]. Additionally, we generate models in Sapic$^+$ [60], a process calculus that (soundly) compiles to a range of backends including Tamarin [126], ProVerif, and DeepSec [61].

Our threat model assumes that the attacker controls a set of functions whose input/output behavior can simulate, for instance, network syscalls or hypercalls in a parallel VM [70, 100]. The underlying execution platform is trusted to implement the machine-code semantics correctly and includes a trusted set of cryptographic functions. When exporting to ProVerif, we assume the Dolev-Yao attacker model with perfect cryptography. We abstract cryptographic primitives as uninterpreted functions and delegate their correctness to separate efforts (e.g., HACL*[179], CompCert + FCF[35], or synthesis approaches like Fiat-Crypto [81]).

We present CryptoBap, which takes as input:

1. The binary implementation of protocol participants,
2. A symbolic model of cryptographic functions, and
3. A trace property to verify (e.g., mutual authentication or secrecy).

Our verification pipeline starts by transpiling the binary into **BIR**, HolBA's internal language, using a formally verified transpiler that preserves machine-code semantics. However, the original **BIR** language lacks constructs for protocol reasoning, such as network I/O and randomness. We address this by extending **BIR** to support protocol-specific behaviors (see Sec. 5.1).

We then symbolically execute the **BIR** code to (a) build an execution tree capturing all paths, (b) instrument the code with trace events for observable actions, and (c) extract high-level models suitable for protocol analysis. This process yields two forms of symbolic models:

- `IML` (see Sec. 4.2,Sec. 5.3.1), which we pass to Csec-modex[6] and convert into the input languages of ProVerif and CryptoVerif;
- Sapic$^+$, an extended version of Sapic [106], targeting Tamarin, DeepSec, and ProVerif (see Sec. 4.3 and Sec. 5.3.2).

To ensure the faithfulness of translation to `IML`, we define a mixed execution semantics that allows protocol participants from different abstraction layers (e.g., binary

code and symbolic models) to run and interact concurrently. This semantics also eliminates the need for concurrency handling in symbolic execution itself.

Moreover, we instantiate the general verification framework described in Part I to demonstrate the end-to-end correctness of our toolchain for extracting SAPIC$^+$ models from low-level protocol implementations. In the symbolic setting, our framework enables fully mechanized security proofs. Unlike the computational setting, which suffers from limited automation and the lack of compositionality, our symbolic methodology scales more effectively to real-world implementations.

Using CRYPTOBAP, implementation-level flaws—such as stack-based buffer overflows (e.g., in Sami FTP Server 2.0.1)—are detected during symbolic execution and protocol-level flaws—such as the triple-handshake attack on TLS [39]—are identified by the protocol verifiers. Our soundness results guarantee that both kinds of vulnerabilities are captured accurately.

To summarize:

- We present CRYPTOBAP to automate the verification of cryptographic protocols' binary. Our framework explores all execution paths of protocols, resolves indirect jumps, and handles (compile-time) bounded loops automatically.
- We extend the vanilla symbolic execution engine in HolBA to automate the model extraction of cryptographic protocols. The way we extend the engine is significant. While HolBA's vanilla symbolic execution considers the program a holistic entity and thus basically encodes the semantics of the language (**BIR**), our semantics regards only a part of the whole program and abstracts cryptographic libraries, attacker calls, and random number generation.
- We formally verify the soundness of our approach and show that verified properties can be transferred back to the binary of analyzed protocols.
- To evaluate CRYPTOBAP, we have successfully verified multiple case studies, ranging from toy examples, e.g., CSur, to TinySSH and WireGuard.

1.1.3 Side-Channel Analysis of Protocols Implementations

We present a methodology for analyzing cryptographic protocol implementations at the binary level, addressing both functional correctness and side-channel resilience. Our work extends CRYPTOBAP by integrating observation models [136, 54] (a.k.a., leakage contracts [89]) that enable automated analysis of real-world protocols against side-channel attacks.

Building on CRYPTOBAP's core methodology—transpiling binaries into an intermediate representation for symbolic execution and model extraction—we introduce the

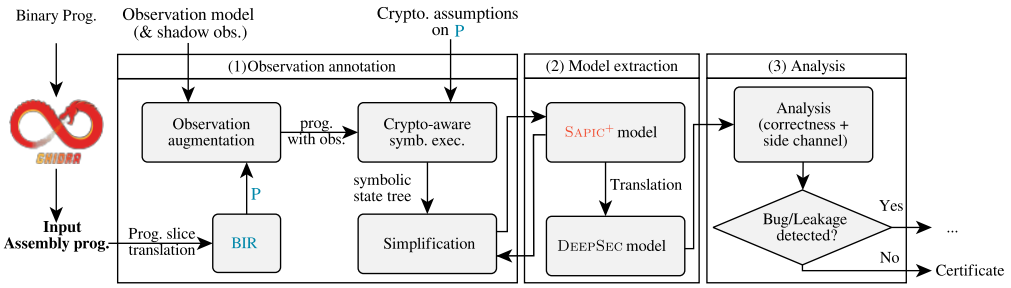


Figure 1.2: Organization of our approach for cryptographic protocols side-channel analysis.

first integration of side-channel analysis into the protocol verifier DeepSec, thus enabling holistic security evaluation of cryptographic protocols.

Unlike prior work that focuses on source code or assumes high-level models, our methodology applies directly to real-world binaries, requiring neither source code nor formal protocol specifications. We explicitly model protocol-level side channels—such as those caused by control-flow divergences during key exchange or variations in error handling—that can undermine cryptographic guarantees even when cryptographic primitives are implemented correctly.

To address the inherent scalability challenges of binary analysis, we combine symbolic execution with optimizations such as live-variable analysis [95] and abstraction of semantically irrelevant details. This enables efficient reasoning over protocols with complex state transitions and interactions.

As Fig. 1.2 illustrates, our approach begins with reverse-engineering the executable binaries using Ghidra [71], after which we transpile the resulting assembly code into **BIR** using the HolBA framework. We then annotate the **BIR** program with attacker observations and symbolically execute it to explore all execution paths. We simplify and translate the resulting symbolic execution tree into a formal model in SAPIC$^+$ calculus, which is then further simplified to improve scalability. Finally, we translate this model into formats compatible with automatic analysis of side-channel resilience using DeepSec. The selection of verifiers in our analysis depends on the specific properties we aim to prove. E.g., when considering privacy-preserving properties, the DeepSec toolchain is the most suitable option, while, for other properties, Tamarin and ProVerif perform better.

As a key case study, we perform the first formal verification of the WhatsApp Desktop binary, extracting a faithful model of its session management protocol (Sesame) and double ratchet mechanism. Despite WhatsApp being the most widely used messaging platform with over 2 billion users [132], its session management implementation has not been previously modeled or verified. Given its closed-source nature and the precedent of vulnerabilities in similar systems (e.g., EternalBlue, `CVE-2017-0144`), this lack of research is concerning.

Analyzing real-world binaries is a challenging task. The main difficulty lies in isolating protocol logic within large codebases, where such logic typically constitutes only a small fraction—e.g., about 0.1% in WhatsApp binary. To address this challenge, we have used Ghidra [71], an open-source reverse-engineering platform, to identify protocol-related code regions. Ghidra's features, including disassembly, decompilation, control-flow analysis, and data dependency analysis, were instrumental in selecting target regions for our analysis. To systematically identify key components such as cryptographic operations and network I/O within the disassembled binary, we employ existing methods similar to those described in [127]. Once these components are located, we use Ghidra's data dependency analysis to develop protocol logic on top of them. Subsequently, using several Ghidra scripts that we have developed, we disassemble the targeted code regions for further analysis with our framework.

Our analysis verifies critical security properties—such as forward secrecy and post-compromise security—and reveals that known clone attacks on Signal [68] also apply to WhatsApp. This undermines the expected guarantees of the double ratchet protocol as implemented in practice. Our extraction process reconstructs the protocol logic from machine code that enables parsing serialization formats, inferring state machines, and identifying cryptographic primitives.

Beyond functional correctness, we demonstrate how integrating side-channel analysis at the protocol level enables the discovery of new privacy attacks. Specifically, we identify a novel vulnerability whereby a malicious co-resident app can leverage hardware side channels to enumerate a victim's communication partners. By observing subtle differences in WhatsApp's session setup—depending on whether the recipient is a new or known contact—the attacker can infer private communication metadata. This finding highlights the significance of taking into account hardware leakages during protocol analysis, which, so far, was mainly considered at the level of cryptographic libraries [58].

Finally, we emphasize the broader implications for the ecosystem. Among the five most-used messaging platforms—WhatsApp, WeChat/Wexim, Facebook Messenger, Telegram, and Snapchat—only Telegram is fully open source [132]. This reinforces the importance of binary-level analysis for assessing security in deployed systems.

To summarize, our contributions are as follows:

1. We extend CryptoBap with leakage contracts, enabling protocol analysis that includes hardware-induced leaks, something prior work had not integrated into protocol verifiers. This yields the first pipeline that joins protocol model extraction with side channels analysis.

2. We extract the first formal model of WhatsApp's implementation using CryptoBap, verify forward secrecy and confirm a post-compromise security break

against a clone attacker. We also identify implementation–specification gaps in ratcheting behavior.

3. Applying our methodology to WhatsApp's session initialization, we discover a privacy leak: an attacker can distinguish first-time from existing contacts, enabling social-graph inference. We also confirm the known unlinkability break in BAC.

To summarize, our framework supports reasoning about leakage sources such as timing variations, memory access patterns, and control-flow divergences intrinsic to protocol logic. By integrating side-channel analysis into the protocol verification process, our work establishes a new paradigm for securing cryptographic protocols, with side-channel resilience a first-class requirement instead of an afterthought.

1.2 Publications

This thesis is based on the following papers, in which I contributed as the main author to all of them. Among these, [P1] and [P2] are peer-reviewed publications, while [P3] is currently under review. Papers [P1, P2, P3] were produced through a collaborative effort involving Hamed Nemati and Robert Künnemann, who engaged in comprehensive discussions, assisted in the evaluation of our results, commented, and contributed to the writing of our papers.

[P1] Nasrabadi, F., Künnemann, R., and Nemati, H. Cryptobap: a binary analysis platform for cryptographic protocols. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. Association for Computing Machinery, Copenhagen, Denmark, Aug. 2023, 1362–1376. URL: https://doi.org/10.1145/3576915.3623090.

[P2] Nasrabadi, F., Künnemann, R., and Nemati, H. Symbolic parallel composition for multi-language protocol verification. In: *2025 IEEE 38th Computer Security Foundations Symposium (CSF'25)*. IEEE Computer Society, Los Alamitos, CA, USA, June 2025, 458–473. URL: https://doi.ieeecomputersociety.org/10.1109/CSF64896.2025.00030.

[P3] Nasrabadi, F., Künnemann, R., and Nemati, H. Automated side-channel analysis of cryptographic protocol implementations. *arXiv preprint* (2025). URL: https://arxiv.org/abs/2511.11385.

Furthermore, the implementation of our contributions, including all source code, is publicly available at https://github.com/FMSecure/CryptoBAP.

1.3 Structure of This Thesis

We organize the thesis into two main parts, each beginning with a chapter that introduces the necessary background for the material that follows. The preliminaries in Part II build on those from Part I, especially the foundational concepts introduced in Sec. 2.3.2 and Sec. 2.3.3. After the two core parts, we conclude with Chapter 8, which surveys related work, and Chapter 9, where we reflect on our results and outline directions for future research.

Part I: Multi-language Symbolic Parallel Composition. The first part of the thesis is based on our paper [P2], in which we develop a general framework for symbolic composition across multiple languages. We begin with foundational concepts in Chapter 2, including labeled transition systems, symbolic execution semantics, the limitations of translation-based approaches, and the attacker and library models in the Dolev-Yao setting. In response to these limitations, we introduce a novel symbolic parallel composition method in Chapter 3, along with formal results on composition and refinement.

Part II: Binary Analysis of Protocol Implementations. In the second part of the thesis, we present the development of the CryptoBap framework, first introduced in our paper [P1] and later extended in our work [P3] to incorporate observation models. CryptoBap is a binary analysis framework that enables the verification of cryptographic protocols directly at the machine-code level, thereby eliminating the need to trust compilers. We extend the HolBA framework to verify ARMv8 and RISC-V machine code of cryptographic protocols in Sec. 5.1 and Sec. 5.2. Next, Sec. 5.3 details the process of extracting formal models from binary code, which are then translated into representations suitable for automated verification tools, including ProVerif, CryptoVerif, Tamarin, and DeepSec. To validate our approach, we instantiate our general composition framework from Part I and prove soundness in both the computational setting (Sec. 6.1) and the symbolic setting (Sec. 6.2). Finally, Chapter 7 presents an evaluation across a range of case studies—from toy examples to real-world systems—including BAC, TinySSH, WireGuard, and WhatsApp.

Part I

Multi-language Symbolic Parallel Composition

Chapter 2

Preliminaries

In this chapter, we start by revisiting the standard definition of labeled transition systems and the traditional communicating sequential processes (CSP)-style asynchronous parallel composition, then describe the symbolic execution technique. Finally, we discuss the protocol verification technique and then define the models for the computational attacker, DY attacker, and DY library, clarifying the unique roles of each.

2.1 LTS and their Composition

Labeled transition systems (LTS) provides a generic semantic model for capturing the operational semantics of systems [73, 148]. An LTS consists of a set of configurations C connected by a transition relation $\xrightarrow{o} \subseteq C \times \mathbb{E} \times C$ that releases an event $o \in \mathbb{E}$ when the system moves between configurations, and an initial configuration $c \in C$ within that space. Given that a language has a formalized semantics, a program behavior can typically be described as an LTS. Thus, it is interesting to combine LTS to reason about heterogeneous systems, wherein some transitions are asynchronous, e.g., programs are performing internal computations independently, while others are synchronized, e.g., one program sends a message and the other one receives it.

The communicating sequential processes (CSP)-style [53] asynchronous parallel composition supports both types of transitions and can be applied to LTS. We prefer a less descriptive, but shorter name, assuming that modern systems require both synchronous and asynchronous transitions. Transitions are synchronous if both carry the same event ($o \in \mathbb{E}_1 \cap \mathbb{E}_2$), and all others are asynchronous. Hence, in a composed configuration (c_1, c_2), we move synchronously to (c_1', c_2') with event o provided *both* systems can move ($c_1 \xrightarrow{o} c_1'$ and $c_2 \xrightarrow{o} c_2'$) and otherwise ($o \notin \mathbb{E}_1 \cap \mathbb{E}_2$), we move to (c_1', c_2) or (c_1, c_2') if either of the systems can make a transition.

Synchronizing events can be used to transmit messages [17, 161]. For example, when combining two systems A and P with a shared event $A2P(m)$, system A can

have a rule that determines m from its current configuration, whereas P has a rule that non-deterministically accepts $A2P(m^*)$ for any m^* and incorporates it into the follow-up configuration. Combining both systems via asynchronous parallel composition, we obtain synchronous message passing from A to P.

2.2 Symbolic Execution Technique

Symbolic execution is a program analysis technique that analyzes programs by assigning their inputs to symbolic values instead of concrete ones [20]. It explores all program execution paths using symbolic values—introduced at the object level—instead of concrete ones for inputs. An example is a language with a memory that maps registers to bitstrings. Its symbolic execution allows the memory to map registers to either symbols or bitstrings. Starting from an initial symbolic configuration, the execution explores all possible paths and collects the execution effects in a final symbolic configuration for each path. Each symbolic configuration, in addition to a map from variables to symbolic expressions (i.e., where symbols represent initial configuration variables), also contains a path condition that is a logical predicate describing what is known about the symbol. For instance, $r_A = \texttt{0x0} \vee r_A = \texttt{0x1}$ if register r_A is known to be either 0 or 1 because it passed some condition.

Using an SMT solver, we can identify symbolic paths that may lead to bugs, such as a division by zero. SMT solvers [78, 72, 63] are tools designed to solve the satisfiability modulo theories (SMT) problem for a practical subset of inputs, and they generate concrete inputs to satisfy symbolic path conditions. This process enables automatic test case generation and helps detect bugs in mission-critical and security applications. To combat the path explosion problem, symbolic execution engines make logical deductions on these predicates to prune paths that are unreachable, e.g., using an SMT solver. The more powerful the deduction engine, the fewer paths need to be explored, but the more computationally expensive these deductions are.

We capture these elements—symbols, predicates, and deductions-—by giving our LTS more structure. Let τ be the silent transition, then:

Definition 2.1 (Symbolic LTS)**.** A symbolic LTS is an LTS $(\tilde{C}, \mathbb{E}, \rightarrow)$ for which there is a symbol space $\mathcal{E}$, a predicate space $\mathcal{P}$, and a deduction relation $\vdash \subseteq 2^{\mathcal{P}} \times \mathcal{P}$ such that:

- $\tilde{C} = 2^{\mathcal{E}} \times 2^{\mathcal{P}} \times C$ for some configuration space C and
- For any predicate set Π, predicate φ, symbols set Σ, and configuration c, we have: $\Pi \vdash \varphi \implies (\Sigma, \Pi, c) \xrightarrow{\tau} (\Sigma, \Pi \cup \{\varphi\}, c)$.

For brevity, we denote such LTS with $(\mathcal{E}, C, \mathbb{E}, \rightarrow, \mathcal{P}, \vdash)$.

The second condition establishes the relation between the logical deduction relation which is language-specific, and the current predicate set: logical deductions can be made at any time, and the knowledge we conclude (encoded inside the predicate) is added to the symbolic configuration. Typically, the configuration space C and event space $\mathbb{E}$ are built on the symbol space $\mathcal{E}$, e.g., in the example above, the symbolic memory was a function from registers to the union of bitstrings and the set of symbols $\Sigma \subseteq \mathcal{E}$. This is only implicit in the mathematical notation, it is, however, explicit in our HOL4 formalization, where the types of C and $\mathbb{E}$ are parametric in the (polymorphic) type $\mathcal{E}$. The first element Σ mainly tracks which symbols have been used so far, increasing monotonically.

Every symbolic LTS, also referred to as a component, must transmit only references to their messages in the form of symbols to other components. Symbols relate to the values that are transmitted like a variable n relates to the set of integers, i.e., as a representation. When a value is manipulated, the relation between the original and the changed value, each represented by a different symbol, is itself represented with a predicate connecting the two symbols. Consequently, a symbol always signifies the same value (in a run), and the predicates associated with distinct components articulate the same properties.

2.3 Protocol Verification Technique

The protocol verification technique employs formal methods and modeling to systematically identify logical flaws in communication protocols to ensure functional correctness. It involves creating a formal model of the protocol, typically in applied pi calculus or multi-set rewrite rules, and employing automated tools to verify security properties of these models. In the study of protocol verification, two principal attacker models are widely used: the computational (cryptographic) model [84] and the symbolic or Dolev–Yao (DY) model [74]. The cryptographic model treats the adversary as an arbitrary probabilistic program that operates within polynomial time relative to the security parameter, such as the key length. This restriction is essential, as without it, an attacker could simply guess keys through exhaustive search. Within this model, cryptographic primitives are defined by their computational hardness, and proofs rely on assumptions, such as the infeasibility of forging digital signatures.

By contrast, the DY model adopts an abstract, symbolic perspective in which messages are treated as terms built from symbols, and keys are idealized as unbreakable. The attacker's capabilities are governed by a finite set of deduction rules: for instance, possessing a key allows one to encrypt or sign messages, but not to perform these actions without it. While the computational model assumes that "everything is possible unless explicitly proven infeasible," the DY model enforces the dual principle that "only

operations permitted by the rules are possible." This symbolic abstraction makes the DY model particularly suited for automated reasoning and formal verification, even though it abstracts away computational hardness assumptions.

2.3.1 Computational Attackers

Within the computational model of cryptography, messages are represented as bitstrings, and cryptographic primitives are formalized as probabilistic algorithms defined over these bitstrings. Potential adversaries are modeled as probabilistic Turing machines, thereby capturing the notion of computationally bounded attackers. Since keys are modeled as bitstrings, any leakage of key material (i.e., partial knowledge of key bits) effectively reduces the computational complexity of the decryption task, thereby weakening the overall system security. The notion of computational security is inherently probabilistic and is defined by bounding the adversary's maximum success probability under explicit constraints on its computational resources. A scheme is considered secure in this setting if every probabilistic polynomial-time adversary has only a negligible probability of success, where negligible denotes a function that decreases asymptotically faster than the inverse of any polynomial in key length. We utilize this model to show the soundness of one of our methodologies in Part II.

2.3.2 Dolev-Yao Attackers

In Dolev-Yao (DY) model, messages are modeled as *terms*[1]. Constants are taken from an infinite set of names $\mathcal{N}$, divided into public names $\mathcal{N}_{\mathsf{pub}}$ (e.g., agent names) and secret names $\mathcal{N}_{\mathtt{priv}}$ (like keys and nonces). We also assume a set of variables $\mathcal{V}$ for values that the DY attacker receives. The set of terms $\mathcal{T}$ is then constructed over names in $\mathcal{N}$, variables in $\mathcal{V}$ and applications of function symbols in $\mathcal{F}$ on terms. Let $f \in \mathcal{F}^n$ denote a function symbol with arity n. For the moment, we consider only two function symbols, $\mathcal{F} = \{\mathsf{senc}, \mathsf{sdec}\} = \mathcal{F}^2$. The term $\mathsf{senc}(m, k)$ models the symmetric encryption of another term m with the key $k \in \mathcal{N}_{\mathtt{priv}}$. A set of equations $E \subset \mathcal{T} \times \mathcal{T}$ provides these terms with a meaning. Let us define $E = \{\mathsf{sdec}(\mathsf{senc}(x, y), y) = x\}$ to account for the fact that decryption reverses encryption (for the same key). We can define an equivalence relation $=_E$ as the smallest equivalence relation containing E that is closed under the application of function symbols and substitution of variables by terms. Now, $\mathsf{sdec}(\mathsf{senc}(m, k), k) =_E m$.

The predicate set of the DY attacker has three types of facts: their knowledge of a term t, written as $\mathcal{K}(t)$, is derivable from the set of predicates Π_{A} seen or derived so

[1]The DY attacker and the DY library (but not the program) use the same terms. To simplify the presentation, we typeset them in *black italics*, as otherwise, they would be RedOrange, sans serif for the DY attacker *and OliveGreen, text italics* for the DY library.

$$\frac{\mathcal{K}(t) \in \Pi_{\mathrm{A}}}{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(t)}\ \mathrm{K}_0 \qquad \frac{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(t_1) \cdots \Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(t_n)\ \ f \in \mathcal{F}^n}{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(f(t_1, \ldots, t_n))}\ \textsc{App}$$

$$\frac{n \in \mathcal{N}_{\mathsf{pub}}}{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(n)}\ \textsc{Pub} \qquad \frac{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(t_1) \qquad \Pi_{\mathrm{A}} \vdash_{\mathrm{A}} t_1 \doteq t_2}{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(t_2)}\ \textsc{Subst}$$

$$\frac{t_1 =_E t_2}{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} t_1 \doteq t_2}\ \textsc{Eq} \qquad \frac{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(x) \qquad \Pi_{\mathrm{A}} \vdash_{\mathrm{A}} x \dot{\mapsto} t}{\Pi_{\mathrm{A}} \vdash_{\mathrm{A}} \mathcal{K}(t)}\ \textsc{Al-Subst}$$

Figure 2.1: The DY attacker's deduction rules.

far, two terms are considered equivalent $t_1 \doteq t_2$ according to $=_E$ and a name n is fresh $\mathsf{Fr}(n)$. The deduction relation of DY is defined in Fig. 2.1 and mostly describes how $\mathcal{K}(\cdot)$ is derived. Fig. 2.1 from top-left to bottom-right presents that (a) the attacker knows the messages it received and can apply function symbols, (b) if a name is public, the attacker knows this name, and if a term is known, any equivalent terms are also known, (c) equivalence modulo E translates into an equivalence judgment, and (d) any terms that correspond to a given symbol are known if the symbol itself is known. $t_1 \doteq t_2$ and $x \dot{\mapsto} t$ represent $=_E$ and $\mapsto$ (i.e., denotes the mapping of variables to terms) at the logical level, respectively. With the deduction relation in place, we can now define the transition relation (Fig. 2.2). Besides the symbol set Σ and the predicate set Π_{A}, the DY attacker is stateless, indicated by ϵ for the empty state.

A message is received by synchronization with the event $P2A(x)$ emitted by another component. As the DY adversary cannot process the incoming message type (e.g., bitstrings) directly, we must assume x is a symbol. Therefore, we set $\mathcal{V} = \mathcal{E}$. Hence, in P2A, x is determined by the environment (e.g., the sending component) and the attacker record the fact that it is known.

If the predicate set Π_{A} witnesses that the symbol x from the set Σ represents a value known to the DY attacker, the attacker can send x to another component (A2P). But not all knowledge predicates within Π_{A} are over symbols; encryption terms, for instance. Hence, the Alias rule can be used to introduce a new symbol, which can be transmitted. Recall that the Al-Subst rule in Fig. 2.1 introduces the required $\mathcal{K}(\cdot)$-predicate via the deduction relation. The transition rule Ded integrates the deduction relation. It is simply the minimal rule required to satisfy the condition in Def. 2.1.

Fresh names can be drawn by the attacker, but also by other components (see Fr-L2A in Fig. 2.3). Fr-A2L is a synchronous step between the DY attacker and other components that deals with the first case, where the attacker learns the name, which is marked as fresh and thus 'taken'. In the second case, another component, typically the crypto library of some party, picks a key (or another high-entropy value) that is marked as fresh. The DY attacker must also mark those names as 'taken', hence the other com-

$$\frac{\Pi_A \vdash_A \pi \qquad \Pi_A' = \Pi_A \cup \{\pi\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{\tau}_A (\Sigma, \Pi_A', \epsilon)} \textsc{Ded}$$

$$\frac{\mathcal{K}(x) \in \Pi_A \qquad x \in \Sigma}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{A2P(x)}_A (\Sigma, \Pi_A, \epsilon)} \text{A2P}$$

$$\frac{x \notin \Sigma \qquad \Pi_A' = \Pi_A \cup \{\mathcal{K}(x)\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{P2A(x)}_A (\Sigma \cup \{x\}, \Pi_A', \epsilon)} \text{P2A}$$

$$\frac{n \in \mathcal{N}_{\mathsf{priv}} \qquad \mathsf{Fr}(n) \notin \Pi_A \qquad \Pi_A' = \Pi_A \cup \{\mathsf{Fr}(n)\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{SFr(n)}_A (\Sigma, \Pi_A', \epsilon)} \textsc{Fr-L2A}$$

$$\frac{n \in \mathcal{N}_{\mathsf{priv}} \qquad \mathsf{Fr}(n) \notin \Pi_A \qquad \Pi_A' = \Pi_A \cup \{\mathsf{Fr}(n), \mathcal{K}(n)\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{Silent(n)}_A (\Sigma, \Pi_A', \epsilon)} \textsc{Fr-A2L}$$

$$\frac{x \notin \Sigma \quad f \in \mathcal{F}^n \quad \forall i \leq n : t_i \in \mathcal{T} \quad 0 < n \quad \Pi_A' = \Pi_A \cup \{x \dot{\mapsto} f(t_1, \ldots, t_n)\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{\mathsf{Alias}(x, f(t_1, \ldots, t_n))}_A (\Sigma \cup \{x\}, \Pi_A', \epsilon)} \textsc{Alias}$$

Figure 2.2: The transition relation rules for Dolev-Yao attacker model.

ponent synchronizes their picking of this value using the synchronous Fr-L2A rule in Fig. 2.3. This synchronization is necessary, as example 2.1 shows.

Example 2.1 (DY communicates with cryptographic library)**.** To generate a random number, a request needs to be sent to the library (e.g., rng). The library maintains a record of the generated random numbers within its predicate set (i.e., $\{\mathsf{Fr}(n)\}$) to ensure the creation of unique names. The DY attacker has the ability to choose a name (e.g., n') as long as it differs from the choice made by the library for the program (i.e., n).
The library's predicate set is updated using the synchronous Fr-L2A rule in Fig. 2.3 and $\mathsf{Fr}(n)$ is added to the predicate set of the attacker by the synchronous Fr-L2A rule in Fig. 2.2 for the library's initial random number generation. The second update is performed by the attacker using the synchronous Fr-A2L rule in Fig. 2.2 and the attacker is not able to pick the library chosen name n as $\mathsf{Fr}(n)$ exists in the attacker predicate set Π_A. Therefore, the attacker chooses a fresh name n', and the $\mathsf{Fr}(n')$ is

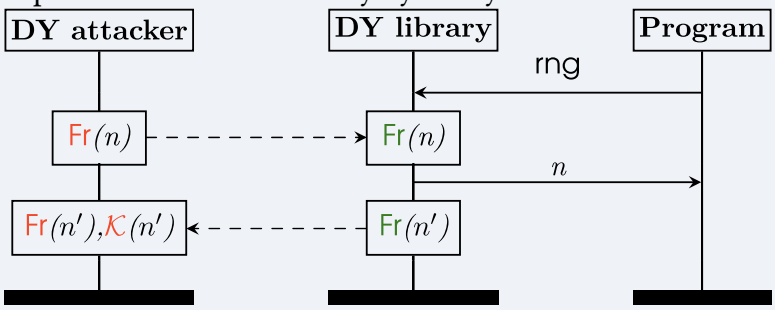
$$\frac{n \in \mathcal{N}_{\mathsf{priv}} \quad \mathsf{Fr}(n) \notin \Pi_L \quad \Pi_L' = \Pi_L \cup \{\mathsf{Fr}(n)\}}{(\Sigma, \Pi_L, \epsilon) \xrightarrow{SFr(n)}_L (\Sigma, \Pi_L', \epsilon)} \text{FR-L2A}$$

$$\frac{n \in \mathcal{N}_{\mathsf{priv}} \quad \mathsf{Fr}(n) \notin \Pi_L \quad \Pi_L' = \Pi_L \cup \{\mathsf{Fr}(n)\}}{(\Sigma, \Pi_L, \epsilon) \xrightarrow{Silent(n)}_L (\Sigma, \Pi_L', \epsilon)} \text{FR-A2L}$$

$$\frac{y \notin \Sigma \quad \forall i \leq n : x_i \in \Sigma \quad f \in \mathcal{F}^n \quad \Pi_L' = \Pi_L \cup \{y \mapsto f(x_1, \ldots, x_n)\}}{(\Sigma, \Pi_L, \epsilon) \xrightarrow{FCall(f, x_1, \ldots, x_n, y)}_L (\Sigma \cup \{y\}, \Pi_L', \epsilon)} \text{FCALL}$$

Figure 2.3: The transition relation rules for Dolev-Yao library model.

added into the predicate set of the library by the synchronous FR-A2L rule in Fig. 2.3.



In our examples, **solid** arrows represent direct communications between components, while **dashed** arrows denote the implicit flow of facts between the DY library and the DY attacker. Also, each step within the action box of components signifies the logical predicates added to their predicate sets during execution.

Observe that DY attackers do not pick honestly generated names (e.g., in example 2.1) due to the synchronization on freshness facts. The ALIAS rule (Fig. 2.2) only generates new symbols, not names (i.e., chooses x from the symbol set Σ). Contrast this with [161], where the syntactic structure of names binds them to protocol roles, including the attacker, or the follow-up work [13] where names do not need to carry structure, but a global restriction on traces is applied to ensure uniqueness. In both cases, this aspect of the DY attacker is thus hard-coded into the (global) trace model, which we can avoid.

2.3.3 Dolev-Yao Libraries

In addition to the DY attacker, it is necessary to model cryptographic operations. This involves abstracting a function's output with a term, such as $\mathsf{senc}(\cdot, \cdot)$ for encryption, to equip a programming language with DY semantics. One way is to integrate the term

Table 2.1: Summary of synchronizing events.

Event	Purpose	Involved components
FCall	Library calls	Program and Library
SFr	Calls to RNG	Program, Library and Attacker
A2P, *P2A*	Network communication	Program and Attacker
Silent	Ensure freshness	Library and Attacker

algebra into the predicate space $\mathcal{P}$ and mark cryptographic outputs via equalities, e.g., a logical predicate saying 'symbol z is equivalent to the DY term $\mathrm{senc}(x, y)$' (where x and y can be other symbols). A more generic way to achieve the same effect is by composing the function calls with a DY library that performs those abstraction steps. Fig. 2.3 shows how the composition can be done, with the FCall applying a function symbol similar to the App but including the Alias. Like the DY attacker, the DY library is stateless.

The deduction relation of the DY library is defined via an equivalence relation $=_E$. In verification tools like Sapic$^+$, the relation $=_E$ arises as the smallest congruence relation that is closed under substitutions and contains the set of equations E provided by the user. For the sake of the formalization, $=_E$ is an arbitrary equivalence relation. The equations E used in our case studies are provided in the input files for the protocol verifiers.

Table 2.1 summarizes the interface to the DY attacker and library from the perspective of a protocol component, which could be, for instance, a **BIR** program, as in our case studies. *FCall*, *SFr*, *A2P*, and *P2A* synchronize with the program component, whereas *Silent* is internal to the DY Libary and DY Attacker. For instance, if the protocol wanted to generate a random number, it would use *SFr*, which synchronizes with Fr-L2A in Fig. 2.2 and Fig. 2.3.

Chapter 3

Methodology

In this chapter, we present our framework for the composition of symbolic labeled transition systems, starting with discussing the limitations of translation methods and providing several illustrative examples to highlight these limitations better, which we compensate for by a novel form of parallel composition in a symbolic semantics. Also, we discuss our framework's capability to deal with other attackers alongside the DY attacker. Finally, we demonstrate the correctness of our approach and, for each theorem, provide access to proofs that we mechanized in HOL4.

3.1 Message Passing Limitations

Most recently, translation approach was used to leverage separation logic for the verification of network systems [161, 13, 12], and earlier to provide sound analyses for Dalvik bytecode [17]. The DY model is a model of cryptography where the attacker only makes deductions defined by a set of rules. It has been enormously successful in verifying security protocols, as it automates the verification procedure [23]. These rules do not necessarily cover all possible attacks and require additional justification [1, 90]. Typically, the DY attacker and the protocol share an unbounded set of *names* that represents keys and other hard-to-guess values. The model ensures the attacker and protocol always draw fresh names, hence key collisions are impossible. Names and public values can be combined with free function symbols to terms. E.g., $\mathrm{senc}(m, k)$ is a term that represents an encryption. It is not interpreted further. This so-called *term algebra* is complemented by a small set of rules that allows operations beyond the application of these symbols. E.g., a rule for decryption that says from $\mathrm{senc}(m, k)$ and k, the attacker learns m. When parallel composing a DY attacker with a language where keys and messages are represented as bitstrings, it is necessary to translate DY terms to bitstrings and vice versa. This, however, has several caveats.

3.1.1 Parsing Assumptions

First, it requires strong and unrealistic parsing assumptions to transform bitstrings back into terms that have more structure. For instance, keys must always be distinguished from bitstrings used elsewhere [12]. When we consider the space of AES keys, which (in reality) covers all bitstrings of length 128 (or 192 or 256), this requires (artificial) tagging to distinguish those from other bitstrings of that size, which real-world implementations do not have and actively avoid for performance. Another issue is the use of bitstring manipulation for message formatting.

Example 3.1 (Bitstring manipulation)**.** Concatenation is essential in the implementation of cryptographic protocols. It is associative and, hence, not easy to reason about automatically; thus, usually, this operation is not part of the DY term algebra. Consider the example where a message $\mathbf{m}$ is concatenated with its length to simplify parsing. Without further workarounds, the DY attacker can not determine $\mathbf{m}$ from $\mathsf{senc}(\mathbf{m}\|\mathsf{len}(\mathbf{m}), k)$, even if it possesses the encryption key k. As we show later, the DY attacker can derive $\mathbf{m}$ from $\mathbf{m}\|\mathsf{len}(\mathbf{m})$ by employing the deduction combinator $\vdash^{\mathsf{bit}}_{12}$ in Eq. bit defined in Sec. 3.3.3.

The translation approach supports message formatting, of course, otherwise it would be impractical. It works around this issue by modeling every message format that is used as a DY function symbol [13, Sec. 3.1]. For full TLS, there are at least 189 message formats [10, Sec. 5.1]. Clever refactoring may reduce this number (formats can be nested), but this is non-trivial and tedious. Most importantly, we would like our mechanism to be protocol-agnostic, even if it is tied to a particular set of cryptographic functions.

In contrast, techniques like DY^* and Comparse [37, 170] integrate bit-level and DY reasoning within the same tool, enabling the analysis of a (protocol-specific) set of message formats at the bit-level and then performing a DY analysis on abstract types. This avoids the problem and is discussed in Chapter 8.

3.1.2 Loss of Bit-level Information

Manipulating DY terms in the context of another language's semantics produces non-DY bitstrings that cannot be properly represented and translated back into their correct form. Therefore, these bitstrings become untraceable to their DY origins and an irreversible element to the transformation. As a result, translation approaches weaken the DY attacker in reasoning about the messages altered by a protocol party using a different language. E.g., say A wants to learn P's secret s and can trick P into encrypting $s+\mathbf{0x1}$ with a known key k. The DY attacker receives a bitstring corresponding to $\mathsf{senc}(s+\mathbf{0x1}, k)$, and after decrypting with k, has to recognize the transformation

applied to s+**0x1** (and that it requires subtracting **0x1**). Examining the huge number of possible transformations is out of the question, particularly when considering Turing-complete machine semantics (e.g., the wrappers in [150]). Typically, as bitstring addition + does not correspond to the image of a term constructor, such unknown bitstrings are translated into garbage DY terms [161] or terms we do not know [13, 12]. In contrast to message formats, this output was unintended. Using examples 3.2 and 3.3, we explain our solutions to this problem in Sec. 3.3.

3.1.3 Not Truly Versatile

The DY model is a symbolic abstraction that is well-accepted in protocol verification, but not throughout information security. It is useful to be able to replace DY attackers with computational attackers for flexibility or to validate the DY attacker's soundness. This is incompatible or difficult, depending on how the translation approach is realized. In [13, 12], a function translates from terms to bitstrings (i.e., the inverse direction to parsing discussed above). In the computational model, this relationship is not functional. For instance, a DY term representing a key, i.e., a *name*, may translate to many different bitstrings, depending on how they are sampled. Consequently, the computational attacker in these works is not an attacker in the traditional sense (an arbitrary probabilistic algorithm limited only in runtime) but the DY attacker inside a function translating from and to bitstrings.

Fortunately, a long line of work on computational soundness [1] explored requirements for such a translation, which must be probabilistic. Alas, known results come with a long list of requirements, both on programs and cryptographic primitives they use, that are hard to fulfill. To even formulate these requirements, the target semantics need to be equipped with a probabilism, non-determinism for communication and a notion of polynomial runtime in the length of some parameter that governs the key size and similar parameters. While there are methods to encode all of these, programming languages are rarely formalized with these features in mind. We can point to Aizatulin's Ph.D. thesis [7] as a case study for such a semantics and the required technical machinery.

3.2 Symbolic Parallel Composition

We now define a parallel composition that behaves like CSP-style asynchronous parallel composition but has an important twist: it is parametric in a *combined deduction relation*, which serves to transfer judgments from one system into the other. In the follow-up, we show that there are several ways to define this that increases the set of possible deductions and, thus, the precision of the analysis, while also being compati-

ble with almost all judgments made in programming languages. From hereon, we will combine different systems with oftentimes incompatible base types. To make it easier for the reader to type-check our statements, we will use colors to remark which system we speak of. Let $\restriction_i : 2^{\mathcal{P}_1 \uplus \mathcal{P}_2} \to 2^{\mathcal{P}_i}$ denote the projection to $i \in \{1, 2\}$. We present this using a disjoint union ($\uplus$) for familiarity and simpler presentation, while we employ a sum type in our HOL4 formalization. Then:

Definition 3.1 (Symbolic Parallel Composition)**.** Given two symbolic LTS $S_i = (\mathcal{E}, C_i, \mathbb{E}_i, \rightarrow_i, \mathcal{P}_i, \vdash_i)$, $i \in \{1, 2\}$ with *identical symbol space* $\mathcal{E}$ and a combined deduction relation $\vdash_{12} \subseteq 2^{(\mathcal{P}_1 \uplus \mathcal{P}_2)} \times (\mathcal{P}_1 \uplus \mathcal{P}_2)$, we define their symbolic parallel composition $S_1 \parallel^{\vdash_{12}} S_2$ as the symbolic LTS $(\mathcal{E}, C_1 \times C_2, \mathbb{E}_1 \cup \mathbb{E}_2, \rightarrow_{12}, \mathcal{P}_1 \uplus \mathcal{P}_2, \vdash_{12})$, where

- $\rightarrow_{12}$ moves asynchronously, i.e., either $(\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{o_1}_{12} (\Sigma', \Pi'_{12}, \mathbf{c'_1}, c_2)$ or $(\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{o_2}_{12} (\Sigma', \Pi'_{12}, \mathbf{c_1}, c'_2)$, if, for $i \in \{1, 2\}$, we can move with $o_i \in \mathbb{E}_i \setminus (\mathbb{E}_1 \cap \mathbb{E}_2)$, i.e., $(\Sigma, (\Pi_{12} \restriction_i), c_i) \xrightarrow{o_i}_i (\Sigma', (\Pi'_{12} \restriction_i), c'_i)$, keeping the complement's predicate set untouched $\Pi_{12} \restriction_{\bar{i}} = \Pi'_{12} \restriction_{\bar{i}}$, where $\bar{i} \in \{1, 2\}$ with $\bar{i} \neq i$, or
- $\rightarrow_{12}$ moves synchronously, i.e. $(\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{o}_{12} (\Sigma', \Pi'_{12}, \mathbf{c'_1}, c'_2)$, if, for $i \in \{1, 2\}$, $(\Sigma, (\Pi_{12} \restriction_i), c_i) \xrightarrow{o}_i (\Sigma'_i, (\Pi'_{12} \restriction_i), c'_i)$, $o \in \mathbb{E}_1 \cap \mathbb{E}_2$, and $\Sigma' = \Sigma'_1 \cup \Sigma'_2$.
- From the second condition of Def. 2.1 we have: $\Pi_{12} \vdash_{12} \varphi_{12} \implies (\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{\tau}_{12} (\Sigma, \Pi_{12} \cup \{\varphi_{12}\}, \mathbf{c_1}, c_2)$.

Def. 3.1 preserves fundamental properties of parallel composition like symmetry and associativity (see *Symmetry* and *Associativity* for the mechanized proofs in HOL4).

Note that even if $\vdash_{12}$ is empty (short: $\parallel^{\text{def}}_{=} \parallel^{\emptyset}$) the symbolic parallel composition is different from the classical parallel composition of the corresponding LTS. The symbol set is shared between both symbolic LTS even when they move asynchronously. Typically, symbolic LTS uses the symbol set to ensure that new symbols are fresh; as we use symbols for communication, we want to ensure they are globally fresh.

Moreover, if $\vdash_{12}$ is not empty, it allows deriving judgments in one system from judgments in the other system. Of course, we want to avoid this relation to be overly tied to one or the other system. Before we discuss how to do that, we will showcase how the DY model is represented as a symbolic LTS. This provides us with concrete examples to illustrate how $\vdash_{12}$ can overcome the issues from Sec. 3.1 (cf. examples 3.2 and 3.3 for their solutions).

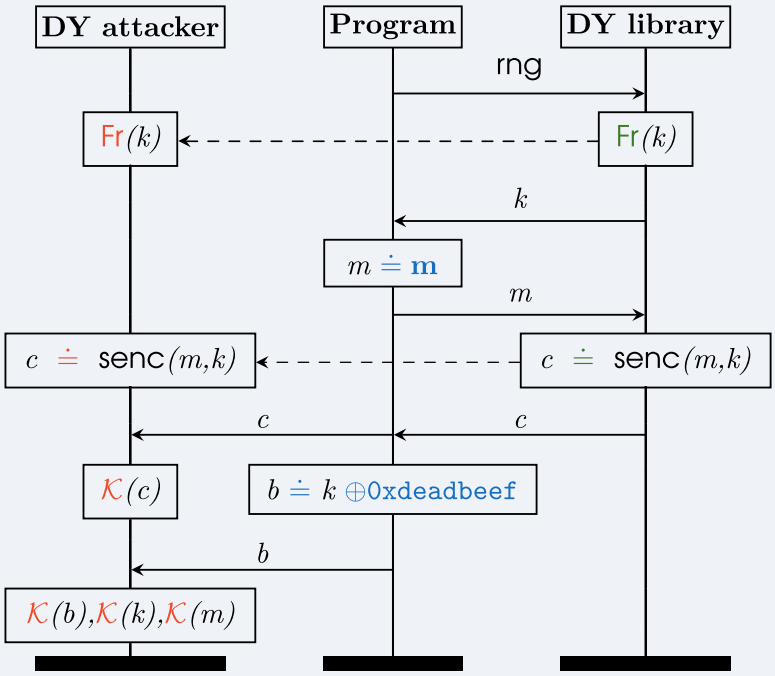
3.3 Deduction Combiners

Symbolic parallel composition's strength lies in its ability to transfer judgments between systems. There is a trade-off between precision and generality. We discuss some useful combiners from the most general to the most precise.

3.3.1 Generic Over- and Under Approximation

In general, bitstring operations can reveal cryptographic information. example 3.2 shows how to under-approximate or over-approximate the adversaries' capabilities on operating with bitstrings.

Example 3.2 (Masked encryption key)**.** In this example, the attacker obtains a message **m** encrypted with a fresh key k, followed by the key masked with a known constant `0xdeadbeef`. Using the combined deduction relation $\vdash^{\dot{\mapsto}}_{LA}$ (which we will define it the next section), the mapping $c \dot{\mapsto} \mathsf{senc}(m, k)$ transfers from DY library to DY attacker. The last message b ought to reveal the plaintext **m**. In the following, we introduce an *over-approximating* deduction combiner $\vdash^{\top}_{12}$ (Eq. over-approx) that allows the DY attacker to infer $\mathcal{K}(k)$ from $\mathcal{K}(b)$ and $b \doteq k \oplus$ `0xdeadbeef` and thus the plaintext **m** (from $\mathcal{K}(c)$, $c \dot{\mapsto} \mathsf{senc}(m, k)$ and $\mathcal{K}(k)$).



With an empty deduction combiner, the masked bitstring in the second network message is only accessible via the symbol b. The DY attacker can perform DY operations on the symbol b, but there is no way to access the k symbol without reasoning about the bitstring. Hence the empty deduction combiner under-approximates the adversaries' capabilities on operating with bitstrings. This is equivalent to the view in [161, 13, 12], where the concrete attacker is simply a translation function around the DY attacker. If a bitstring that cannot be parsed is encountered, it can only be ignored.

At the opposite end of the spectrum, Backes et al. [17] aimed for computational soundness, which entails that all attacks that could be mounted by a Turing machine

must be captured by the DY attacker. As the Turing machine can reverse the $\oplus$ operation in the above example, this required an over-approximation where all bitstring operations were represented in the DY model as *transparent* function symbols, i.e., function symbols whose input parameters are fully accessible.

We can generically represent this over-approximation in our framework, if we have an equality predicate $\doteq$ in the program's predicate set and we can identify the set of symbols that appear on either side, say, using a function named *symbols*:

$$\Pi_1 \uplus \Pi_2 \vdash^{\top}_{12} \mathcal{K}(z) \iff \exists\, x, y.\ \mathcal{K}(x) \in \Pi_2 \wedge x \doteq y \in \Pi_1 \wedge z \in \mathit{symbols}(y) \qquad \text{(over-approx)}$$

This over-approximation can introduce spurious equalities that lead to false attacks. For example, it is reasonable that a logic for bitstrings can conclude $a \doteq a \oplus x \oplus x$ for any a and x. This could easily introduce a spurious dependency between some a transmitted to the attacker and an arbitrary symbol x.

3.3.2 Sharing Equalities

If we can identify at least one equality predicate in both component's predicate space, however, we can find a much more useful middle ground between both extremes (i.e., over- and under approximation). Connecting equality judgments in both systems may allow tracking data flow across system boundaries, while requiring nothing more than to point out the equality predicates. This task could even be automated by (heuristically) identifying an equality as a predicate of arity two that is symmetric, reflexive and transitive.

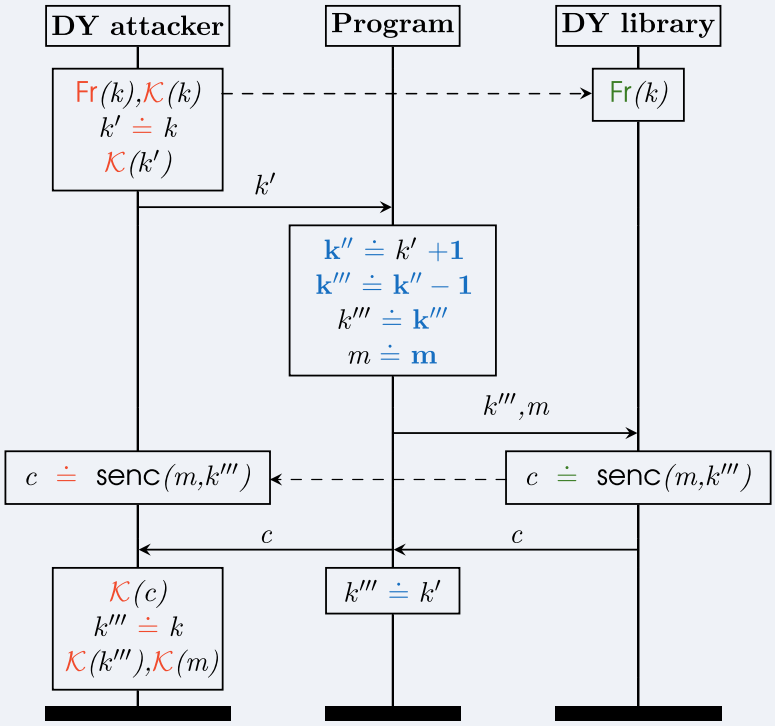
Let $\mathcal{P}_1$ and $\mathcal{P}_2$ contain atoms $\doteq$ and $\doteq$ such that symbols can appear on each side of either of them, i.e., $x \doteq_i y \in \mathcal{P}_i$ for $i \in \{1, 2\}$ and $x, y \in \Sigma_1 = \Sigma_2$. Then we can transfer equalities with the minimal deduction combiner defined by the following statements:

$$\Pi_1 \uplus \Pi_2 \vdash^{\mathrm{eq}}_{12} x \doteq z \iff \exists\, y.\ x \doteq y \in \Pi_1 \wedge y \doteq z \in \Pi_2 \qquad (\doteq)$$

$$\Pi_1 \uplus \Pi_2 \vdash^{\mathrm{eq}}_{12} x \doteq z \iff \exists\, y.\ x \doteq y \in \Pi_1 \wedge y \doteq z \in \Pi_2 \qquad (\doteq)$$

The following example shows how we address the *loss of bit-level information* discussed in Sec. 3.1.2, where the DY attacker could not analyze cryptographic secrets with bit-level modifications. Even though the DY attacker still cannot directly analyze bitstrings, they can now leverage the program's analysis by transferring equivalences through equality combiners (Eq. $\doteq$ and Eq. $\doteq$). This is essential when the protocol implementation involves packing (i.e., formatting messages so that the other party on the network can read them) and unpacking (i.e., extracting the message).

Example 3.3 (Transferable equalities). Equality can easily be transferred to accrue logical deduction relations. A $\doteq$ predicate can be added to the program's predicate set, e.g., similar to what will be discussed in Sec. 6.2.1. In the following procedure block, the deduction relation $\vdash_1$ is used to deduce $k''' \doteq k'$. Given $k''' \doteq k'$ and $k' \doteq k$, the attacker infers $k''' \doteq k$ using Eq. $\doteq$. Knowledge of k''' is derived from $\mathcal{K}(k)$ and $k''' \doteq k$ employing the SUBST rule in Fig. 2.1. Consequently, the attacker learns $\mathbf{m}$ by knowing k''', c, and $c \overset{\cdot}{\mapsto} \mathsf{senc}(m, k''')$.



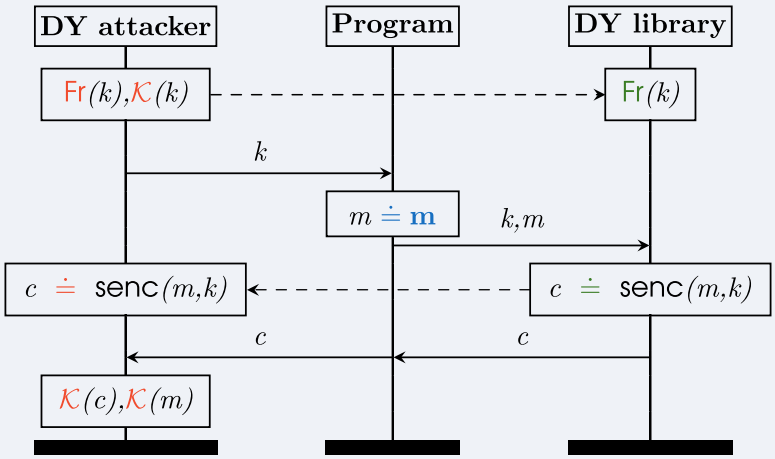
3.3.2.1 Sharing Mappings

Similar to equality sharing, we can also define other sharing predicates between components that have comparable predicates, for example, mapping. As we know the DY library and DY attacker and their respective predicate sets, we can use the combined deduction relation $\vdash^{\overset{\cdot}{\mapsto}}_{LA}$ to share the mapping predicates, as follows:

$$\Pi_L \uplus \Pi_A \vdash^{\overset{\cdot}{\mapsto}}_{LA} x \overset{\cdot}{\mapsto} t \iff x \overset{\cdot}{\mapsto} t \in \Pi_L$$
$$\Pi_L \uplus \Pi_A \vdash^{\overset{\cdot}{\mapsto}}_{LA} x \overset{\cdot}{\mapsto} t \iff x \overset{\cdot}{\mapsto} t \in \Pi_A \qquad (\vdash^{\overset{\cdot}{\mapsto}}_{LA})$$

Example 3.4 (Logical truth). Predicates can be shared between components without explicit communication. Thus, the DY attacker can uncover the message $\mathbf{m}$ using the known key k and the mapping $c \overset{\cdot}{\mapsto} \mathsf{senc}(m, k)$, without communicating with the

library. The steps to acquire the message $\mathbf{m}$ are as follows: upon receiving c from the program and obtaining $c \mathrel{\dot{\mapsto}} \mathsf{senc}(m,k)$ through $\vdash^{\dot{\mapsto}}_{LA}$, the attacker uses the AL-SUBST rule to get $\mathcal{K}(\mathsf{senc}(m,k))$. Next, the attacker utilizes their knowledge and $\mathsf{sdec} \in \mathcal{F}^2$ to learn $\mathsf{sdec}(\mathsf{senc}(m,k),k)$ using the APP rule in Fig. 2.1. Leveraging the relation $=_E$ detailed in Sec. 2.3.2, along with the EQ and SUBST rules (Fig. 2.1), the attacker obtains the knowledge of m. Without the mapping predicate linking the ciphertext and encryption term, the attacker would lack the necessary knowledge to apply the APP rule for decryption, leaving the encryption term undisclosed.



example 3.4 shows how the DY library, the attacker, and the program cooperate when the library generates a ciphertext using an adversarial key. As a nice extra, such a library allows us to prove a composition property that is convenient when different programs use multiple libraries (cf. Sec. 3.4.1).

3.3.3 Combined Reasoning

Equality sharing can transfer many statements derived from the other component into the predicate space of the DY attacker, but (a) only those that discuss the relation between term sent or deduced by the attacker (as only those have symbols), and, (b) only if the other component has sufficient information to derive an equality judgment.

Coming back to example 3.2, we see that the masking around the encryption key must be removed to deduce k from b. But as the program does not perform that operation, the necessary equality (between k and the potential result of such an operation) is not produced. The *ability* to perform this operation must be described via the $\mathcal{K}(\cdot)$ predicate rather than $\doteq$. A sound way of doing that would be to enhance the DY attacker with bitstring manipulation via constant values.

$$\Pi_1 \uplus \Pi_2 \vdash^{\mathsf{bit}}_{12} \mathcal{K}(x) \Leftrightarrow \exists\, y, c.\ \mathcal{K}(y) \in \Pi_2 \wedge y \doteq \mathsf{op}(x,c) \in \Pi_1 \wedge \mathsf{const}\ c \in \Pi_1 \quad \text{(bit)}$$

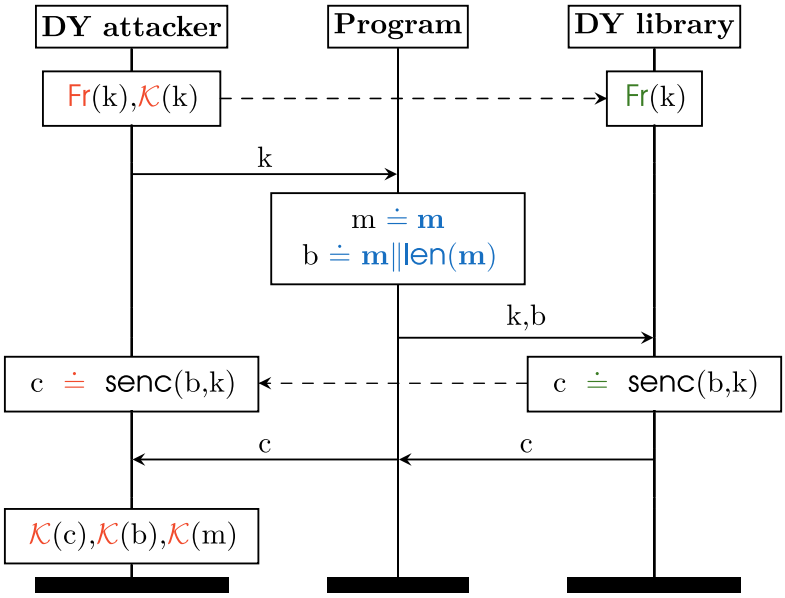


Figure 3.1: A DY attacker removing bit-level masking using $\vdash^{\text{bit}}_{12}$ in example 3.1

This combinator depends on the predicate space $\mathcal{P}_1$ providing a predicate const c that indicates a constant and needs to explicitly list all binary operators $\text{op}(x, c)$. It thus cannot be regarded as generic, although these concepts (operators and constants) should apply to many programming languages. Again recalling example 3.2, we can use $\vdash^{\text{bit}}_{12}$ to derive $\mathcal{K}(k)$, from $\mathcal{K}(b)$, $b \doteq k \oplus \texttt{0xdeadbeef}$ and const `0xdeadbeef`. Similarly, when we come back to example 3.1, we can see how $\vdash^{\text{bit}}_{12}$ helps the DY attacker derive $\mathcal{K}(m)$ in Fig. 3.1. As $\text{len}(m)$ is a constant and $\|$ is an operation applied to m and $\text{len}(m)$, the DY attacker obtains $\mathcal{K}(m)$ from $\mathcal{K}(b)$, $b \doteq m \| \text{len}(m)$ and const $\text{len}(m)$. We have now addressed the issue of parsing assumptions (Sec. 3.1.1) in example 3.1 and the loss of bit-level information (Sec. 3.1.2) in example 3.2.

In summary, the symbolic view on composition improves the accuracy of judgment in particular when combining with the DY attacker as examples 3.1 to 3.3 witness. This is hardly surprising, as the translation approach sets up both DY attacker and program in a concrete execution semantics with concrete (classical) composition, although the DY attacker is symbolic in nature. By instead lifting the language to the symbolic level, we turn the composition approach back on its feet and observe—at the level of the composed system—that we have two methods of deduction at our disposal. What is surprising, is that we can achieve a significant improvement with relatively simple deduction combinators. It should be difficult to find logics where one *cannot* find an equality predicate. Even a closer integration as sketched in the previous paragraph, would apply to a large set of programming languages while yielding immediate benefits.

3.4 Correctness

The correctness of parallel composition ($\|$) is defined in terms of a partially synchronized interleaving ($|||$) of the traces of each component, i.e., a permutation of the union of trace sets that maintains the relative order of elements within each set. Partially synchronized interleaving on traces generalizes interleaving composition by requiring certain actions from two or more components to occur in a specific relative order that signifies synchronization points. Conversely, other actions remain unconstrained and may be interleaved in any arbitrary manner. This is stronger than trace inclusion, as it implies that all non-synchronizing traces of $\mathfrak{T}(\mathbf{M})$ or $\mathfrak{T}(M)$ are contained in $\mathfrak{T}(\mathbf{M}) \,|||\, \mathfrak{T}(M)$, where $\mathfrak{T}(M)$ is the set of traces produced by an LTS M and a trace $\mathtt{t} \in \mathfrak{T}$ is a sequence of events.

Definition 3.2 (Partially Synchronized Interleaving on Traces)**.** For any LTS $\mathbf{M}$ and M, and two sets of traces produced by these LTS, respectively, $\mathfrak{T}(\mathbf{M})$ and $\mathfrak{T}(M)$, the Partially Synchronized Interleaving on Traces $\mathfrak{T}(\mathbf{M}) \,|||\, \mathfrak{T}(M)$ is the set of all possible traces $\mathscr{T}$ such that:

- $\mathscr{T}$ is a permutation of $\mathfrak{T}(\mathbf{M}) \cup \mathfrak{T}(M)$.
- The relative order of elements in $\mathfrak{T}(\mathbf{M})$ and $\mathfrak{T}(M)$ are preserved in $\mathscr{T}$:
  - For all traces $\mathtt{t} \in \mathfrak{T}(\mathbf{M})$ and $\mathtt{t} \in \mathscr{T}$, i, j, m, and n such that $0 \leq i < j < m$, there exist k and l such that $0 \leq k < l < m + n$, $\mathtt{t}[k] = \mathtt{t}[\mathbf{i}]$ and $\mathtt{t}[l] = \mathtt{t}[\mathbf{j}]$.
  - For all traces $\mathtt{t} \in \mathfrak{T}(M)$ and $\mathtt{t} \in \mathscr{T}$, x, y, m, and n such that $0 \leq x < y < n$, there exist z and d such that $0 \leq z < d < m+n$, $\mathtt{t}[z] = \mathtt{t}[\mathrm{x}]$ and $\mathtt{t}[d] = \mathtt{t}[\mathrm{y}]$.
- For all traces $\mathtt{t} \in \mathfrak{T}(\mathbf{M})$, $\mathtt{t} \in \mathfrak{T}(M)$ and $\mathtt{t} \in \mathscr{T}$, i, j, m, and n such that $0 \leq i < m$, $0 \leq j < n$, and $\mathtt{t}[\mathbf{i}] = \mathtt{t}[\mathrm{j}]$, there exists a k such that $0 \leq k < m + n$ and $\mathtt{t}[k] = \mathtt{t}[\mathbf{i}] = \mathtt{t}[\mathrm{j}]$.

Moreover, Def. 3.3 defines when adding a predicate can activate a transition in our system.

Definition 3.3 (Transition Enabling)**.** Given two symbolic LTS $S_i = (\mathcal{E}, C_i, \mathbb{E}_i, \rightarrow_i, \mathcal{P}_i, \vdash_i)$, $i \in \{\mathbf{1}, 2\}$, their symbolic parallel composition $\mathbf{S_1} \;\|^{\vdash_{12}}\; S_2 = (\mathcal{E}, C_1 \times C_2, \mathbb{E}_1 \cup \mathbb{E}_2, \rightarrow_{12}, \mathcal{P}_1 \uplus \mathcal{P}_2, \vdash_{12})$, a predicate set $\Pi_{12} \in 2^{(\mathcal{P}_1 \uplus \mathcal{P}_2)}$ and a predicate $\varphi_{12} \in (\mathcal{P}_1 \uplus \mathcal{P}_2)$, such that $\Pi_{12} \vdash_{12} \varphi_{12}$, we say the predicate φ_{12} enables the transition $\rightarrow_{12}$ if:

- Either $(\Sigma, \Pi_{12} \cup \{\varphi_{12}\}, \mathbf{c_1}, c_2) \xrightarrow{o_1}_{12} (\Sigma', \Pi'_{12} \cup \{\varphi_{12}\}, \mathbf{c'_1}, c_2)$ or $(\Sigma, \Pi_{12} \cup \{\varphi_{12}\}, \mathbf{c_1}, c_2) \xrightarrow{o_2}_{12} (\Sigma', \Pi'_{12} \cup \{\varphi_{12}\}, \mathbf{c_1}, c'_2)$, and, without adding the predicate φ_{12}, it is not possible to move with $o_i \in \mathbb{E}_i \setminus (\mathbb{E}_1 \cap \mathbb{E}_2)$ for $i \in \{\mathbf{1}, 2\}$, i.e., $(\Sigma, (\Pi_{12} \downharpoonright_i), c_i) \xrightarrow{o_i}_i (\Sigma', (\Pi'_{12} \downharpoonright_i), c'_i)$, keeping the complement's predicate set untouched $\Pi_{12} \downharpoonright_{\bar{i}} = \Pi'_{12} \downharpoonright_{\bar{i}}$,

- Or $(\Sigma, \Pi_{12} \cup \{\varphi_{12}\}, \mathbf{c_1}, \mathbf{c_2}) \xrightarrow{o}_{12} (\Sigma', \Pi'_{12} \cup \{\varphi_{12}\}, \mathbf{c'_1}, \mathbf{c'_2})$, and, $(\Sigma, (\Pi_{12} \downharpoonright_i), c_i) \xrightarrow{o}_i (\Sigma'_i, (\Pi'_{12} \downharpoonright_i), c'_i)$ is not possible without adding the predicate φ_{12}, for $i \in \{1, 2\}$, $o \in \mathbb{E}_1 \cap \mathbb{E}_2$, and $\Sigma' = \Sigma'_1 \cup \Sigma'_2$.

Adding predicates may also disable transitions within the system. The definition for when adding a predicate disables transitions is similar to Def. 3.3 and obtained by negating logical entailment.

The correctness result covers *all* events, including synchronizing events for the DY attacker, the DY library and non-synchronizing events that occur only in the program we translate. Verification methodology will typically only consider a specific subset. In our case studies, for instance, the program emits non-synchronizing events when special functions are reached and the verification tool describes security properties as trace properties over these events.

We denote the symbolic parallel composition by $\|^{\cdot}_s$ and traditional parallel composition for concrete systems by $\|_c$. To avoid any ambiguity, we use notation like $\mathbf{t_{12}} \in \mathfrak{T}_{12}(\mathbf{M} \parallel \mathrm{M})$ to refer to the sequence of events produced by a composite system and $\mathfrak{T}^s$ to distinguish the set of symbolic semantics traces from the set of concrete semantics traces $\mathfrak{T}^c$.

Theorem 3.1 (Symbolic Composition Correctness)**.** *For any symbolic LTS* $\mathbf{M}$ *and* M*, and for any combined deduction relation* $\vdash_{12}$*:*

1. *If all predicates* $\vdash^{\mathsf{ena}}_{12}$ *produces may enable additional transitions, but not disable them, we call* $\vdash^{\mathsf{ena}}_{12}$ enabling *and* $\mathfrak{T}^s_{12}(\mathbf{M} \|^{\vdash^{\mathsf{ena}}_{12}}_s \mathrm{M}) \supseteq \mathfrak{T}^s(\mathbf{M}) \,|||\, \mathfrak{T}^s(\mathrm{M})$.
2. *If all predicates* $\vdash^{\mathsf{dis}}_{12}$ *produces may disable transitions, but never enable new transitions, we call* $\vdash^{\mathsf{dis}}_{12}$ disabling *and* $\mathfrak{T}^s_{12}(\mathbf{M} \|^{\vdash^{\mathsf{dis}}_{12}}_s \mathrm{M}) \subseteq \mathfrak{T}^s(\mathbf{M}) \,|||\, \mathfrak{T}^s(\mathrm{M})$.

Proof. By induction over the length of the composed trace. The base case is trivial (no step is taken). The inductive case is proved by a case distinction over synchronous and asynchronous events. *Correctness-Enable* and *Correctness-Disable* mechanize the proof of Theorem 3.1's cases in HOL4. □

Theorem 3.1 enables compositional analysis of symbolic systems, as Theorem 3.2 shows. Let refinement (or security) be expressed in terms of trace inclusion. Then, if component $\mathbf{M_1}$ refines $\mathbf{M_2}$, written in the same language, and the same holds for components $\mathrm{M_1}$ and $\mathrm{M_2}$, then the combined system $\mathbf{M_1} \|^{\vdash_{12}}_s \mathrm{M_1}$ refines $\mathbf{M_2} \|^{\vdash_{12}}_s \mathrm{M_2}$.

Lemma 3.2 (Symbolic Compositional Trace Inclusion)**.** *Let* $\mathfrak{T}^s_1$*,* $\mathfrak{T}^s_2$*,* $\mathfrak{T}^s_1$*, and* $\mathfrak{T}^s_2$ *be the sets of traces produced by any symbolic LTS* $\mathbf{M_1}$*,* $\mathbf{M_2}$*,* $\mathrm{M_1}$*, and* $\mathrm{M_2}$*. If* $\mathfrak{T}^s_1 \subseteq \mathfrak{T}^s_2$ *and* $\mathfrak{T}^s_1 \subseteq \mathfrak{T}^s_2$*, then:*

- *For the empty combined deduction relation* $\emptyset$*, it holds that* $\mathfrak{T}^s_{12}(\mathbf{M_1} \parallel_s M_1) \subseteq \mathfrak{T}^s_{12}(\mathbf{M_2} \parallel_s M_2)$.
- *For any combined deduction relation* $\vdash_{12} \in \{\vdash^{\top}_{12}, \vdash^{\mathsf{eq}}_{12}, \vdash^{\mathsf{bit}}_{12}\}$ *(defined in Sec.3.3), it holds that* $\mathfrak{T}^s_{12}(\mathbf{M_1} \parallel^{\vdash_{12}}_s M_1) \subseteq \mathfrak{T}^s_{12}(\mathbf{M_2} \parallel^{\vdash_{12}}_s M_2)$.
- *For any disabling combined deduction relation* $\vdash^{\mathsf{dis}}_{12}$ *on the refined system (left) and any enabling combined deduction relation* $\vdash^{\mathsf{ena}}_{12}$ *on the abstract system (right), it holds that* $\mathfrak{T}^s_{12}(\mathbf{M_1} \parallel^{\vdash^{\mathsf{dis}}_{12}}_s M_1) \subseteq \mathfrak{T}^s_{12}(\mathbf{M_2} \parallel^{\vdash^{\mathsf{ena}}_{12}}_s M_2)$.

In the following, we instantiate Theorem 3.1 to enable merging and splitting DY libraries containing the same or distinct function signatures, as protocol parties often utilize different implementations for cryptographic libraries.

3.4.1 Composing and Decomposing DY Libraries

Protocol parties are often implemented in different languages that potentially incorporate different implementations of the same cryptographic library. Additionally, each party may employ additional libraries tailored to their specific needs, which could differ from those used by others. Therefore, our framework needs to account for both scenarios in the composition of protocol participants. We use function symbols, which represent cryptographic operations, to distinguish between the two scenarios where DY libraries have identical or distinct function symbols. We introduce the following corollary—and mechanize its proof in HOL4 to enable the composition or decomposition of DY libraries. See *Same-Signature* and *Distinct-Signatures* for the proof of Theorem 3.3.

Corollary 3.3. *For all DY libraries* $\mathrm{DY}\textsc{lib}_{\mathcal{F}_1}$ *and* $\mathrm{DY}\textsc{lib}_{\mathcal{F}_2}$*, where* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *can be the same or distinct function signatures, we have that* $\mathfrak{T}^s_{12}(\mathrm{DY}\textsc{lib}_{\mathcal{F}_1} \parallel_s \mathrm{DY}\textsc{lib}_{\mathcal{F}_2}) = \mathfrak{T}(\mathrm{DY}\textsc{lib}_{\mathcal{F}_1}) ||| \mathfrak{T}(\mathrm{DY}\textsc{lib}_{\mathcal{F}_2})$.

Theorem 3.3 serves not only in the composition but also in the decomposition of a single DY library. This allows us to break down a DY library, containing function symbols, into the composition of two DY libraries, each with either the exact same signature or distinct signatures. Consequently, each protocol participant's library can be decomposed into two parts, such as $\mathrm{DY}\textsc{lib}_{\mathcal{F}} \parallel_s \mathrm{DY}\textsc{lib}_{\mathcal{F}}$ or $\mathrm{DY}\textsc{lib}_{\mathcal{F}} \parallel_s \mathrm{DY}\textsc{lib}_{\mathcal{F}}$.

Following this line of reasoning, when composing multiple parties, it becomes possible to independently compose each part of each participant's library (i.e., $\mathrm{DY}\textsc{lib}_{\mathcal{F}} \parallel_s \mathrm{DY}\textsc{lib}_{\mathcal{F}}$ for the common and $\mathrm{DY}\textsc{lib}_{\mathcal{F}} \parallel_s \mathrm{DY}\textsc{lib}_{\mathcal{F}}$ for the remainder). Now the common part can be merged into one ($\mathrm{DY}\textsc{lib}_{\mathcal{F}}$). For more details about the application of Theorem 3.3 in one of our case studies, see Sec. 6.2.4.

3.5 Refinement

While Theorem 3.1 and Theorem 3.2 are used throughout the instantiation of our framework in Sec. 6.2.3, we need an additional theorem to carry the analysis to the concrete system semantics. This follows from the fact that both theorems only make statements about symbolic semantics traces ($\mathfrak{T}^s$) instead of concrete semantics traces ($\mathfrak{T}^c$). We, thus, need to relate the two.

Symbolic execution semantics are usually defined sound using a refinement relation, which we denote as $\sqsubseteq$. To define it, we have to assume that we have a way to apply a (component-specific) *interpretation function*, i.e., a function ι from symbolic variables to concrete values (e.g., the function H utilized in [115]), to a symbolic trace. E.g., let *apply* denote this application, $\mathbf{t}^c \sqsubseteq \mathbf{t}^s \Leftrightarrow \exists \iota.\ \mathbf{t}^c = \mathit{apply}(\mathbf{t}^s, \iota)$ and likewise for $\sqsubseteq$. We define how this refinement applies to the composed system. Let $\downarrow_i : 2^{\mathbb{E}_1 \cup \mathbb{E}_2} \to 2^{\mathbb{E}_i}$ denotes the trace projection to $i \in \{1, 2\}$, then:

Definition 3.4 (Composed System Refinement)**.** Let t^c_{12} be a concrete composed trace and t^s_{12} be a symbolic composed trace, then a refinement relation between these two traces is $\mathrm{t}^c_{12} \sqsubseteq \mathrm{t}^s_{12}$ such that there exist interpretation functions ι_1 and ι_2 where

- $\mathrm{t}^c_{12}{\downarrow_1} = \mathit{apply}(\mathrm{t}^s_{12}{\downarrow_1}, \iota_1)$
- $\mathrm{t}^c_{12}{\downarrow_2} = \mathit{apply}(\mathrm{t}^s_{12}{\downarrow_2}, \iota_2)$

With this notation, we describe how refinement transfers to the composed system.

Theorem 3.4 (Refinement)**.** *For any enabling combined deduction relation* $\vdash^{\mathsf{ena}}_{12}$*, any concrete LTS* $\mathbf{M_c}$ *and* $\mathrm{M_c}$*, any symbolic LTS* $\mathbf{M_s}$ *and* $\mathrm{M_s}$*, we have*

$$\frac{\mathfrak{T}^c(\mathbf{M_c}) \sqsubseteq \mathfrak{T}^s(\mathbf{M_s}) \qquad \mathfrak{T}^c(\mathrm{M_c}) \sqsubseteq \mathfrak{T}^s(\mathrm{M_s})}{\mathfrak{T}^c_{12}(\mathbf{M_c} \parallel_c \mathrm{M_c}) \sqsubseteq \mathfrak{T}^s_{12}(\mathbf{M_s} \parallel_s^{\vdash^{\mathsf{ena}}_{12}} \mathrm{M_s})}$$

In Theorem 3.4, the enabling deduction relation is to ensure broader coverage of behaviors during symbolic execution compared to concrete execution.

Proof. From the left-hand side, we apply a concrete variant of Theorem 3.1 (cf. Theorem 3.5) to describe the composed concrete system via interleaving. From the right-hand side, we apply Theorem 3.1 (case 1) itself, to obtain a similar interleaving, but of the composed symbolic system. We then use the refinements $\sqsubseteq$, $\sqsubseteq$, and $\sqsubseteq$ and instantiate the interpretation functions in $\sqsubseteq$ to map the composed traces in the concrete domain to those in the symbolic domain. See *Refinement* for proof mechanization in HOL4. □

We avoid communication between symbolic and concrete components by avoiding hybrid systems altogether, which is why we need both Theorem 3.2 (for abstraction within the symbolic domain) and Theorem 3.4 (for abstraction from the concrete to the symbolic domain).

The reader may wonder if the interpretation function ι, used in the definition of $\sqsubseteq$, would also constitute a translation of the kind we criticized. We criticize the implication of a translation at the object level, i.e., translation *within* the system between concrete programs and symbolic DY attackers *in the same system*. By contrast, the interpretation function resides at the proof level rather than the object level, and (the existence of it) is merely a constraint that the symbolic execution is a consistent abstraction. Concretely, it can be created on the fly and it is not required to be computable or consistent across multiple (symbolic) executions.

3.5.1 Concrete World

In the CSP-style parallel composition of concrete labeled transition systems, synchronization and communication enable interaction among sub-components in a composed system. A correspondence can be established between traces of a composed system using CSP-style asynchronous parallel composition and the interleaving of traces of each sub-component.

Theorem 3.5 (Concrete Composition Correctness)**.** *For any concrete LTS* $\mathbf{M}$ *and* M*, we have* $\mathfrak{T}^c_{12}(\mathbf{M} \parallel_c \mathsf{M}) = \mathfrak{T}^c(\mathbf{M}) \mid\mid\mid \mathfrak{T}^c(\mathsf{M})$.

Proof. The goal is to show that for all traces of the composition of concrete LTS, there is an equivalent trace resulting from interleaving the traces of each concrete LTS and vice versa. We prove the theorem using induction over the length of the composed traces. Considering no steps were undertaken, the base case is straightforward. For the inductive case, we utilize case distinction over synchronous and asynchronous events. □

Theorem 3.5 enables compositional analysis, as evidenced by the following corollary, wherein individual components can be refined while preserving trace inclusion for the composed system.

Corollary 3.6 (Concrete Compositional Trace Inclusion)**.** *For any concrete LTS* $\mathbf{M}_1$, $\mathbf{M}_2$, M_1, *and* M_2, *we have*

$$\frac{\mathfrak{T}^c(\mathbf{M}_1) \subseteq \mathfrak{T}^c(\mathbf{M}_2) \qquad \mathfrak{T}^c(\mathsf{M}_1) \subseteq \mathfrak{T}^c(\mathsf{M}_2)}{\mathfrak{T}^c_{12}(\mathbf{M}_1 \parallel_c \mathsf{M}_1) \subseteq \mathfrak{T}^c_{12}(\mathbf{M}_2 \parallel_c \mathsf{M}_2)}$$

Similar results were previously established for CSP-style asynchronous parallel composition of concrete systems (see, e.g., [157]), but we have formalized and proven

them on top of HOL4. Complete mechanized proofs are available at *Concrete-Composition* and *Concrete-Trace-Inclusion*.

3.6 Beyond Dolev-Yao Attackers

Besides the DY model, which is used in protocol verification, there are two other attacker models that we want to discuss in the context of this framework. The first is the *unbounded attacker* used in programming languages and system-level verification. This attacker is used in settings where cryptographic primitives are either not used at all, or where their security guarantees are built into the language semantics [114]. The unbounded attacker can be a program or program context in the same language as the program under verification, or the trace of inputs that the program interacts with. In both cases, computational limitations (even decidability) are rarely relevant to the security argument. The decoupling of the attacker is thus only interesting if the attacker is intended to communicate with multiple other components. In this case, there is no need for deduction by the unbounded attacker (each of its inputs is arbitrary, so fresh symbols) but deduction combiners can be useful for components that share information, e.g., via the attacker.

The second attacker model is the *computational attacker*, explained in Sec. 2.3.1. There, we discussed how the translation approach struggles with probabilistic choice, unless the language provides the means to draw random keys. A naive formulation of the computational attacker encodes the Turing machine semantics or any other probabilistic semantics. E.g., when a key is drawn, there are approximately 2^n possible next configurations, with n being the key length, each describing a different value of this key after sampling. It is clear that such a modeling has little use for verification, as the configuration space is enormous.

Instead, we can apply our previous argument that a symbolic semantics for the program ought to be composed with a symbolic semantics for the attacker and library. Thus, we should find a symbolic representation of the random process producing, e.g., a distribution over keys. Bana and Comon propose a model where symbolic rules with a computational interpretation are individually proven sound, but can be used to reason symbolically [22, 21]. It can be reasoned about interactively with the Squirrel prover [19]. We only sketch the idea here and leave a full realization for future work. As for the DY attacker, the computationally-complete symbolic attacker (CCSA) represents messages as terms over a set of function symbols and names, however, they are interpreted w.r.t. a security parameter. A name describes the process of sampling a random bitstring. A term describes a recursive process of evaluating each function symbol using some polynomial algorithm and sampling each name as described (but only once). In contrast to the DY model, where the function symbols define what the

attacker can do (and everything else is disallowed), the CCSA model retains compatibility with the computational model by symbolically formulating what the attacker *cannot do* (and everything else is allowed). Consequently, there is no equational theory; equality is evaluated literally on the resulting bitstrings (in the interpretation). Instead, CCSA features axioms that are proven sound w.r.t. the above mentioned interpretation of terms as probabilistic polynomial-time Turing machines. The CCSA is thus simpler to define than the DY attacker: it does not retain a predicate set or an equality predicate. The predicate set, however, is the first-order logic described by Bana and Comon [22]. Scerri's decision procedure allows handling a fragment of these formulas [151], hence there is even potential for automation.

Part II

Binary Analysis of Protocols Implementations

Chapter 4

Preliminaries

In this part of the thesis, we introduce CryptoBap, a binary analysis framework that extends security protocol verification to machine code, thereby eliminating the need to trust compilers. CryptoBap extends the HolBA framework to verify ARMv8 and RISC-V machine code of cryptographic protocols (Sec. 4.1). It achieves this by extracting formal models of the protocol under analysis in two distinct modified versions of the applied pi calculus: one suitable for automated verification with ProVerif and CryptoVerif, using the Csec-modex toolchain (Sec. 4.2), and another for verification with ProVerif, Tamarin, and DeepSec, using the Sapic$^+$ toolchain (Sec. 4.3). Besides ensuring functional correctness, cryptographic protocols need to be examined for side-channel leakage. To address this, we introduce side channels and observational models (Sec. 4.4), which provide a foundation for analyzing such leakages. Moreover, we introduce several running examples in Sec. 4.5, which will reappear throughout the thesis to demonstrate how abstract notions map to concrete protocol behaviors.

In this chapter, we present the necessary preliminaries required to understand the CryptoBap structure and our contributions to this part of the thesis. Readers primarily interested in binary-level protocol verification may proceed directly to this part; however, many of the definitions and proof techniques here instantiate the general symbolic parallel composition framework developed in Part I, particularly the handling of heterogeneous components and deduction combiners introduced in Chapter 3.

4.1 HolBA Framework and BIR

CryptoBap relies on HolBA [116] to transpile the binary of protocols to the **BIR**[1] representation. **BIR** is a simple and architecture-agnostic language used as the internal language of HolBA and is designed to simplify the binary analysis of programs and

[1]We depict **BIR** in **RoyalBlue, bold roman**, SBIR in Emerald, roman, `IML` in `Plum, typewriter` and Sapic$^+$ in RedOrange, sans serif. Elements common to all languages are typeset in black, italic.

$$
\begin{aligned}
\mathbf{P} \in \mathbf{prog} &:= \mathbf{block}^* \\
\mathbf{block} &:= (\mathbf{v}, \mathbf{stmt}^*) \\
\mathbf{v} \in \mathbf{Bval} &:= \mathit{string} \mid \mathit{int} \\
\mathbf{stmt} &:= \mathbf{halt} \mid \mathbf{jmp}(\mathbf{e}) \mid \mathbf{cjmp}(\mathbf{e}, \mathbf{e}, \mathbf{e}) \\
&\quad \mid \mathbf{assign}(\mathit{string}, \mathbf{e}) \mid \mathbf{assert}(\mathbf{e}) \\
\mathbf{e} \in \mathbf{Bexp} &:= \mathbf{v} \mid \diamond_{\mathbf{u}}\ \mathbf{e} \mid \mathbf{e}\ \diamond_{\mathbf{b}}\ \mathbf{e} \mid \mathbf{var}\ \mathit{string} \\
&\quad \mid \mathbf{ifthenelse}(\mathbf{e}, \mathbf{e}, \mathbf{e}) \mid \mathbf{load}(\mathbf{e}, \mathbf{e}, \mathit{int}) \mid \mathbf{store}(\mathbf{e}, \mathbf{e}, \mathbf{e}, \mathit{int})
\end{aligned}
$$

Figure 4.1: BIR's syntax

facilitate building analysis tools. HolBA is proof-producing and ensures that the transpilation preserves the semantics of the binary.

Fig. 4.1 depicts BIR's syntax. A BIR program P consists of uniquely labeled blocks, with each block containing a sequence of statements. Labels correspond to specific locations in the program and are commonly used as the target for jump instructions. BIR statements include (a) assign, to assign a BIR expression to a variable, (b) jumps (i.e., jmp or cjmp), (c) halt, which serves as the termination instruction, and (d) assert, which evaluates a boolean expression and terminates execution if the assertion fails. Expressions in BIR include constants, variables, conditionals (i.e. ifthenelse), arithmetic operations, denoted by $\diamond_{\mathbf{b}}$ for binary and $\diamond_{\mathbf{u}}$ for unary operations, as well as memory operations such as load and store.

A BIR state $(\eta^b, \mathbf{pc}) \in \mathbf{S}^b$ consists of an environment $\eta^b : \mathbf{Bvar} \mapsto \mathbf{Bval}$ which maps variables, i.e., registers r_i and memory locations *Mem*, to values and a program counter pc that holds the label of the executing BIR block. The relation $\rightarrow \subseteq \mathbf{S}^b \times \mathbf{S}^b$ models the execution of a BIR block. The execution of n steps is denoted by $\rightarrow^*$ if $n \geq 0$ and $\rightarrow^+$ or $\rightarrow^n$, if $n > 0$. We write $s_i^b \rightarrow_L^n s_j^b$ to restrict the transition from s_i^b to s_j^b to the label set L.

Fig. 4.7 presents a BIR snippet for the running example.

4.1.1 Vanilla Symbolic Execution

HolBA provides a proof-producing symbolic execution for BIR [115] which CryptoBap uses in its verification pipeline. This symbolic execution formalizes the symbolic generalization of BIR (hereafter SBIR) to explore all execution paths of the program. The symbolic semantics align with concrete semantics, enabling guided execution that maintains a set of reachable states arising from an initial symbolic state.

HolBA's symbolic execution allows for verifying functional correctness, but not (directly) protocol security, as it lacks a suitable attacker model and concurrent behav-

ior. CRYPTOBAP bridges this gap by extracting formal models of the protocols from their implementations.

The symbolic semantics is bisimilar to the concrete one and allows guiding the execution while maintaining a sound set of reachable states from an initial symbolic state (we call this the *symbolic execution structure*). To generalize from **BIR** to SBIR, symbolic expressions SE are defined that can be interpreted to **BIR** values **Bval** via an interpretation $H : SE \to \mathbf{Bval}$. In addition to the symbolic environment $\eta^s : \mathbf{Bvar} \mapsto SE$, the SBIR state $(\Pi^s, \eta^s, \mathbf{pc}) \in S^s$ also contains a path condition $\Pi^s \in SE$ and a **pc** that is kept concrete to obtain a concrete control flow.

Let $\twoheadrightarrow : S^s \times S^s$ be the single-step transition relation of SBIR and $\twoheadrightarrow^n$ (or $\twoheadrightarrow^+$) denote a multi-step symbolic transition. For the HolBA's vanilla symbolic execution, Lindner et al. [115] proved that a single SBIR execution step soundly matches a single **BIR** execution step, characterized by the following simulation theorem:

Property 1. For all s_i^b, s_j^b, H, s_i^s s.t. $s_i^b \sim_H s_i^s$, if $s_i^b \to s_j^b$ then there exist an H' and s_j^s s.t. $H \subseteq H'$ and $s_i^s \twoheadrightarrow s_j^s$ and $s_j^b \sim_{H'} s_j^s$.

The simulation relation $\sim_H$ asserts the consistency of corresponding **BIR** and SBIR states, i.e., their program counters are equal, their environments are equal through the interpretation H, and the evaluation of Π^s under H results in *true*. Then the soundness of the symbolic execution structure for multiple steps corresponds to the extension of Property 1 to a multi-step simulation theorem.

4.2 CSEC-MODEX Toolchain and IML

Aizatulin et al. [8] proposed an automated technique to verify the security of cryptographic protocols' *C* implementation. At a high level, CSEC-MODEX takes as input the *C* code of protocol participants together with a template file for the verifier (PROVERIF or CRYPTOVERIF). The toolchain extracts the IML model of the protocol, which is then converted into the verifier's input language. The template encodes assumptions about cryptographic primitives in the implementation, the environment process which spawns the participants and generates shared cryptographic material, and a query for the security property that is checked for the implementation.

The intermediate model language, IML, is a version of the applied pi calculus extended with bitstring manipulation primitives. In Fig. 4.2, $\mathtt{BS} = \{0,1\}^*$ is the set of finite bitstrings, Ops is the set of operations, including cryptographic primitives, and $\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_m)$ denotes function application. IML expressions are evaluated with respect to an environment $\eta^\iota : \mathtt{Ivar} \mapsto \mathtt{BS} \cup \{\bot\}$ which maps variables to bitstrings or $\bot$.

P and Q represent input/output processes. An executing process is the basic unit of execution in IML and has the form $(\eta^\iota, \mathtt{P})$, where P is either an input or output process.

$\mathtt{d}, \mathtt{e} \in \mathtt{Iexp}$	:=	expression
$\mathtt{b} \in \mathtt{BS},\ \mathtt{x} \in \mathtt{Ivar}$		bitstrings, variables
$\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_m)$		computation, $\mathtt{op} \in \mathtt{Ops}$
$\mathtt{P}, \mathtt{Q} \in \mathtt{IML}$	:=	process
$\mathtt{0}, \mathtt{P} \vert \mathtt{Q}$		nil, parallel composition
$!^{i \leq m}\ \mathtt{P}$		replicating P, m times
$\mathtt{new}\ \mathtt{x} : t; \mathtt{P}$		randomness
$\mathtt{in}(\mathtt{c}[\mathtt{e}_1, \ldots, \mathtt{e}_m], \mathtt{x}); \mathtt{P}$		input
$\mathtt{out}(\mathtt{c}[\mathtt{e}_1, \ldots, \mathtt{e}_m], \mathtt{e}); \mathtt{P}$		output
$\mathtt{event}(\mathtt{d}_1, \ldots, \mathtt{d}_m); \mathtt{P}$		event
$\mathtt{if\ e\ then\ P}\ [\mathtt{else\ Q}]$		conditional
$\mathtt{let\ x = e\ in\ P}$		assignment
$\mathtt{assume\ e}; \mathtt{P}$		assumption

Figure 4.2: A fragment of IML syntax

The input process 0 does nothing. In IML, inputs and outputs are performed using in and out in which c denotes the channel name and $\mathtt{e}_1, \ldots, \mathtt{e}_m$ indicate the protocol participants' identifier. The construct new x : t generates a uniform random number of type t and $\mathtt{event}(\mathtt{d}_1, \ldots, \mathtt{d}_m)$ is used to raise an event during the execution.

An IML state $(\eta^\iota, \mathtt{P}), \mathcal{Q} \in \mathtt{S}^\iota$ includes an output process P and a multiset of executing input processes $\mathcal{Q}$. The initial configuration of an input process Q is defined as $(\emptyset, \mathtt{out}(\mathtt{c}, \varepsilon); \mathtt{0}), \mathit{reduce}(\emptyset, \mathtt{Q})$ where *reduce* represents a function that executes a sequence of processes inside Q (e.g., $\mathtt{Q} = \mathtt{P}_1; \mathtt{P}_2; \ldots$) until an input process waiting for a message from channel c is reached (see the last rule in Fig. 4.3).

We have borrowed the semantics of the IML transition relations from [7, p. 23]. Fig. 4.3 presents the IML semantics. In this figure *truncate* cuts messages according to the provided length and *maxlen* is the maximum size of the channel. The construct $\mathtt{event}(\mathtt{d}_1, \ldots, \mathtt{d}_m)$ is used to raise an event $\mathtt{ev}(\mathtt{b}_1, \ldots, \mathtt{b}_m)$ during the execution where ev is an event symbol and $\mathtt{b}_1, \ldots, \mathtt{b}_m$ are bitstrings stored for expressions $\mathtt{d}_1, \ldots, \mathtt{d}_m$ in the IML environment η^ι. We extend the random number generation rule with an event fr which represents the creation of a fresh bitstring b. This simplifies stating our invariants but is operationally the same. To execute the construct let x = e in P, the bitstring b stored for the expression e in the IML environment η^ι is fetched. Subsequently, the value of the variable x is updated with the bitstring b in the environment η^ι and the construct reduces to P. The construct assume e; P reduces to P only in the case that the expression e evaluates to *true* with respect to the IML environment η^ι. The if e then P else P′ construct reduces to P if the expression e evaluates to *true* with respect to the IML environment η^ι, otherwise reduces to P′.

$$\frac{\forall j \leq m : [\![\mathtt{d_j}]\!]_{\eta'} = \mathtt{b_j} \neq \bot}{(\eta', \mathtt{event}(\mathtt{d_1}, \ldots, \mathtt{d_m}); \mathtt{P}), Q \xrightarrow{\mathtt{ev}(\mathtt{b_1},\ldots,\mathtt{b_m})} (\eta', \mathtt{P}), Q} \text{ Event}$$

$$\frac{t = \mathtt{fixed}_n \text{ for some } n \in \mathbb{N} \quad |\mathtt{b}| = n}{(\eta', \mathtt{new\ x} : t; \mathtt{P}), Q \xrightarrow{\mathtt{fr(b)}}_{\frac{1}{2^n}} (\eta'[\mathtt{x} \mapsto \mathtt{b}], \mathtt{P}), Q} \text{ New}$$

$$\frac{[\![\mathtt{e}]\!]_{\eta'} = \mathtt{b} \in \{\mathtt{BS} \cup \{\bot\}\}}{(\eta', \mathtt{let\ x = e\ in\ P}), Q \longrightarrow (\eta'[\mathtt{x} \mapsto \mathtt{b}], \mathtt{P}), Q} \text{ Let}$$

$$\frac{[\![\mathtt{e}]\!]_{\eta'} = \text{true}}{(\eta', \mathtt{assume\ e}; \mathtt{P}), Q \longrightarrow (\eta', \mathtt{P}), Q} \text{ Assume}$$

$$\frac{[\![\mathtt{e}]\!]_{\eta'} = \text{true}}{(\eta', \mathtt{if\ e\ then\ P\ else\ P'}), Q \longrightarrow (\eta', \mathtt{P}), Q} \text{ IfTrue}$$

$$\frac{[\![\mathtt{e}]\!]_{\eta'} = \text{false}}{(\eta', \mathtt{if\ e\ then\ P\ else\ P'}), Q \longrightarrow (\eta', \mathtt{P'}), Q} \text{ IfFalse}$$

$$\frac{\begin{array}{c}[\![\mathtt{e}]\!]_{\eta'} = \mathtt{b} \neq \bot \quad \mathtt{b'} = \mathit{truncate}(\mathtt{b}, \mathit{maxlen}(\mathtt{c})) \\ \forall j \leq m : [\![\mathtt{e_j}]\!]_{\eta'} = \mathtt{b_j} \neq \bot \quad Q' = \mathit{reduce}(\{(\eta', \mathtt{Q})\}) \\ \exists!(\eta', \mathtt{Q'}) \in Q : \mathtt{Q'} = \mathtt{in}(\mathtt{c}[\mathtt{e'_1}, \ldots, \mathtt{e'_m}], \mathtt{x'}); \mathtt{P'} \wedge \forall j \leq m : [\![\mathtt{e_j}']\!]_{\eta'} = \mathtt{b_j} \neq \bot\end{array}}{(\eta', \mathtt{out}(\mathtt{c}[\mathtt{e_1}, \ldots, \mathtt{e_m}], \mathtt{e}); \mathtt{Q}), Q \longrightarrow (\eta'[\mathtt{x'} \mapsto \mathtt{b'}], \mathtt{P'}), Q \uplus Q' \setminus \{(\eta', \mathtt{Q'})\}} \text{ Out}$$

Figure 4.3: The semantics of IML [7, p. 23].

We use $\xrightarrow{o'}_{p} \subseteq S' \times S'$ to denote the IML transition relation with the probability p and the event o'. The event o' may be empty, $\mathtt{fr(b)}$, or $\mathtt{ev}(\mathtt{b_1}, ..., \mathtt{b_m})$. Moreover, an IML trace is defined as $R' = s'_1 \xrightarrow{o'_1}_{p_1} \cdots \xrightarrow{o'_{n-1}}_{p_{n-1}} s'_n \subseteq \mathcal{R}'(Q)$.

4.3 SAPIC+

SAPIC+ is a dialect of applied pi calculus that provides a language that soundly translates to TAMARIN [126], PROVERIF [45] and DEEPSEC [61]. SAPIC+ enhances SAPIC [106] by introducing destructors and let bindings with pattern matching and else branches. Fig.4.4 presents the SAPIC+'s syntax. 0 indicates the nil process, P|Q stands for the parallel execution of processes P and Q, and !P denotes the replication of P, enabling an

$\langle P, Q \rangle$ ::=

0	| !P
| P | Q	| P + Q
| new n; P	| lock st; P
| event e; P	| unlock st; P
| in(t, x); P	| delete st; P
| out(t_1, t_2); P	| insert st, t; P
| if ϕ then P else Q	| lookup st as x in P else Q
| let $t_1 = t_2$ in P else Q	

Figure 4.4: The syntax of SAPIC+ process calculus.

unbounded number of sessions in protocol runs. SAPIC+ contains the non-deterministic choice operator, denoted as + and introduced in [16]. A process P + Q can either move as if it were P, or as if it were Q. The new construct creates fresh values, and in and out receives and sends messages over the channel. The event construct raises events that security properties can refer to, but otherwise does not change the execution. They will be used to capture event functions. Conditionals are described by first-order formulae ϕ over equalities on terms, possibly containing variable quantifiers, as in [106].

Moreover, SAPIC+ syntax includes *stateful* processes that manipulate globally shared states, i.e., some database, register or memory. These shared states can be read using lookup, deleted by delete, and added to with insert from different parallel threads. Access from other threads can be restricted by lock, or the restriction can be removed by unlock.

The transition relation of SAPIC+ is defined by the rules described in Fig. 4.5. The SAPIC+ semantics is characterized by a labeled transition relation among process configurations, denoted as $(\mathcal{N}, \mathcal{S}, \mathscr{P}, \sigma, \mathscr{L})$. Here, $\mathcal{N}$ represents the set of fresh names produced by the processes, $\mathcal{S}$ is a partial function depicting the functional store, $\mathscr{P}$ is a multiset of ground processes executing concurrently, σ is a ground substitution representing messages sent to the environment, and $\mathscr{L}$ is the set of locks currently held. For a given process P, $\mathfrak{T}^{sp}(P)$ denotes the set of all possible traces generated by process P. We define a trace $t^{sp} \in \mathfrak{T}^{sp}$ as a sequence of SAPIC+ events such that $t^{sp} = o_1^{sp} \dots o_m^{sp}$.

SAPIC+ assumes that protocols only transmit valid messages, as is captured by the Msg predicate in Fig. 4.5. Additionally, postfix notation is used for the substitution σ, which means that $t\sigma$ is the term where each variable x in the domain of σ and in t is replaced by $\sigma(x)$.

SAPIC+ facilitates the analysis of equivalence properties through its backends, specifically DEEPSEC, a specialized tool designed for this purpose. DEEPSEC focuses on indistinguishability properties, particularly trace equivalence. It employs a language similar

Standard operations:

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{0\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}|\mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}, \mathsf{Q}\}, \sigma, \mathcal{L})$$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{!\mathsf{P}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{!\mathsf{P}, \mathsf{P}\}, \sigma, \mathcal{L})$$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{new}\ n; \mathsf{P}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N} \cup \{n'\}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\{n \mapsto n'\}\}, \sigma, \mathcal{L})$$
if $n' \in \mathcal{N}_{\mathsf{priv}}$ is fresh

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{if}\ \phi\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma, \mathcal{L})$$ if ϕ holds

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{if}\ \phi\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{Q}\}, \sigma, \mathcal{L})$$ if ϕ does not hold

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{let}\ t_1 = t_2\ \mathsf{in}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\tau\}, \sigma, \mathcal{L})$$
if $t_1\tau =_E t_2$ and τ is grounding for t_1

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{let}\ t_1 = t_2\ \mathsf{in}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{Q}\}, \sigma, \mathcal{L})$$ if for all τ, $t_1\tau \neq_E t_2$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{event}\ e; \mathsf{P}\}, \sigma, \mathcal{L}) \xrightarrow{e} (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma, \mathcal{L})$$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \xrightarrow{\mathsf{K}(t)} (\mathcal{N}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$$
if $t =_E R\sigma$ for some $R \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{\mathsf{pub}}, \mathcal{V})$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{out}(t_1, t_2); \mathsf{P}, \mathsf{in}(t, x); \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup \{\mathsf{P}, \mathsf{Q}\{x \mapsto t_2\}\}, \sigma, \mathcal{L})$$
if $t_1 =_E t$ and $\mathsf{Msg}(t_2)$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{out}(t_1, t_2); \mathsf{P}\}, \sigma, \mathcal{L}) \xrightarrow{\mathsf{Out}(R), \mathsf{K}(t_1)} (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma \cup \{\mathsf{att}_n \mapsto t_2\}, \mathcal{L})$$
if $t_1 =_E R\sigma$ for some $R \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{\mathsf{pub}}, \mathcal{V})$
$\mathsf{Msg}(t_2)$ and $n = |\sigma| + 1$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{in}(t, x); \mathsf{P}\}, \sigma, \mathcal{L}) \xrightarrow{\mathsf{In}(R, R'), \mathsf{K}(\langle t, R'\sigma \rangle)} (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\{x \mapsto R'\sigma\}\}, \sigma, \mathcal{L})$$
if $t =_E R\sigma$ and $\mathsf{Msg}(R'\sigma)$
for some $R, R' \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{\mathsf{pub}}, \mathcal{V})$

Operations on global state:

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{insert}\ st, t;\ \mathsf{P}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}[st \mapsto t], \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma, \mathcal{L})$$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{delete}\ st;\ \mathsf{P}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}[st \mapsto \bot], \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma, \mathcal{L})$$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ st\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\{u/x\}\}, \sigma, \mathcal{L})$$
if $\mathcal{S}(t) =_E u$ is defined and $st =_E t$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ st\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{Q}\}, \sigma, \mathcal{L})$$
if $\mathcal{S}(t)$ is undefined for all $t =_E st$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{lock}\ st;\ \mathsf{P}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma, \mathcal{L} \cup \{st\})$$ if $st \notin_E \mathcal{L}$

$$(\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{unlock}\ st;\ \mathsf{P}\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{N}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{P}\}, \sigma, \mathcal{L} \setminus \{t \mid t =_E st\})$$

Figure 4.5: The operational semantics of SAPIC+ [60, p. 18].

to PROVERIF but without the "!" operator for unbounded replication, as it supports only bounded verification. Unlike PROVERIF, DEEPSEC provides a decision procedure that guarantees termination, given sufficient resources. As a result, DEEPSEC can effectively check trace equivalence in cases where PROVERIF fails to terminate, though it requires bounding the number of replications.

4.4 Side Channels and Observational Models

Resource sharing is inevitable in computing due to limitations in available resources. However, if not done carefully, it can introduce unintended information flow channels, also known as side channels. These channels can potentially be exploited by a malicious process to exfiltrate secret information from trusted ones.

Attacks that exploit the data and instruction caches are among the most commonly used side-channel attacks [168, 4, 137, 167]. One widely used technique for extracting information via caches is known as Prime+Probe [146]. In an instruction-cache attack using this technique, first, the attacker **primes** the cache by filling it with their own instructions. Then, while the victim executes, some of the attacker's cached instructions may be evicted. Finally, the attacker **probes** the cache by *measuring access times* to their instructions to detect evictions that reveal the victim's execution behavior.

The number of attack techniques exploiting microarchitectural features, like caches, to leak secret data continues unabated. Consequently, the study of information flow analysis techniques to ensure the absence of information leakages due to side channels is a topic of increasing relevance. A formal model of side channels is essential for such an analysis. However, explicitly modeling all the intricate features of modern processors—like cache hierarchies, replacement policies, and memory interactions—is almost infeasible due to their complexity and because many of these microarchitectural details are not publicly available. To address this challenge, *abstract observational models* [136, 54] (a.k.a., *leakage contracts* [89]) provide an alternative by overapproximating an attacker's capabilities.

An observation model $\mathcal{M}$ extends the abstract representations of a processor's operational semantics by a set of system states S, a set of possible attacker observations Obs and a labeled transition relation $\rightarrow_m \subseteq S \times Obs \times S$ indexed with the execution mode $m \in \{r, t\}$. When $m = r$, we mean that the processor executes at the software-visible ISA level using a sequential transition system, while with $m = t$, we denote the transition relation of some target microarchitecture where the information flow may be affected by optimizations such as out-of-order or speculative execution. Essentially, observations define which parts of the processor state influence the side channel at each transition. This enables information flow analysis without requiring us to know the exact microarchitectural behavior.

The primary property to formalize the absence of microarchitectural leakages due to side channels is *conditional non-interference* [88]. Let $s \in S$ be a system state, including microarchitectural components like caches, $traces : T \mapsto 2^{Obs}$ be a function to extract the sequence of observations from a given execution trace $\tau \in T$, and $s \sim_{\mathcal{M}} s'$ is the state's indistinguishability relation w.r.t the model $\mathcal{M}$. Then we say:

Definition 4.1 (Conditional Non-interference)**.** A system is *conditionally non-interferent*

Pesudo-(ARM)Assembly	Observations
`ldr x2, [x0]`	load from x0 ($\mathcal{M}_{ct}$ observation)
`if x0 < x1`	branch on x0<x1 ($\mathcal{M}_{ct}$ observation)
`  ldr x3, [x2]`	load from x2 ($\mathcal{M}_{ct}$ observation)
`else`	
`  x2* = x2`	none
`  x3* = x3`	none
`  ldr x3*, [x2*]`	load from x2* ($\mathcal{M}_{spec}$ observation)

Figure 4.6: The Spectre V1 example instrumented via $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$. We marked shadow observations with ⋆.

if for all indistinguishable initial states s and s' (i.e., $s \sim_{\mathcal{M}} s'$), if for every execution sequence $\tau_1^r = s \xrightarrow{o_1}_r s_1 \ldots \xrightarrow{o_n}_r s_n$ there exists a corresponding sequence $\tau_2^r = s' \xrightarrow{o'_1}_r s'_1 \ldots \xrightarrow{o'_n}_r s'_n$ such that $traces(\tau_1^r) = traces(\tau_2^r)$, then for every execution $\tau_1^t = s \xrightarrow{o_1}_t s_1 \ldots \xrightarrow{o_n}_t s_n$, there must also exist a corresponding $\tau_2^t = s' \xrightarrow{o'_1}_t s'_1 \ldots \xrightarrow{o'_n}_t s'_n$ such that $traces(\tau_1^t) = traces(\tau_2^t)$.

A common strategy to prevent cache timing side channels in literature is the *constant time* (CT) policy [26], which requires that memory accesses and control flow decisions should depend only on public (non-secret) information. In this thesis, the observational model $\mathcal{M}_{ct}$ formalizes this policy and it makes the program counter of each instruction and the accessed memory addresses observable.

Alas, speculative execution introduces new attack vectors that break the assumptions of CT execution. *Spectre* attacks [102] exploit speculation to leak data through side channels like caches. These attacks are characterized by a speculation primitive that allows leaking secret during speculative execution. We use the observational model $\mathcal{M}_{spec}$ proposed by Buiras et al. [54] to observe transient sensitive operations like memory accesses. They models attacker observations during speculation using *refined* (a.k.a., *shadow*) observations, which enables the attacker to observe sensitive operations that may happen in mispeculation.

4.4.1 Observation Refinement

Technically, an observation model $\mathcal{M}$ groups states into equivalence classes where states appear indistinguishable. Observation refinement improves this partitioning by introducing a refined model $\mathcal{M}'$ that further partitions these classes. Essentially, $\mathcal{M}'$ captures additional behavioral variations, particularly those linked to side-channel effects, that $\mathcal{M}$ may overlook. For instance, Fig. 4.6 depicts the Spectre V1 primitive annotated with attacker's observation from the $\mathcal{M}_{ct}$ model and shadow observation from the $\mathcal{M}_{spec}$ that enable the attacker to observe operation that may execute during the speculation.

HolBA supports side channel analysis using Scam-V [136], which is embedded in

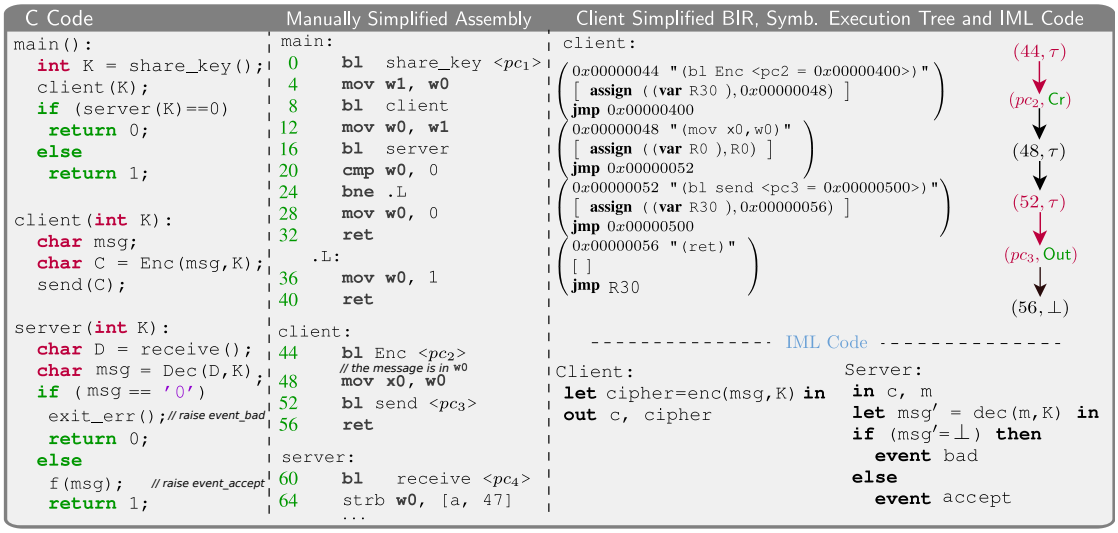


Figure 4.7: Running example 1.

HolBA, to annotate the side channel's observational power in the resulting BIR program. Scam-V extends the transpilation process of HolBA to inline observation statements into BIR and annotates the resulting BIR with observations according to the observational model being analyzed. In our work, we adopt established observational models from the Scam-V platform (see Sec. 5.1.5).

4.5 Running Examples

To support clarity throughout the thesis, we introduce a set of running examples that are used consistently across different stages of the methodology. These examples serve to concretize abstract concepts, highlight practical challenges, and illustrate how our proposed techniques apply in realistic settings.

Running Example 1. Our first running example, Fig. 4.7, consists of a client and a server that use a symmetric-key encryption scheme to communicate securely. This example shows a weak form of authentication, called aliveness [122]: the server will accept the connection to the (single) client only if it can successfully decrypt the received message using the pre-shared key. First, the client encrypts a message using the shared key, and sends it to the server. Second, the server receives the encrypted message at the other end and decrypts it using the same key. Depending on whether the decryption succeeds or fails, either `event_accept`, to show acceptance of the connection with the client, or `event_bad` will be released. Note that function addresses are chosen randomly and line numbers and pc_i are addresses of instructions and functions in the memory.

Running Example 2. We will use the Basic Access Control (BAC) protocol to

C Code

```
int passport(char k_e, char k_m) {
 received = get_challenge();
 if (compare(received,"CHALLENGE") == 0){
  gen_n_t = get_nonce();
  send_nonce(gen_n_t);
  cipher = get_msg();
  mac_input = get_msg();
  calc_mac = mac(cipher, k_m);
  if (compare(calc_mac, mac_input) == 0){
   if (sdec(cipher,k_e,xn_r,xn_t,xk_r)){
    if (compare(xn_t, gen_n_t) == 0) {
     k_t = get_nonce();
     response=senc(gen_n_t,xn_r,k_t,k_e);
     mac_resp = mac(response, k_m);
     send(response, mac_resp);
    } else return 0;
   } else return 0;
 } else return 0;}}
```

Simplified Assembly

```
.
.
bl  0x180 <get_nonce>
strb w0, [sp, #16]
add x0, sp, 0x128
bl  0x20 <send_nonce>
bl  0x52 <get_msg>
str x1, x0
bl  0x52 <get_msg>
bl  0x160 <mac>
bl  0x120 <compare>
cmp x0, 0x0
bne  0x200 <exit>
ldr x2, [sp]
.
.
```

Figure 4.8: Running example 2 (C and assembly).

better explain our side-channel analysis approach. Fig. 4.8 illustrates the C implementation of a function designed to establish secure communication with the e-passport chip, which we have developed due to the lack of source code. The assembly snippet in Fig. 4.8 corresponds to the highlighted C code. This function adheres to the Basic Access Control (BAC) protocol and complies with the International Civil Aviation Organization standards [82]. The corresponding snippet of simplified assembly code that we use throughout the thesis to explain our methodology is also presented in Fig. 4.8. BAC employs a three-pass challenge-response protocol that facilitates mutual authentication between e-passports and readers. The e-passport continuously awaits a challenge from the reader. Upon receiving this challenge, it generates a fresh nonce and sends it back to the reader. Once the reader receives the nonce from the e-passport, it creates its own random nonce and encrypts its nonce along with the e-passport nonce using a pre-shared key k_e. The reader also creates a Message Authentication Code (MAC) for the resulting cipher using the k_m key and transmits both the cipher and its associated MAC back to the e-passport. Upon receipt of the cipher and the MAC, the e-passport calculates the MAC of the received cipher and compares it with the transmitted MAC. If the MAC check is successful, the e-passport attempts to decrypt the cipher. If decryption is successful, it verifies whether the generated nonce is present within the encrypted message. After completing all verification checks, the e-passport authenticates the reader and subsequently validates itself to the reader in a similar manner.

Fig. 4.9 is a portion of **BIR** blocks of the BAC protocol illustrated in Fig. 4.8, along with the corresponding **BIR** observations, symbolic execution events and extracted Sapic$^+$ model. $0x200$ is the address of the last **BIR** statement, i.e., **halt** statement, which translates to 0. The Sapic$^+$ process will then translate into the DeepSec process, which closely mirrors what is illustrated here. Jumps (at lines 0, 2, 3, and 4) are the translation of *branch and link* instruction used for function calls in ARM, which requires updating

	BIR Block	BIR Observation	SBIR Observation	SAPIC+ Process (and similarly, DEEPSEC Process)
0	[R30=1;](0x90,jmp(0x52)) // get-msg	Pc(0x90)	In($R0_1$)	out(0x90); in($R0_1$); //mac of cipher
1	(0x94,assign(R1, R0))	Pc(0x94)	Asn($R1, R0_1$)	out(0x94); let $R1 = R0_1$ in
2	[R30=3;](0x98,jmp(0x52)) // get-msg	Pc(0x98)	In($R0_2$)	out(0x98); in($R0_2$); //cipher
3	[R30=4;](0x9c,jmp(0x160))// mac	Pc(0x9c)	FCall(mac, $R0_2, R0_3$)	out(0x9c); let $R0_3$ = mac($R0_2$) in
4	[R30=5;](0xa0,jmp(0x120))// compare	Pc(0xa0)	FCall(compare, $R0_3, R1, R0_4$)	out(0xa0); let $R0_4$ = compare($R0_3, R1$) in
5	(0xa4,assign(Z, (R0 Equal 0x0)))	Pc(0xa4)	Asn(Z, ($R0_4$ Equal 0x0))	out(0xa4); let Z = equal($R0_4$, 0x0) in
6	(0xa8,cjmp(Z, 0xac, 0x200))	Pc(0xa8), Cnd(Z, 0xac, 0x200)	Cnd(Z, 0xac, 0x200)	out(0xa8); out(Z); if Z
7	(0xac,assign(R2,load(mem,SP,64)))	Pc(0xac), Ld(SP)	Asn($R2$,load(mem, SP, 64))	then out(0xac);out(SP);let $R2$ = load(mem, SP) in
	...	...	...	... else out(0x200); 0.

Figure 4.9: Running example 2 (BIR till SAPIC+).

the *link register* $R30$. We present this register update in [..] to mean that it is not relevant to what we intend to present in this example.

Chapter 5

Methodologies

The previous chapter established the preliminaries of binary-level protocol analysis within the HolBA framework and introduced the necessary background on side channels, observational models, and the Csec-modex toolchain. However, the standard BIR language lacks the constructs needed to faithfully capture cryptographic protocol behaviors, such as secure randomness generation, structured network communication, and cryptographic operations. In this chapter, we extend BIR with such protocol-specific semantics, enabling the symbolic execution engine to reason about both functional correctness and security guarantees. Next, we explain how we have *significantly* extended HolBA's vanilla symbolic execution (Sec. 4.1.1) to handle network communication, calls to cryptographic primitives and event functions, and random number generation, which are essential to reason about protocols' security. Finally, having extended BIR with protocol-aware constructs and implemented a crypto-aware symbolic execution engine, we will turn to the task of producing high-level models suitable for automated protocol verification.

5.1 BIR with Cryptography

BIR, as described in [116], does not support ingredients required to reason about the security of cryptographic protocols. To resolve these issues, we model random number generation and abstract network communications and formulate assumptions on state transformation in certain function calls on top of the existing BIR semantics. Such an extension preserves the verified properties of BIR and, thus, the soundness of binary transpilation.

Using the information in the (unstripped) binaries' header and preprocessing of lifted programs, we split the address space of BIR into five label sets: $\mathbf{L} = \mathbf{L}_{\mathcal{N}} \uplus \mathbf{L}_{\mathbf{Op}} \uplus \mathbf{L}_{\mathcal{A}} \uplus \mathbf{L}_{\mathcal{R}} \uplus \mathbf{L}_{\mathcal{E}}$. The sets $\mathbf{L}_{\mathbf{Op}} = \bigcup_{\mathbf{op} \in \mathbf{Op}}$, $\mathbf{L}_{\mathcal{A}}$, $\mathbf{L}_{\mathcal{R}}$, $\mathbf{L}_{\mathcal{E}}$ correspond to the *cryptographic libraries*, *attacker* calls, *random number generation*, and *event* functions. Ad-

dresses outside these label sets are classified as normal execution points in $\mathbf{L}_{\mathcal{N}}$. Moreover, a specific label set $\mathbf{L}_{\mathcal{L}} \subset \mathbf{L}_{\mathcal{N}}$ defines loop entry points. For each label set, we axiomatize the expected behavior of the BIR program by defining a number of assumptions. We also define a specific entry point for each function—denoted by ξ_ℓ for $\ell \in \{\mathbf{L}_{\mathcal{N}}, \mathbf{L}_{\mathrm{Op}}, \mathbf{L}_{\mathcal{A}}, \mathbf{L}_{\mathcal{R}}, \mathbf{L}_{\mathcal{E}}\}$—and ensure that function calls are done only through the specified entry points.

In the crypto-aware symbolic execution (cf. Sec. 5.2), these function calls will be treated as atomic operations. We thus introduce some notation to indicate with an event whenever a *sequence* of steps passes via these special functions.

We extend the BIR transition relation with events, $\xrightarrow{\mathbf{a}} \subseteq \mathbf{S}^b \times \mathcal{O}^b \times \mathbf{S}^b$, where $\mathcal{O}^b$ is the set of observable events plus the silent transition τ. Then, a multi-step BIR transition $s_0^b \xrightarrow{(\mathbf{o}_1^b, \ldots, \mathbf{o}_m^b)}{}^{+} s_n^b$ exists if $s_0^b \xrightarrow{\mathbf{o}_1^b}{}^{*} s_1^b \ldots s_{n-1}^b \xrightarrow{\mathbf{o}_m^b}{}^{*} s_n^b$ where $s_i^b \xrightarrow{\mathbf{o}^b}{}^{*} s_j^b$ if $s_i^b \xrightarrow{\tau}{}^{*} \xrightarrow{\mathbf{o}^b} \xrightarrow{\tau}{}^{*} s_j^b$ is reminiscent to the big-step semantics. Also, we use $\mathbf{R}^b(\mathbf{P}, s_0^b)$ to denote the set of execution traces of P starting from the initial state s_0^b. Moreover, a sequence of transitions produces a BIR observation trace $\mathbf{t}^b = \mathbf{o}_1^b, \ldots, \mathbf{o}_m^b$.

We define $ret : \mathbf{S}^b \to \mathbf{L}$ to obtain the next execution point immediately reachable after returning from a call. $mstore : (\mathbf{Bvar} \mapsto \mathbf{Bval}) \times \mathbf{Bvar} \times 2^{\mathbf{Bvar}} \times \mathtt{BS} \times \mathbb{N} \to (\mathbf{Bvar} \mapsto \mathbf{Bval}) \times \mathbf{Bval}$ stores bitstrings into the memory in the BIR environment. Given $heap \in Mem$ and $l \in \{1, 8, 16, 32, 64, 128\}$ and $|\mathtt{b}|$ a multiple of l that can be encoded in 128 bits, $mstore(\eta^b, heap, Mem, \mathtt{b}, l)$ stores $\mathtt{b}$ in l-bit chunks, preceded by l (encoded as a word) in the memory $Mem \subseteq \mathbf{Bvar}$, starting from the pointer stored in $heap$. $mstore$ returns a new environment and the address of the data within it, as indicated by $heap$ in the *previous* environment η^b. We also introduce notation for reading this bitstring. Let $\|$ be the byte concatenation operator. Then, $mload(\eta^b, \mathbf{a}) \stackrel{\text{def}}{=} \Big\|_{i=1\ldots\eta^b(\mathbf{a})} \eta^b(\mathbf{a} + i)$ for the given η^b and the address '$\mathbf{a}$'.

CryptoBap supports *call-by-value*, *call-by-reference*, and data passing via global variables (which our case study TinySSH uses, see Sec. 7.2) call conventions. For brevity, we focus on the call-by-value convention; the others follow a similar pattern.

5.1.1 Random Number Generation

BIR is a deterministic language; as a result, we are unable to draw cryptographic keys without an external source of randomness that the attacker cannot predict. Thus, we allocate a memory region RM in the initial state for storing $k \in \mathbb{N}$ random values of size $l \in \mathbb{N}$, for the security parameter $n = l * w$ that is a multiple of some supported word length w. We assume RM is an ordered list of $k * l$ consecutive addresses. To track the number of words read from RM, we define a counter $\mathbf{r_k}$ and store it in the environment. Given an initial state s_0^b and a random tape $\mathtt{rm_k} \in \{0,1\}^{k*l*w}$ the state $s_0^b.\eta^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0]$ is an instance of this initial state. To extract a random

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_s} \quad \mathbf{pc}' = ret(\eta^b, \mathbf{pc}) \quad \mathbf{a} = \eta^b[r_0] \quad \mathsf{e} = mload(\eta^b, \mathbf{a})}{\mathbf{P} \vdash (\eta^b, \mathbf{pc}), \mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m] \xrightarrow{\mathbf{Out}(\mathsf{e}, (\mathbf{e}_1, \ldots, \mathbf{e}_m))} (\eta^b, \mathbf{pc}'), \mathsf{e} :: \mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m]} \; \mathcal{A}_s$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_r} \quad \mathbf{pc}' = ret(\eta^b, \mathbf{pc}) \quad (\eta^{b\prime}, \mathbf{a}) = mstore(\eta^b, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, \mathsf{e}', 128)}{\mathbf{P} \vdash (\eta^b, \mathbf{pc}), \mathsf{e}' :: \mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m] \xrightarrow{\mathbf{In}(\mathsf{e}', (\mathbf{e}_1, \ldots, \mathbf{e}_m))} (\eta^{b\prime}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m]} \; \mathcal{A}_r$$

Figure 5.1: The semantics of BIR network communication.

number of size l from RM, we define $\mathfrak{R} : (\mathbf{Bvar} \to \mathbf{Bval}) \times \mathbb{N} \to \mathtt{BS}$ which returns a value from RM yet unread: $\mathfrak{R}(\eta^b, n) \stackrel{\text{def}}{=}$ let $\mathbf{x} = \eta^b[\mathbf{r_k}]$ in $\|_{i=0\ldots l-1} \eta^b[\mathbf{RM} + \mathbf{x} + i]$. This construction is reminiscent of probabilistic Turing machines, only that the random number generator is finite due to the finite-memory restriction of BIR's memory.

We call a function from $\mathbf{L}_{\mathcal{R}}$ with $\mathbf{l}_{\mathcal{R}} \in \mathbf{L}_{\mathcal{R}}$ one of its entry points an RNG function, if for any entering state $(\eta_0^b, \mathbf{l}_{\mathcal{R}})$ for which $\mathfrak{R}(\eta_0^b, n)$ is defined, and execution point $(\eta^{b\prime\prime}, \mathbf{l})$ after returning from RNG, i.e., $\mathbf{l} = ret(\eta_0^b, \mathbf{l}_{\mathcal{R}})$, the output register holds the address of a copy of the random value and $\mathbf{r_k}$ is updated, i.e., $\eta^{b\prime\prime} = \eta^{b\prime}[r_0 \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta_0^b[\mathbf{r_k}] + l]$ with $(\eta^{b\prime}, \mathbf{a}) = mstore(\eta_0^b, heap, Mem, \mathsf{x_i}, 128)$ s.t. $\mathsf{x_i} = \mathfrak{R}(\eta_0^b, n)$. We denote RNG steps as $(\eta_0^b, \mathbf{l}_{\mathcal{R}}) \xrightarrow{\mathbf{Fr}(\mathsf{x_i})}{}^*_{\mathbf{L}_{\mathcal{R}}} (\eta^{b\prime\prime}, \mathbf{l})$, with $i = \left\lfloor \frac{\eta_0^b[\mathbf{r_k}]}{l} \right\rfloor + 1$ being a counter for the number of times the RNG function was called.

5.1.2 Network Communication

Protocols relate events on different participants. Therefore, a setting where multiple parties run in parallel is essential to analyze protocols' correctness. Fig. 6.3 introduces a mixed execution, in which BIR programs run in parallel with IML processes. The latter model protocol participants for which we do not have a BIR implementation, but also the adversary.

Our BIR programs rely on an IML channel for communication that has the form $\mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m]$, where $\mathbf{e}_1, \ldots, \mathbf{e}_m$ are expressions which identify communicating parties and their channel c. To send a message e ($\xi_{\mathcal{A}_s}$ in Fig. 5.1), we fetch the value of e from the memory address $\eta^b[r_0]$, put it on the channel $\mathsf{e} :: \mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m]$, and release the $\mathbf{Out}(\mathsf{e}, (\mathbf{e}_1, \ldots, \mathbf{e}_m))$ event.

To receive a message $\mathsf{e}' :: \mathsf{c}[\mathbf{e}_1, \ldots, \mathbf{e}_m]$, represented by $\xi_{\mathcal{A}_r}$ in Fig. 5.1, we store it in a buffer that is only accessible to libraries and return (via register r_0) the address, i.e., '$\mathbf{a}$', of the memory region where the message e' is stored. Passing the address via r_0 is just one way to model the send and receive functions that also accommodates passing the buffer address by reference. We model these functions according to the

implementation.

5.1.3 Cryptographic Libraries

We establish a set of concrete assumptions on the way cryptographic libraries operate. That is, a crypto-library call, like **op**, computes the correct result, never invokes another function, and only changes its own memory, i.e., Mem_{Op}. We denote library steps with $(\eta_0^b, \mathbf{l_{op}}) \xrightarrow{\mathbf{Cr}(\mathsf{v})}{}^*_{\mathbf{L_{Op}}} (\eta^{b\prime\prime}, \mathbf{l})$, and expect *transitions using labels outside* $\mathbf{L_{Op}}$ *do not change the memory of library calls.*

We call $\mathbf{L_{op}} \subseteq \bigcup_{\mathbf{op}\in\mathbf{Op}}$ the library implementation of **op** (with arity m) and $\mathbf{l_{op}} \in \xi_{\mathbf{Op}}$ one of its entry points, if for any entering state $(\eta_0^b, \mathbf{l_{op}})$ and the return state $(\eta^{b\prime\prime}, \mathbf{l})$, the function result $\mathsf{v} = \mathbf{op}(\mathsf{b}_1, \ldots, \mathsf{b}_m)$ for $\mathsf{b}_i = mload(\eta^b, r_i)$, $i \in \{1, \ldots, m\}$, is stored in a heap and its address is put into r_0: $\eta^{b\prime\prime} = \eta^{b\prime}[r_0 \mapsto \mathbf{a}]$ where $(\eta^{b\prime}, \mathbf{a}) = mstore(\eta_0^b, heap_{\mathrm{Op}}, Mem_{\mathrm{Op}}, \mathsf{v}, 128)$.

5.1.4 Event Functions

Event functions identify specific steps in our program that we want to argue about. For example, when a protocol ends with the establishment of a key, that key is used to transmit some data. We want to show that, whenever this step is reached, it is authenticated, i.e., the purported communication partner has requested the execution of this step (e.g., `f(msg)` in Fig. 4.7). What happens in this step is not important for us, only that it is reached. We hence assume, for simplicity, that such functionality is replaced by stand-ins we call event functions. These only raise a visible event, but do not alter the memory. We denote the transition corresponding to an event function call with $(\eta_0^b, \mathbf{l}_{\mathcal{E}}) \xrightarrow{\mathbf{Ev}(\mathsf{b}_1, \ldots, \mathsf{b}_m)}{}^*_{\mathbf{L}_{\mathcal{E}}} (\eta_0^b, \mathbf{l})$ where $\mathbf{l}_{\mathcal{E}} \in \mathbf{L}_{\mathcal{E}}$ is the entry point, $\mathsf{b}_i = mload(\eta^b, r_i)$ for $i \in \{1, \ldots, m\}$ are event parameters, and $\mathbf{l} \in \mathbf{L}_{\mathcal{N}}$.

5.1.5 BIR with Observation Models

To integrate side-channel leakage into our analysis, we instrument BIR programs by annotating each BIR block with attacker observations, which are expressions that describe information that may leak through side channels. These observations can be conditional, meaning they only occur if specific conditions hold in the current state. Our *observation set* $\mathbf{Obs}^b$ includes:

$$\mathbf{Obs}^b = \tau \mid \mathbf{Ld}(\mathbf{a}) \mid \mathbf{St}(\mathbf{a}) \mid \mathbf{Cnd}(\phi, \mathbf{a}_1, \mathbf{a}_2) \mid \mathbf{Pc}(\mathbf{a})$$

Observation $\mathbf{Pc}(\mathbf{a})$ exposes the label of the BIR block, which corresponds to the program counter, $\mathbf{Cnd}(\phi, \mathbf{a}_1, \mathbf{a}_2)$ reveals the outcome of the conditional branch's condition and exposes the addresses of each branch, and $\mathbf{Ld}(\mathbf{a})$/$\mathbf{St}(\mathbf{a})$ exposes the address operand

of **load**/**store** instructions. All other instructions are considered non-leaking and emit the empty observation τ. Depending on the observation models ($\mathcal{M}_{ct}$, $\mathcal{M}_{spec}$, or any other), the observations assigned to each **BIR** block can be either normal or shadow observations (see section 4.4). For instance, take the last line in fig. 4.6, the $\mathcal{M}_{spec}$ observation annotated to the **BIR** block of the corresponding assembly code is **Ld**(x2*). By adopting established observational models from the Scam-V platform, we implement more detailed observations of **load** and **store** statements compared to other platforms. While we simplify the presentation, we preserve the same level of granularity.

Fig. 4.9 illustrates a **BIR** program snippet (first column) and its annotated observations (second column).

5.2 Crypto-aware Symbolic Execution

Symbolic execution explores all program execution paths using symbolic values— introduced at the object level—instead of concrete ones for inputs. An example is a language with a memory that maps registers to bitstrings. We defined the rules for our symbolic execution in Fig. 5.2. In this figure, $range(f)$ returns the image of f. For library calls, we define an *oracle* $\mathcal{L} : \xi_{\mathrm{Op}} \times (\mathbf{Bvar} \mapsto \mathrm{SE}) \rightarrow \mathrm{SE}$ to compute the result of the invoked function w.r.t. the current **pc** and symbolic environment. For the symbolic execution, we initialize the memory region to store random numbers **RM** with symbolic values. Thus, $\mathfrak{R}^s$ signifies the symbolic lifting of $\mathfrak{R}$, and RNG generates a fresh symbolic expression to represent the extracted value.

Similar to **BIR** transitions, we extend the symbolic transition relation of SBIR with events, i.e., $\xrightarrow{\mathrm{o}^s}\!\!\!\!\!\rightarrow \subseteq \mathrm{S}^s \times \mathcal{O}^s \times \mathrm{S}^s$, and use $\mathrm{R}^s(\mathbf{P}, s_0^s)$ to denote the set of symbolic execution traces of **P** starting at s_0^s. Let $\mathrm{t}^s \in \mathfrak{T}^s$ denote a SBIR observation trace as a sequence of observations such that $\mathrm{t}^s = \mathrm{o}_1^s, \ldots, \mathrm{o}_m^s$. SBIR observations $\mathrm{o}^s \in \mathcal{O}^s$ include **BIR** observations on symbolic values or expressions, as well as observations related to network communication and calls to cryptographic primitives, event functions and random number generation.

5.2.1 Loops

Loops can naïvely be handled by *unrolling*. This, however, is inefficient in most cases and can quickly result in a path explosion. To avoid this, we summarize loops following Strejček [163]. The algorithm summarizes the loops' effect on program variables and path conditions to compute a necessary condition on the loop's inputs to reach a specific execution point in the program. The summary is computed in terms of a tuple of *iterated symbolic state* and *looping condition*. The iterated symbolic state computes for each variable modified within the loop its symbolic value based on the initial value

$$\frac{\mathbf{pc} \in \xi_{\mathcal{E}} \qquad \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) \qquad (\mathrm{d}_1^s, \ldots, \mathrm{d}_m^s) \notin range(\eta^s)}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}) \xrightarrow{\mathrm{Ev}(\mathrm{d}_1^s, \ldots, \mathrm{d}_m^s)} (\Pi^s, \eta^s, \mathbf{pc}')} \text{ event}$$

$$\frac{\begin{array}{cc} \mathbf{pc} \in \xi_{\mathcal{R}} & \mathrm{x}_i^s \notin range(\eta^s) \\ \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) & \mathrm{x}_i^s = \mathfrak{R}^s(\eta^s, n) \\ i = \left\lfloor \frac{\eta^s[\mathrm{r}_k^s]}{l} \right\rfloor + 1 & (\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap, Mem, \mathrm{x}_i^s, 128) \\ & \eta^{s\prime\prime} = \eta^{s\prime}[r_0 \mapsto \mathbf{a}; \mathrm{r}_k^s \mapsto \eta^s[\mathrm{r}_k^s] + l] \end{array}}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}) \xrightarrow{\mathrm{Fr}(\mathrm{x}_i^s)} (\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}')} \text{ RNG}(n)$$

$$\frac{\begin{array}{cc} \mathbf{pc} \in \xi_{\mathrm{Op}} & \mathrm{v}^s \notin range(\eta^s) \\ \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) & \mathrm{v}^s = \mathcal{L}(\mathbf{pc}, \eta^s) \\ & (\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap_{\mathrm{Op}}, Mem_{\mathrm{Op}}, \mathrm{v}^s, 128) \end{array}}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}) \xrightarrow{\mathrm{Cr}(\mathrm{v}^s)} (\Pi^s, \eta^{s\prime}[r_0 \mapsto \mathbf{a}], \mathbf{pc}')} \text{ library}$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_s} \qquad \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) \qquad \mathbf{a} = \eta^s[r_0] \qquad \mathrm{e}^s = mload(\eta^s, \mathbf{a})}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}), \mathrm{c}[\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s] \xrightarrow{\mathrm{Out}(\mathrm{e}^s, (\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s))} (\Pi^s, \eta^s, \mathbf{pc}'), \mathrm{e}^s :: \mathrm{c}[\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s]} \mathcal{A}_s$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{L}} \qquad \mathbf{pc}' = exit(\mathbf{pc}) \qquad \mathrm{t}^s \notin range(\eta^s) \qquad (\Pi^{s\prime}, \eta^{s\prime}) = processLoop(\Pi^s, \eta^s, \mathbf{pc})}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}) \xrightarrow{\mathrm{loop}(\mathrm{t}^s)}{}^{+} (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}')} \text{ loop}$$

$$\frac{\begin{array}{cc} \mathbf{pc} \in \xi_{\mathcal{A}_r} & \mathrm{e}^s \notin range(\eta^s) \\ \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) & (\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, \mathrm{e}^s, 128) \end{array}}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}), \mathrm{e}^s :: \mathrm{c}[\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s] \xrightarrow{\mathrm{In}(\mathrm{e}^s, (\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s))} (\Pi^s, \eta^{s\prime}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathrm{c}[\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s]} \mathcal{A}_r$$

Figure 5.2: Crypto-aware symbolic execution semantics.

of the program's variables and *path counters*. Each path counter indicates the number of iterations of a specific path within the loop leading from the loop entry point to itself. For each path in the loop, a path condition is computed, and the conjunction of all such conditions is the looping condition.

We have *automated* the loop summarization process in our symbolic execution. In Fig. 5.2, the function $processLoop : \mathrm{S}^s \rightarrow \mathrm{SE} \times (\mathbf{Bvar} \mapsto \mathrm{SE})$ represents our implementation to summarize loops' effect. It takes as input the symbolic state of the loop entry point and reflects the effect of the loop body in its exit state (computed by

Simplified C Code

```
do {
 k = receive();
 if (!k) assert(false);
} while (!newkey(k));
```

Figure 5.3: A simplified loop from the TinySSH binary.

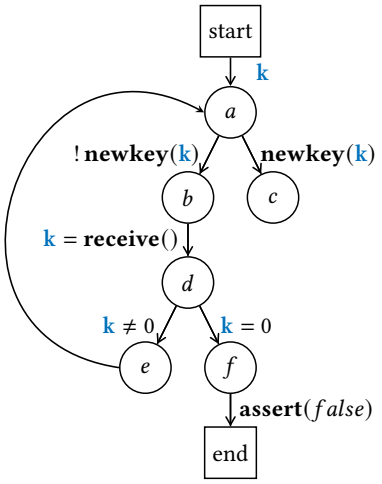


Figure 5.4: Flowgraph of the example loop.

exit : $\xi_{\mathcal{L}} \longrightarrow \mathbf{L}$). The rule also raises the event loop(t^s) with t^s being the number of loop iterations. Note that the vanilla symbolic execution (see Sec. 4.1.1) uses unrolling for loops, while our crypto-aware symbolic execution uses the loop summary to handle bounded loops.

Loops in protocol implementations are often not bounded; typically, each session runs in a $while(true)\{..\}$ loop until the server is externally terminated. However, the semantics of IML, like most cryptographic standard models, assumes a bound on the protocol. Thus, we need to assume that such loops are externally terminated after some polynomial time in the security parameter. This is captured by our automated loop summarization and by translation to the replication operator.

5.2.1.1 Loop Summarization Example

Our symbolic execution engine handles *bounded loops* using the summarization technique of Strejček [163]. We show by an example how this method is used in our symbolic execution to summarize loops. Fig. 5.3 is (the simplified version of) a loop from the TinySSH's implementation, and Fig. 5.4 indicates its flowgraph. Our example loop consists of nodes $\{a, b, d, e\}$, with a being the entry node of the loop, and a single path $\pi_1 = abdea$. The program continues to execute from node c if a new key is received.

The loop's effect can be described by an iterated symbolic environment $\eta^s_{\vec{t}}$ for $\vec{t}$ iterations of the loop. The $\vec{t}$ is a vector of path counters, which in our example, consists of a single counter t_1 that is assigned to π_1. The only variable which is modified within the loop is $\mathbf{k}$. In our example, the value of $\mathbf{k}$ is independent of the number of loop iterations and the initial values of program variables, and it can be described by a symbolic value $\eta^s_{\vec{t}}[\mathbf{k}] = \mathrm{k}^s_{\vec{t}}$.

The looping condition describes the necessary condition to keep looping along the

acyclic paths inside a given loop. For our example, the looping condition, $\Pi^s_{\vec{t}} = \Pi^s_1$, consists of a single formula Π^s_1 which describes the required condition to loop along the acyclic path π_1. The formula Π^s_1 is based on two assertions, namely, $!\,\mathbf{newkey}(\mathbf{k})$ and $\mathbf{k} \neq 0$. Let's say that in the symbolic environment, the stored values of $\mathbf{k}$ in $(t'_1 - 1)$-st and t'_1-st iteration of the path π_1 are $k^s_{t'_1-1}$ and $k^s_{t'_1}$, respectively. Then, to get t_1 iterations of π_1, the formula Π^s_1 says that for each t'_1 satisfying $0 < t'_1 \leq t_1$, $k^s_{t'_1-1}$ is not a new key, $k^s_{t'_1}$ is received from the channel, and it is not equal to zero:

$$\Pi^s_{\vec{t}} \equiv \Pi^s_1$$

$$\Pi^s_1 \equiv \forall t'_1.\ (0 < t'_1 \leq t_1 \rightarrow\ (\ !\,\mathbf{newkey}(k^s_{t'_1-1}) \wedge k^s_{t'_1} = \mathbf{receive}()\ \wedge\ k^s_{t'_1} \neq 0))$$

The loop summary is characterized by $(\eta^s_{\vec{t}}, \Pi^s_{\vec{t}})$ which we attach it to the symbolic state at the entry node a. Afterward, we proceed to execute the program from the exit point c.

5.2.2 Indirect Jumps

If during symbolic execution of the code, we encounter an indirect jump, e.g., **jmp** e, we evaluate e w.r.t. the current state to get an expression e'; we then query the SMT solver for a satisfiable assignment to $tgt = e' \wedge \Pi^s$, assuming that tgt does not occur in e' and Π^s. The solver returns one possible target, say t. We repeat this procedure, each time asking the solver to exclude found targets, until the query becomes unsatisfiable. This technique was sufficient for our experiments; however, for more complex cases, some optimizations would be required, e.g., considering only a subset of possible targets instead of enumerating all.

5.2.3 SBIR with Observation Models

To analyze side-channel leakages across all feasible paths, we extend HolBA's vanilla symbolic execution to track attacker observations alongside path constraints. Each symbolic state $s^s \in S^s$ now encapsulates a path condition—a logical constraint indicating a condition under which a path is taken—and a list of symbolic observations which represent leaked information (e.g., memory addresses and branch targets) that an attacker could infer.

The SBIR observations of our running example are depicted in Fig. 4.9. Note that SBIR observations include **BIR** observations, but with symbolic values, which we have not presented.

5.3 Model Extraction

Model extraction is the bridge between low-level binary execution traces and the abstract formal models consumed by tools such as ProVerif, Tamarin, CryptoVerif and DeepSec. This chapter describes the translation process in detail, comparing our generated models with those from existing toolchains, and explaining how observational information is preserved to support both functional and side-channel analyses. These extracted models serve as the inputs for the soundness proofs in Chapter 6.

The SBIR tree is constructed from a **BIR** program and an initial symbolic state as follows: the root is the initial state. For any node, including the root, the crypto-aware symbolic execution gives us up to two successors states. If the node represents a branching statement, we obtain two successor states. We store the statement's condition in a branching node and proceed to translate the two successor states into subtrees. If the node represented any other statement, there can only be one or no successor state, and we store an event node with, or respectively without, a successor tree. The exceptions are the nodes containing *indirect jumps* which may have multiple successors that are discovered iteratively using an SMT solver following the approach outlined in Sec. 5.2.2.

Using symbolic execution, we derive the execution tree T of a **BIR** program, which is used to extract the formal models. The leaves in T are due to the **BIR halt** statement that marks the end of a complete path. A node in T is either a branching node $\mathsf{Branch}(\mathbf{pc}, \phi, \mathsf{T}_1, \mathsf{T}_2)$, where **pc** locates the conditional statement in the program, ϕ is the condition, and T_i are the sub-trees for $i \in \{1, 2\}$; or an event node $\mathsf{node}(\mathbf{pc}, \mathsf{ev}) :: \mathsf{T}'$ with the sub-tree T' and **pc** specifying where the event ev occurred. In T, an edge connects two nodes if they are in the transition relation. Since we abstract function calls and loops, we safely assume that each node in the tree can be uniquely identified by the **pc** of its statement. We define the selection operator "[]" to extract the node for a given program counter, e.g., $\mathsf{T}[\mathbf{pc}]$ will return a node indexed by **pc**.

5.3.1 To IML

We now proceed to explain how to automatically extract the IML model from protocols' **BIR** representation. Our model extraction approach relies on translating the symbolic execution tree T of the protocol under adversarial semantics into its corresponding IML model. We translate T into an executing process Q^{full} according to the rules depicted in Fig. 5.5, where $(\!|\mathsf{T}|\!)^{\iota}$ represents the compiled process, i.e. $\mathsf{Q}^{full} = (\!|\mathsf{T}|\!)^{\iota}$. Since T contains all possible execution of protocols and their interactions with the cryptographic primitives and the attacker, the extracted model includes all behaviors of the protocol at its binary representation (i.e., all attacks present at the binary level are preserved in the extracted model).

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{T} = \mathsf{Leaf} \mid \mathsf{node}(\mathbf{pc}, \mathsf{ev}) :: \mathsf{T}' \mid \mathsf{Branch}(\mathbf{pc}, \phi, \mathsf{T}_1, \mathsf{T}_2) \text{ event tree}} \\
(\!|\mathsf{Leaf}|\!)^\iota & \mapsto & \mathtt{0} \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{ev}) :: \mathsf{T}'|\!)^\iota & := & \text{events nodes} \\
(\!|\mathsf{node}(\mathbf{pc}, \tau) :: \mathsf{T}'|\!)^\iota & \mapsto & (\!|\mathsf{T}'|\!)^\iota \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{Ev}(\mathsf{d}^s_1, \ldots, \mathsf{d}^s_m)) :: \mathsf{T}'|\!)^\iota & \mapsto & \mathtt{event}(\mathsf{d}^s_1, \ldots, \mathsf{d}^s_m); (\!|\mathsf{T}'|\!)^\iota \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{In}(\mathsf{v}^s, (\mathsf{e}^s_1, \ldots, \mathsf{e}^s_m))) :: \mathsf{T}'|\!)^\iota & \mapsto & \mathtt{in}(\mathtt{c}[\mathsf{e}^s_1, \ldots, \mathsf{e}^s_m], \mathsf{v}^s); (\!|\mathsf{T}'|\!)^\iota \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{Out}(\mathsf{e}^s, (\mathsf{e}^s_1, \ldots, \mathsf{e}^s_m))) :: \mathsf{T}'|\!)^\iota & \mapsto & \mathtt{out}(\mathtt{c}[\mathsf{e}^s_1, \ldots, \mathsf{e}^s_m], \mathsf{e}^s); (\!|\mathsf{T}'|\!)^\iota \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{Cr}(\mathsf{v}^s)) :: \mathsf{T}'|\!)^\iota & \mapsto & \mathtt{let}\ \mathtt{x} = \mathsf{v}^s\ \mathtt{in}\ (\!|\mathsf{T}'|\!)^\iota \text{ where } \mathtt{x} \text{ is fresh} \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{Fr}(\mathsf{x}^s_i)) :: \mathsf{T}'|\!)^\iota & \mapsto & \mathtt{new}\ \mathsf{x}^s_i;\ (\!|\mathsf{T}'|\!)^\iota \\
(\!|\mathsf{node}(\mathbf{pc}, \mathsf{loop}(\mathsf{t}^s)) :: \mathsf{T}'|\!)^\iota & \mapsto & !^{t^s \le m}\, \mathit{LoopProc}(\mathbf{pc});\ (\!|\mathsf{T}'|\!)^\iota \\
\multicolumn{3}{l}{(\!|\mathsf{Branch}(\mathbf{pc}, \phi, \mathsf{T}_1, \mathsf{T}_2)|\!)^\iota \mapsto \mathtt{if}\ (\!|\phi|\!)^\iota\ \mathtt{then}\ (\!|\mathsf{T}_1|\!)^\iota\ \mathtt{else}\ (\!|\mathsf{T}_2|\!)^\iota} \\
(\!|\phi \in \mathbf{Bexp}|\!)^\iota & := & \mathbf{BIR} \text{ expressions} \\
(\!|\mathbf{b} \in \mathbf{Bval}|\!)^\iota & \mapsto & (\!|\mathbf{b}|\!)^\iota \in \mathtt{BS} \\
(\!|\mathbf{var}\ x|\!)^\iota & \mapsto & (\!|x|\!)^\iota \in \mathtt{Ivar} \\
(\!|\phi_1 \diamond_\mathsf{b} \phi_2|\!)^\iota & \mapsto & (\!|\phi_1|\!)^\iota (\!|\diamond_\mathsf{b}|\!)^\iota (\!|\phi_2|\!)^\iota \quad \text{Binary operations} \\
(\!|\diamond_\mathsf{b}|\!)^\iota & \mapsto & \begin{cases} \wedge & \mathbf{AND} \\ \vee & \mathbf{OR} \\ = & \mathbf{Equal} \\ + & \mathbf{Plus} \\ \ldots \end{cases} \\
(\!|\diamond_\mathsf{u} \phi'|\!)^\iota & \mapsto & (\!|\diamond_\mathsf{u}|\!)^\iota (\!|\phi'|\!)^\iota \quad \text{Unary operations} \\
(\!|\diamond_\mathsf{u}|\!)^\iota & \mapsto & \begin{cases} \neg & \mathbf{Not} \\ \bot & \text{otherwise} \end{cases} \\
(\!|\mathsf{f}(\mathsf{e}^s_1, \ldots, \mathsf{e}^s_m)|\!)^\iota & \mapsto & (\!|\mathsf{f}|\!)^\iota ((\!|\mathsf{e}^s_1|\!)^\iota, \ldots, (\!|\mathsf{e}^s_m|\!)^\iota)
\end{array}
$$

Figure 5.5: Rules for the translation of the symbolic execution tree T to IML model.

Fig.4.7 shows a fragment of the symbolic execution tree for the client of our running example. Note that, each function call is depicted with two nodes: the first node loads the address of the callee into **pc**, and the second node is the actual call, represented as an atomic transition.

Our translation converts leaf nodes into a `nil` process `0`. For internal nodes, we translate the event stored in each node into its IML counterpart. Loops are modeled using the replication operator of IML; *LoopProc* : $\xi_{\mathcal{L}} \rightarrow$ P converts the loop body into its corresponding IML process using the defined rules. Notice that we do not translate τ events. Fig. 5.5 also presents our rules to translate symbolic **BIR** expressions. Intuitively, the symbolic execution is used to symbolically compute the effects of such transitions, while the protocol model only contains the interactions with the network. Our rules to translate expressions are standard. The only interesting one is the translation of the function application, which is used, e.g., to translate memory **load**/**store**

Csec-modex's IML model

```
assume argv0 = argv0 in
                        :
new var1: fixed_20;
let nonce1 = var1 in
assume Defined(pad) in
assume len(pad) = 21 in
let xor1 = XOR((1)^[u,1]|nonce1, pad) in
                        :
let msg1 = xor1 in
out(c, msg1);
```

CryptoBap's IML model

```
new OTP_48: fixed_64;
let Conc1_66 = conc1(OTP_48) in
let XOR_70 = exclusive_or(Conc1_66,pad) in
out(c, XOR_70);
```

Figure 5.6: IML models produced by CryptoBap vs. Csec-modex for the client side of simple XOR case study.

and bitwise operations. For example, this rule translates a memory load operation **load**$(Mem, \mathbf{pa}, l)$, for $l \in \{1, 8, 16, 32, 64, 128\}$, into read$(x_1^s, l)$, where x_1^s is the fresh name chosen for the symbolic value in *Mem* at the address **pa**.

Fig.4.7 presents the IML model of the running example. In this model, c is the input and output channel, bad is the event that we release if the decryption is not successful, and enc (dec) is the encryption (resp., the decryption).

5.3.1.1 CryptoBap vs. Csec-modex IML models

Our derived IML models are simpler than Csec-modex without losing accuracy. To demonstrate this, we use the simple XOR case study from Csec-modex set of case studies. Simple XOR implements a protocol in which the one-time pad includes both protocol parties. The methodology we employ to derive the IML model from binary code differs significantly from that used in Csec-modex which leads to much simpler models. The most glaring difference between CryptoBap and Csec-modex is that our analysis produces a symbolic tree that is translated into an IML process with conditionals and replication (see Fig. 5.5) instead of a single path that is translated into a linear IML process. Even for the linear subprocesses, our toolchain produces processes that are shorter and often human-readable (at least for small case studies, e.g., XOR). This is because the Csec-modex translates each symbolic CVM process (their intermediate representation of C) into IML and performs the most simplification steps concerning the bitwise operations (concatenation, extraction, etc.) later at the translation step into CryptoVerif and ProVerif. For instance, in Csec-modex's IML model for the client side of simple XOR case study, shown in Fig. 5.6, `(1)^[u,1] | nonce1` is simplified to `conc1(nonce1)` in its CryptoVerif model. Instead, we leverage the support for simplification of these operations at symbolic execution time, because (i) support there is much more mature (ii) will benefit from future development (iii) simplified constraints can also be used for path elimination and simplification in follow-up states. In addition to this, our IML model for the client side of the simple XOR case study is more con-

$$
\begin{array}{lll}
\mathrm{T} = \mathrm{Leaf} \mid \mathrm{node}(\mathbf{pc}, \mathrm{ev}) :: \mathrm{T}' \mid \mathrm{Branch}(\mathbf{pc}, \phi, \mathrm{T}_1, \mathrm{T}_2) & & \text{event tree} \\
(\!|\mathrm{Leaf}|\!)^{sp} & \mapsto & 0 \\
(\!|\mathrm{node}(\mathbf{pc}, \mathrm{ev}) :: \mathrm{T}'|\!)^{sp} & := & \text{events nodes} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, \tau) :: \mathrm{T}'|\!)^{sp} & \mapsto & (\!|\mathrm{T}'|\!)^{sp} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, \mathrm{Ev}(e)) :: \mathrm{T}'|\!)^{sp} & \mapsto & \mathrm{event}\ e; (\!|\mathrm{T}'|\!)^{sp} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, A2P(x)) :: \mathrm{T}'|\!)^{sp} & \mapsto & \mathrm{in}(x); (\!|\mathrm{T}'|\!)^{sp} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, P2A(x)) :: \mathrm{T}'|\!)^{sp} & \mapsto & \mathrm{out}(x); (\!|\mathrm{T}'|\!)^{sp} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, FCall(f, x_1, \ldots, x_n, y)) :: \mathrm{T}'|\!)^{sp} & \mapsto & \mathrm{let}\ y = f(x_1, \ldots, x_n)\ \mathrm{in}\ (\!|\mathrm{T}'|\!)^{sp}\ \mathrm{else}\ 0 \\
\quad (\!|\mathrm{node}(\mathbf{pc}, \mathrm{Asn}(x, \mathrm{e}^s)) :: \mathrm{T}'|\!)^{sp} & \mapsto & \mathrm{let}\ x = (\!|\mathrm{e}^s|\!)^{sp}\ \mathrm{in}\ (\!|\mathrm{T}'|\!)^{sp} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, SFr(n)) :: \mathrm{T}'|\!)^{sp} & \mapsto & \mathrm{new}\ n;\ (\!|\mathrm{T}'|\!)^{sp} \\
\quad (\!|\mathrm{node}(\mathbf{pc}, \mathrm{Loop}) :: \mathrm{T}'|\!)^{sp} & \mapsto & !\ (\!|\mathrm{T}'|\!)^{sp} \\
(\!|\mathrm{Branch}(\mathbf{pc}, \phi, \mathrm{T}_1, \mathrm{T}_2)|\!)^{sp} & \mapsto & (\!|\mathrm{T}_1|\!)^{sp} + (\!|\mathrm{T}_2|\!)^{sp} \\
(\!|\phi, \mathrm{e}^s \in \mathbf{Bexp}|\!)^{sp} & := & \mathbf{BIR}\ \text{expressions} \\
(\!|\mathbf{b} \in \mathbf{Bval}|\!)^{sp} & \mapsto & 'b' \in \mathcal{N}_{\mathsf{pub}} \\
(\!|\mathbf{var}\ x|\!)^{sp} & \mapsto & x \in \mathcal{V} \\
(\!|\phi_1 \diamond_{\mathbf{b}} \phi_2|\!)^{sp} & \mapsto & (\!|\diamond_{\mathbf{b}}|\!)^{sp}((\!|\phi_1|\!)^{sp}, (\!|\phi_2|\!)^{sp}) \quad \text{Binary operations} \\
(\!|\diamond_{\mathbf{b}}|\!)^{sp} & \mapsto & \begin{cases} \text{equal} & \mathbf{Equal} \\ \text{plus, mult, } \ldots & \mathbf{Plus}, \mathbf{Mult}, \ldots \end{cases} \\
(\!|\diamond_{\mathbf{u}} \phi'|\!)^{sp} & \mapsto & (\!|\diamond_{\mathbf{u}}|\!)^{sp}((\!|\phi'|\!)^{sp}) \quad \text{Unary operations} \\
(\!|\diamond_{\mathbf{u}}|\!)^{sp} & \mapsto & \begin{cases} \text{not} & \mathbf{Not} \\ \bot & \text{otherwise} \end{cases} \\
(\!|\mathrm{f}(\mathrm{e}^s_1, \ldots, \mathrm{e}^s_m)|\!)^{sp} & \mapsto & (\!|\mathrm{f}|\!)^{sp}((\!|\mathrm{e}^s_1|\!)^{sp}, \ldots, (\!|\mathrm{e}^s_m|\!)^{sp})
\end{array}
$$

Figure 5.7: Rules for the translation of the symbolic execution tree T to SAPIC$^-$ model.

cise since several assumptions made in CSEC-MODEX's IML model were unnecessary for verifying this particular case study. Fig.5.6 presents the different IML models produced by CRYPTOBAP and CSEC-MODEX for the client side of simple XOR case study.

5.3.2 To SAPIC$^-$

SAPIC$^-$ has the same syntax as SAPIC$^+$, but its semantics remove the DY attacker: instead of invoking SAPIC$^+$'s internal DY deduction relation, communication (in, out) in SAPIC$^-$ emits events ($A2P$, $P2A$) that synchronizes with an outside attacker. Apart from DY attacker and library events (see Sec. 2.3.2 and Sec. 2.3.3), $\mathrm{Ev}(e)$ signifies the occurrence of a visible event e, Loop denotes the start of a loop, and $\mathrm{Asn}(x, \mathrm{e}^s)$ represents the assignment of the SBIR expression e^s to the variable x.

The protocol model is obtained by translating T into its SAPIC$^-$ model. We translate T using the rules in Fig. 5.7 to $(\!|\mathrm{T}|\!)^{sp}$. We translate leaves into a *nil* process 0, and the event ev from the event nodes into their corresponding SAPIC$^-$ construct. The branch-

ing nodes of T (i.e., $\mathsf{Branch}(\mathbf{pc}, \phi, \mathsf{T}_1, \mathsf{T}_2)$) are translated into a non-deterministic choice ($+$) between (the translation of) both possible paths. If these branches are not already pruned by symbolic execution, there might still be bit-level conditions that are relevant for the protocol verifier, but not sufficient to prune the branch during symbolic execution. While we did not encounter this case, it would be possible to translate to $(\mathsf{event}\ E_1; (\!|\mathsf{T}_1|\!)^{sp} + \mathsf{event}\ E_2; (\!|\mathsf{T}_2|\!)^{sp})$ for E_1, E_2 some unique events. As Sapic$^+$ supports restricting the trace set based on formulas, we can reflect necessary conditions that are expressible in these tools. For instance, if the condition was $x \oplus y = \mathbf{0}$, we may require that the occurrence of E_1, i.e., a traversal into the positive branch, entails that $y \neq x+1$, if that helps exclude a false attack. In all our case studies, most paths are pruned by our symbolic execution engine and the remaining not require such a refinement. Nevertheless, this feature would be easy to add (and prove correct for any condition entailed by the combined deduction relation).

In order to illustrate the methodology employed for model extraction, the extracted Sapic$^-$ process of the **BIR** program from example 3.2 is presented in 6.2.

5.3.3 With Observations

To analyse protocol implementations for side channel leakages, we derive a symbolic execution tree T from the **BIR** program and use it to extract the protocol's Sapic$^-$ model. The only difference in the representation of the tree T compared to the last two sections is the branching node ($\mathsf{Branch}(\mathbf{Cnd}(\phi, \mathbf{a_1}, \mathbf{a_2}), \mathsf{T}_1, \mathsf{T}_2)$) where also contain the respective addresses of subtrees $\mathbf{a_i}$. Having constructed T, we extract the protocol model by translating T into its Sapic$^-$ model using the rules in Fig. 5.7. Compare to previous section, we also translate the attacker observations $\mathbf{Ld}(\mathbf{a})$, $\mathbf{St}(\mathbf{a})$, and $\mathbf{Pc}(\mathbf{a})$ within event nodes to the $\mathsf{out}((\!|\mathbf{a}|\!)^{sp})$ constructs. In addition, the T's branching nodes are translated into Sapic$^-$ constructs as follows:

$$(\!|\mathsf{Branch}(\mathbf{Cnd}(\phi, \mathbf{a_1}, \mathbf{a_2}), \mathsf{T}_1, \mathsf{T}_2)|\!)^{sp} \quad \mapsto \quad \mathsf{out}((\!|\phi|\!)^{sp}); \mathsf{if}\ (\!|\phi|\!)^{sp}\ \mathsf{then}\ \mathsf{out}((\!|\mathbf{a_1}|\!)^{sp}); (\!|\mathsf{T}_1|\!)^{sp} \ \mathsf{else}\ \mathsf{out}((\!|\mathbf{a_2}|\!)^{sp}); (\!|\mathsf{T}_2|\!)^{sp}$$

The observations $\mathbf{Cnd}(\phi, \mathbf{a_1}, \mathbf{a_2})$ are preserved in the branching nodes of T, where each condition ϕ is a symbolic **BIR** expression translated into an equivalent Sapic$^-$ term. To maintain the attacker's observations during simplification (cf. Sec. 5.3.4), we first disclose the translation of ϕ to the attacker before translating the branching node into the Sapic$^-$ conditional construct, where $(\!|\phi|\!)^{sp}$ serves as its condition. Each branch of this construct includes the translation of the corresponding addresses $\mathbf{a_i}$—also revealed to the attacker—and the translation of respective subtrees T_i. Note that $\mathbf{a_i}$ and conditional observations are either normal or shadow observations.

The translation of branching nodes differs from Sec. 5.3.2, where we translate each

Table 5.1: Simplification rules

SAPIC⁻ Process	if ϕ then P else P	let ϕ in 0 else 0	let $x = e$ in P s.t. $x \notin vars(P)$
Simplified	P	0	P

branching node into a non-deterministic choice between both branches, as an equivalence between Dolev-Yao terms in SAPIC⁻ does not necessarily encompass all possible equivalences between bitstrings. At this point, we give up soundness to have a fast attack-finding procedure. Fig. 5.7 presents the standard rules for translating expressions. The more interesting case is the translation of function applications used, e.g., to translate memory load/store and bitwise operations. For instance, a **load**(mem, a, l), where $l \in \{1, 8, 16, 32, 64, 128\}$, translates to load$(mem, a)$, with mem and a representing symbolic values for the memory and the address respectively.

The fourth column in Fig. 4.9 shows the extracted SAPIC⁻ process of the running example. An attacker capable of measuring microarchitectural states can detect if the MAC check fails (by observing program counter $0x200$) or succeeds (by observing program counter $0xac$). However, this is not visible without accounting for side-channel observations, as the program halts when the MAC check fails.

5.3.4 Simplification

Extracting the model of a protocol with every detail, including the memory operations (load/store) and possible attacker observations, produces a model that is too big for state-of-the-art protocol provers. Therefore, to reduce the model complexity, we have sed abstractions in **BIR**, pruned paths at SBIR, and applied a few simplification rules at the SAPIC⁻ level, which are displayed in Table 5.1.

At the **BIR** level, we can determine the desired abstraction level and identify which protocol code segments should be prioritized. We also introduce a *storage abstraction* (similar to how Klee abstracts files, pipes and terminals [56]) to abstract the SQLite database engine's, including database creation, outlining the tables, read/write operations related to a database file. Specifically, WhatsApp stores session keys in the SQLite database.

During transpilation of binary code into **BIR**, we insert assertions that encode *well-formedness invariants* of executions, e.g., after each stack operation we add **assert**$(sp_{low} \leq sp \leq sp_{high})$ to confine the stack pointer to the current frame. By construction, each inserted **assert**(χ) is an invariant for executions of the original binary that start from well-formed initial states. Consequently, any SBIR path that violates such an assertion is infeasible. We therefore cause the path to fail at that point and prune its suffix. This does not affect our side-channel analysis because the inserted assertions hold for

all executions from well-formed states, and our observation semantics are guarded. Hence, assertion-violating paths cannot produce observations.

At the SBIR level, further simplifications are also possible. An example is pruning paths that the SMT solver marks as unreachable, like branches of conditionals that are always false. While for functional correctness analysis, eliminating such paths is possible, we cannot apply this simplification when we look for side-channel leakages as unreachable branches can leak during speculative execution.

In the extracted SAPIC$^-$ model, nil processes indicate the end of a complete path. When a let construct results in nil processes in both directions, it can also be removed and substituted with a nil process (0). Moreover, for a conditional that involves paths following the same sequence of constructs, like if ϕ then P else P, the conditional can be replaced with the corresponding constructs in those paths, P. In order to ensure that these simplifications would not affect the soundness of our side channel analysis, we preserve the observations related to the conditional (if any) and add it to the simplification results.

Live variable analysis is a data-flow analysis used in compiler optimization to identify live variables at every point in a program [95]. In this analysis, a variable x is live at a given point p in the program if x will be used along some path starting from p. An important application of live variable information arises when we aim to eliminate a let construct. We use this technique to determine whether the variable x used in the let construct let $x = e$ in P remains alive within the sub-process P. If so, we retain the let construct, otherwise, we substitute it to P. *vars* utilized in Table 5.1, provides a set of variables for a given process.

Finally, observational models like $\mathcal{M}_{ct}$ require making the program counter of instructions (i.e., **Pc**(**a**)) visible to the attacker. When extracting a formal model of a program annotated with such observational models, we generate an out($\langle$**a**$\rangle^{sp}$) for each instruction (see Fig.5.7). However, the large number of output constructs makes DEEPSEC computation expensive without a significant benefit, mainly when they originate from the same branch. Therefore, leaking the program counters of conditional branches is sufficient and other program counter leakages can be eliminated. This greatly simplifies the extracted SAPIC$^-$ process (e.g., see Table 7.4).

5.3.5 Detecting Speculative Leak with DEEPSEC

We use the protocol verifier DEEPSEC [61] to identify side-channel leakages that can be triggered by a network attacker. DEEPSEC specializes in analyzing privacy-related security properties modeled through trace equivalence. Specifically, DEEPSEC checks whether two protocol processes, P_1 and P_2, are indistinguishable by a Dolev-Yao attacker capable of observing traces of protocol execution.

For example, in the BAC protocol, an attacker can exploit observable differences

in control-flow paths (e.g., out($0xac$) versus out($0x200$) in Fig. 4.9) to infer structural information about messages. By replaying previously observed messages, the attacker can differentiate between successful and unsuccessful MAC verifications, thereby breaking BAC's intended privacy guarantees.

DeepSec is particularly well-suited for such analyses, as it supports establishment of equivalence properties. Formally, for protocol processes P_1 and P_2, DeepSec computes whether $P_1 \approx P_2$ holds, indicating their indistinguishability. Unlike other tools, DeepSec guarantees termination with correct results when provided with sufficient computational resources, which makes it reliable for analyzing trace equivalences.

To address vulnerabilities introduced by speculative execution, we evaluate conditional noninterference, as in Def.4.1. This property distinguishes between leaks present in the $\mathcal{M}_{ct}$ model and additional leaks that emerge under the speculative execution model $\mathcal{M}_{spec}$. If protocol properties are preserved under $\mathcal{M}_{ct}$ but violated under $\mathcal{M}_{spec}$, this indicates a speculative leak.

We check the conditional noninterference with DeepSec by encoding the 4-trace CNI implication as two DeepSec equivalence checks on the same processes, once with the reference observations $\mathcal{M}_{ct}$ and once with the refined observations $\mathcal{M}_{spec}$. $P_1 \sim_{\mathcal{M}_{ct}} P_2$ but not $P_1 \sim_{\mathcal{M}_{spec}} P_2$ witnesses a speculative-only leak. Following this, if enough resources (time and memory) are available, DeepSec confirm both processes as equivalent or supply two traces showing that they are not.

Chapter 6

Soundness

In the preceding chapters, we introduced the theoretical foundations of our verification framework, detailed the design of the CryptoBap framework, and demonstrated its applicability through model extraction from binary protocol implementations. In this chapter, we establish the soundness of our approach: we prove that security properties verified on the extracted models indeed hold for the original binary implementations under the respective threat models. Our proofs are carried out in two complementary settings. First, in the computational setting (Sec. 6.1), we argue correctness with respect to a probabilistic, time-bounded adversary, ensuring that the binary execution preserves the intended cryptographic guarantees. Second, in the symbolic setting (Sec. 6.2), we formalize the end-to-end correspondence between binary-level behaviors and their abstract representations in protocol verification tools, leveraging the compositional framework developed in Part I. Together, these results provide the theoretical assurance that the verification pipeline is both faithful to the analyzed binaries and suitable for reasoning about their security properties at higher abstraction levels.

6.1 In Computational Setting

The extracted IML model should preserve the **BIR** program's behaviors to ensure that we can transfer the verified properties back to the binary of protocols. Fig. 6.1 shows the interconnection between different layers in our proofs.

6.1.1 Translation to IML

To show that our extracted IML model preserves the semantics of the protocols' binary, we need to prove that our translation from a crypto-aware symbolic execution tree into an IML process is sound, i.e., *for each path in the symbolic execution tree there is an equivalent* IML *execution trace.*

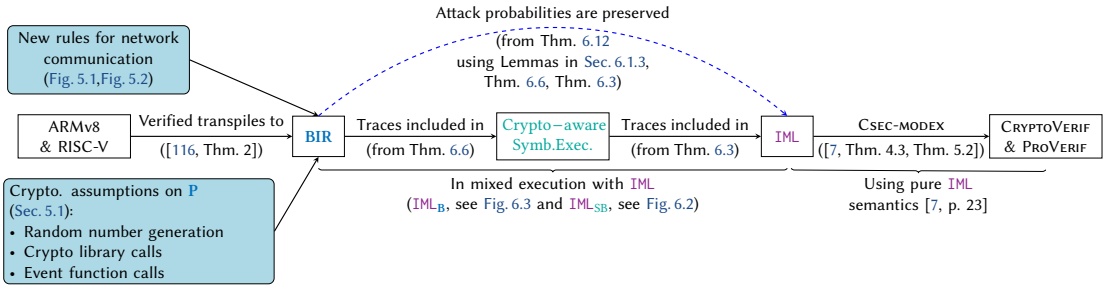


Figure 6.1: Organization of our proofs in computational setting.

Our symbolic execution supports communication with the attacker, which, like honest protocol parties given by specification, is represented as an IML process. Thus, we need to prove soundness in the context of an IML process, i.e., that each execution trace obtained by symbolically executing the **BIR** program in parallel with an IML attacker has an equivalent IML trace where the translated processes run in parallel with the same attacker. Our strategy to prove this is to construct an IML-SBIR, $\mathtt{IML}_{\mathrm{SB}}$, mixed execution semantics to facilitate the communication of **BIR** programs and the IML attacker. $\mathtt{IML}_{\mathrm{SB}}$ is generic and considers **BIR** programs and IML processes as independent entities running in parallel and communicating through a channel.

The IML process already describes the parallel execution of parties and how they share secrets. We only need to integrate **BIR** into this framework. Therefore, we extend IML with a construct $\mathtt{run}(\mathbf{pc}, (y_1, \ldots, y_m))$ to initialize **BIR** symbolic memory and transfer control to the **BIR** program specified by the **pc**. To share the secrets, we generate fresh symbolic values $y_1^s, \ldots, y_m^s$ and store them in the environment η^s of the **BIR** program.

In the following, we use $\mathtt{I}\{\mathbf{P}\}$ as a pair of an IML process I extended with the run construct and a **BIR** program **P** that defines the entry points therein. Slightly misusing notation, $\mathtt{I}\{s_i^s\}$ also denotes states of the mixed semantics.

Fig. 6.2 shows the operational semantics of $\mathtt{IML}_{\mathrm{SB}}$ which combines IML input and output processes [7, p. 23] with the transition relations of SBIR in Fig. 5.2. In the figure, rules I*to*SB and SB*to*I define the communication between the symbolic **BIR** program and the IML process. Using these rules a protocol participant can receive a sent message if its channel identifiers have the same evaluation as the channel identifiers of the sender. To send a message, i.e., when **pc** is in the label set $\xi_{\mathcal{A}_s}$, we first fetch the symbolic value e^s from the memory location r_0, truncate the interpretation of the message according to the maximum length of the IML channel c, and then place it in the IML environment η'. Truncation is a requirement from CSEC-MODEX's correctness proof for the translation to PROVERIF and CRYPTOVERIF. As the attacker's polynomial bound is chosen after the process, the attacker could send a large message that the process runs out of time reading it. When **pc** is in $\xi_{\mathcal{A}_r}$ and the **BIR** program receives input from the

$$\frac{\mathbf{pc} \in \mathbf{L}_{\mathcal{N}} \uplus \xi_{\mathrm{Op}} \uplus \xi_{\mathcal{R}} \uplus \xi_{\mathcal{E}} \qquad (\Pi^s, \eta^s, \mathbf{pc}) \xrightarrow{o^s} (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}')}{\mathbf{P} \vdash (\Pi^s, \eta^s, \mathbf{pc}), Q^{\iota\times s} \xrightarrow{o^s}\!\!\!\!\rightarrow_{1,\mathrm{H}} (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), Q^{\iota\times s}}\ normal_{\mathrm{IS}}$$

$$\frac{\begin{array}{c}\forall j \le m : [\![\mathtt{y_j}]\!]_{\eta^\iota} = \mathtt{b_j} \neq \bot \quad (\mathtt{y}^s_1, ..., \mathtt{y}^s_m) \notin range(\eta^s_0) \quad \forall j \le m : \mathrm{H}(\mathtt{y}^s_j) = \mathtt{b_j} \neq \bot \\ \forall j \le m : (\eta^s_{j+1}, \mathbf{a_i}) = mstore(\eta^s_j, heap, Mem, \mathtt{y}^s_j, 128)\end{array}}{\mathbf{P} \vdash (\eta^\iota, \mathtt{run}(\mathbf{pc}, (\mathtt{y}_1, ..., \mathtt{y}_m))), Q^{\iota\times s} \twoheadrightarrow\!\!\!\!\rightarrow_{1,\mathrm{H}} (\Pi^s, \eta^s_{m+1}[r_0 \mapsto \mathbf{a}_1, ..., r_m \mapsto \mathbf{a}_m], \mathbf{pc}), Q^{\iota\times s}}\ run_{\mathrm{IS}}$$

$$\frac{\begin{array}{c}\mathbf{pc} \in \xi_{\mathcal{A}_s} \quad \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) \quad Q^{\iota\times s\prime} = \{(\Pi^s, \eta^s, \mathbf{pc}')\} \quad \mathbf{a} = \eta^s[r_0] \quad \mathtt{e}^s = mload(\eta^s, \mathbf{a}) \\ \mathrm{H}(\mathtt{e}^s) = \mathtt{b} \neq \bot \quad \mathtt{b}' = truncate(\mathtt{b}, maxlen(\mathtt{c})) \quad \forall j \le m : \mathrm{H}(\eta^s[\mathtt{e}^s_j]) = \mathtt{b_j} \neq \bot \\ \exists!(\eta^\iota, \mathtt{Q}) \in Q^{\iota\times s} : (\mathtt{Q} = \mathtt{in}(\mathtt{c}[\mathtt{e}_1, ..., \mathtt{e}_m], \mathtt{x}); \mathtt{P} \ \wedge \ \forall j \le m : [\![\mathtt{e_j}]\!]_{\eta^\iota} = \mathtt{b_j} \neq \bot) \quad Q^{\iota\times s\prime\prime} = \{(\eta^\iota, \mathtt{Q})\}\end{array}}{\mathbf{P} \vdash ((\Pi^s, \eta^s, \mathbf{pc}), \mathtt{c}[\mathtt{e}^s_1, ..., \mathtt{e}^s_m]), Q^{\iota\times s} \xrightarrow{\mathrm{Out}(\mathtt{e}^s, (\mathtt{e}^s_1, ..., \mathtt{e}^s_m))}\!\!\!\!\rightarrow_{1,\mathrm{H}} (\eta^\iota[\mathtt{x} \mapsto \mathtt{b}'], \mathtt{P}), Q^{\iota\times s} \uplus Q^{\iota\times s\prime} \setminus Q^{\iota\times s\prime\prime}}\ \mathrm{SB}to\mathrm{I}$$

$$\frac{\begin{array}{c}\forall j \le m : [\![\mathtt{e_j}]\!]_{\eta^\iota} = \mathtt{b_j} \neq \bot \quad [\![\mathtt{e}]\!]_{\eta^\iota} = \mathtt{b} \neq \bot \quad \mathtt{b}' = truncate(\mathtt{b}, maxlen(\mathtt{c})) \quad \mathrm{H}(\mathtt{e}^s) = \mathtt{b}' \\ \mathtt{e}^s \notin range(\eta^s) \quad \exists!((\Pi^s, \eta^s, \mathbf{pc}), \mathtt{c}[\mathtt{e}^s_1, ..., \mathtt{e}^s_m]) \in Q^{\iota\times s} \ : \ (\mathbf{pc} \in \xi_{\mathcal{A}_r} \ \wedge \ \forall j \le m \ : \ \mathrm{H}(\eta^s[\mathtt{e}^s_j]) = \mathtt{b_j} \neq \bot) \\ (\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, \mathtt{e}^s, 128) \quad \eta^{s\prime\prime} = \eta^{s\prime}[r_0 \mapsto \mathbf{a}] \quad \mathbf{pc}' = ret(\Pi^s, \eta^s, \mathbf{pc}) \\ Q^{\iota\times s\prime} = reduce(\{(\eta^\iota, \mathtt{Q})\}) \quad Q^{\iota\times s\prime\prime} = \{((\Pi^s, \eta^s, \mathbf{pc}), \mathtt{c}[\mathtt{e}^s_1, ..., \mathtt{e}^s_m])\} \quad Q^{\iota\times s\prime\prime\prime} = Q^{\iota\times s} \uplus Q^{\iota\times s\prime} \setminus Q^{\iota\times s\prime\prime}\end{array}}{\mathbf{P} \vdash (\eta^\iota, \mathtt{out}(\mathtt{c}[\mathtt{e}_1, ..., \mathtt{e}_m], \mathtt{e}); \mathtt{Q}), Q^{\iota\times s} \xrightarrow{\mathrm{In}(\mathtt{e}^s, (\mathtt{e}^s_1, ..., \mathtt{e}^s_m))}\!\!\!\!\rightarrow_{1,\mathrm{H}} ((\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}'), \mathtt{c}[\mathtt{e}^s_1, ..., \mathtt{e}^s_m]), Q^{\iota\times s\prime\prime\prime}}\ \mathrm{I}to\mathrm{SB}$$

Figure 6.2: The mixed semantics of SBIR and IML shown by $\mathrm{IML}_{\mathrm{SB}}$.

IML channel c, we receive the truncated bitstring $\mathtt{b}'$ from an IML state and generate a fresh symbolic value $\mathtt{e}^s$ such that the interpretation of $\mathtt{e}^s$ is equal to bitstring $\mathtt{b}'$. Then, we store the symbolic value $\mathtt{e}^s$ in the memory and return its address in r_0.

We use the standard notion of *trace inclusion* to show the translations' soundness (see Theorem 6.3), i.e., the set of $\mathrm{IML}_{\mathrm{SB}}$ execution traces is a subset of the IML execution traces. To prove this formally, we define a simulation relation $\sim_{\mathrm{H},(\!|.|\!)^\iota} \subseteq S^{\iota\times s} \times \mathrm{S}^\iota$ between states/events of these two abstraction layers and show that it is preserved by the single-step executions. The simulation relation, $\mathrm{I}\{s^s_i\} \sim_{\mathrm{H},(\!|.|\!)^\iota} s^\iota_i$, checks if (i) the IML output process in the given IML state is the correct translation of the symbolic state in T according to the rules in Fig. 5.5, i.e., $s^\iota_i.\mathrm{P} = (\!|\mathrm{T}[s^s_i.\mathbf{pc}]|\!)^\iota$, and (ii) the environments of the two abstractions are related through the interpretation H, i.e., for all $\mathtt{x}^s \in dom(\mathrm{I}\{s^s_i\}.\eta^s)$ there are $\mathtt{x} \in dom(s^\iota_i.\eta^\iota)$ and an H s.t. $\mathrm{H}(\mathrm{I}\{s^s_i\}.\eta^s[\mathtt{x}^s]) = s^\iota_i.\eta^\iota[\mathtt{x}]$. Theorem 6.1 shows that the initial states of $\mathrm{IML}_{\mathrm{SB}}$ and the derived IML process are in the relation.

Lemma 6.1 ($\mathrm{IML}_{\mathrm{SB}}$-IML Initial State Equivalence). *For a symbolic execution tree* T *of the* BIR *program* P*, an* IML *process* I *and any* $k \in \mathbb{N}$ *the size of the random memory, let* $\mathrm{I}\{s^s_0\} = (True, \eta^s_0[\mathbf{RM} \mapsto \mathrm{rm}^s_k, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$ *be an initial symbolic state in* $\mathrm{IML}_{\mathrm{SB}}$ *and* $s^\iota_\emptyset = (\eta^\iota_\emptyset, \mathtt{Q}^{full})$ *the corresponding initial* IML *state. Then, for all* H*:* $\mathrm{I}\{s^s_0\} \sim_{\mathrm{H},(\!|.|\!)^\iota} s^\iota_\emptyset$.

Proof. By construction of our symbolic tree we know that for $\mathbf{pc_0}$, we have $(\!|\mathbb{T}|\!)^\iota$. Thus, from translation rule $(\!|\mathbb{T}|\!)^\iota = \mathbb{Q}^{full}$ in Fig. 5.5 we get that $s^\iota_0.\mathtt{P} = \mathbb{Q}^{full}$. Then, we can conclude that $(True, \eta^s_0[\mathbf{RM} \mapsto \mathrm{rm}^s_k, \mathbf{r_k} \mapsto 0], \mathbf{pc_0}) \sim_{\mathrm{H},(\!|.|\!)^\iota} (\eta^\iota{}_\emptyset, \mathbb{Q}^{full})$. □

Next, we show that single-step transitions preserve the simulation relation.

Lemma 6.2 ($\mathtt{IML_{SB}}$-$\mathtt{IML}$ State/Event Equivalence). *Let* $\mathbf{P}$ *be a* **BIR** *program and* $\mathtt{I}$ *be an* $\mathtt{IML}$ *process, then, for all* $s^\iota_i, \mathtt{I}\{s^s_i\}, \mathtt{I}\{s^s_j\}$ *and* H *s.t.* $\mathtt{I}\{s^s_i\} \sim_{\mathrm{H},(\!|.|\!)^\iota} s^\iota_i$ *and* $\mathtt{I}\{s^s_i\} \xrightarrow{o^{\iota\times s}}{}^{+}_{\mathrm{p},\mathrm{H}} \mathtt{I}\{s^s_j\}$, *there exist an* H' *and* s^ι_j *s.t.* $\mathrm{H} \subseteq \mathrm{H}'$, $s^\iota_i \xrightarrow{o^\iota}{}_{\mathrm{p}} s^\iota_j$ *and* $\mathtt{I}\{s^s_j\} \sim_{\mathrm{H}',(\!|.|\!)^\iota} s^\iota_j$ *and if* $\mathtt{o}^\iota \neq \bot$ *then* $o^{\iota\times s} =_{\mathrm{H}'} \mathtt{o}^\iota$.

Proof of Theorem 6.2 is done by a case split on the type of label sets in the **BIR** program $\mathbf{P}$.

Proof. We prove the statement by a case split based on the type of the program counter $\mathtt{I}\{s^s_i\}.\mathbf{pc}$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{E}}$
  Since $\mathtt{I}\{s^s_i\} \sim_{\mathrm{H},(\!|.|\!)^\iota} s^\iota_i$, we know that $s^\iota_i.\mathtt{P} = (\!|\mathbb{T}[s^s_i.\mathbf{pc}]|\!)^\iota$. Based on the translation rules in Fig. 5.5, we get that $s^\iota_i.\mathtt{P} = \mathtt{event}(\mathrm{d}^s_1, \ldots, \mathrm{d}^s_m); (\!|\mathbb{T}'|\!)^\iota$. Also, by construction of the symbolic execution tree we know that there exist $\mathbb{T}_0$ and $\mathbb{T}_1$ such that $\mathbb{T} = \mathbb{T}_0 :: (\mathbf{pc}, \mathrm{Ev}(\mathrm{d}^s_1, \ldots, \mathrm{d}^s_m)) :: (\mathbf{pc}', \mathrm{ev}) :: \mathbb{T}_1$. Therefore, based on the operational semantics of $\mathtt{IML}$ processes [7, p. 23] and translation rules in Fig. 5.5, we get $s^\iota_i \xrightarrow{\mathrm{ev}(\mathrm{b}_1,\ldots,\mathrm{b}_m)}{}_1 s^\iota_j$ and $s^\iota_j = (\eta^\iota, (\!|\mathbb{T}[s^s_j.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$.
  Based on the semantics of $\mathtt{IML_{SB}}$, Fig. 6.2, we find $\mathtt{I}\{s^s_j\} = (\Pi^s, \eta^s, \mathbf{pc}'), \mathcal{Q}^{\iota\times s}$ that is in the relation $\mathtt{I}\{s^s_i\} \xrightarrow{o^{\iota\times s}}{}_{1,\mathrm{H}} \mathtt{I}\{s^s_j\}$ such that $o^{\iota\times s} = \mathrm{Ev}(\mathrm{d}^s_1, \ldots, \mathrm{d}^s_m)$.
  Finally, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(\mathrm{d}^s_j) = \mathrm{b}_j$ for $0 < j \leq m$. Therefore, we can conclude that $\mathrm{Ev}(\mathrm{d}^s_1, \ldots, \mathrm{d}^s_m) =_{\mathrm{H}'} \mathrm{ev}(\mathrm{b}_1, \ldots, \mathrm{b}_m)$ and $(\Pi^s, \eta^s, \mathbf{pc}'), \mathcal{Q}^{\iota\times s} \sim_{\mathrm{H}',(\!|.|\!)^\iota} (\eta^\iota, (\!|\mathbb{T}[s^s_i.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{C}}$
  Since $\mathtt{I}\{s^s_i\} \sim_{\mathrm{H},(\!|.|\!)^\iota} s^\iota_i$, we know that $s^\iota_i.\mathtt{P} = (\!|\mathbb{T}[s^s_i.\mathbf{pc}]|\!)^\iota$. Based on the translation rules in Fig. 5.5, we get that $s^\iota_i.\mathtt{P} = \mathtt{let\ x = v^s\ in}\ (\!|\mathbb{T}'|\!)^\iota$. Also, by construction of the symbolic execution tree we know that there exist $\mathbb{T}_0$ and $\mathbb{T}_1$ such that $\mathbb{T} = \mathbb{T}_0 :: (\mathbf{pc}, \mathrm{Cr}(\mathrm{v}^s)) :: (\mathbf{pc}', \mathrm{ev}) :: \mathbb{T}_1$.
  Therefore, based on the operational semantics of $\mathtt{IML}$ processes [7, p. 23] and translation rules in Fig. 5.5, we get that $s^\iota_i \longrightarrow\!\!\!\!\!\rightarrow_1 s^\iota_j$ and $s^\iota_j = (\eta^\iota[\mathrm{x} \mapsto \mathrm{b}], (\!|\mathbb{T}[s^s_j.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$. Based on the semantics of $\mathtt{IML_{SB}}$, Fig. 6.2, we find $\mathtt{I}\{s^s_j\} = (\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}'), \mathcal{Q}^{\iota\times s}$

that is in the transition relation $\mathtt{I}\{s_i^s\} \xrightarrow{o'^{\times s}}\!\!\!\!\rightarrow\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$ such that $o'^{\times s} = \mathrm{Cr}(v^s)$ and $\eta^{s\prime\prime} = \eta^{s\prime}[r_0 \mapsto \mathbf{a}]$ where $(\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap_{\mathrm{Op}}, Mem_{\mathrm{Op}}, v^s, 128)$.

Finally, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(v^s) = \mathtt{b}$. Therefore, we can conclude that $(\Pi^s, \eta^{s\prime}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathcal{Q}'^{\times s} \sim_{\mathrm{H}', (\!|.|\!)^\iota} (\eta'[x \mapsto b], (\!|\mathrm{T}[s_i^s.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{R}}$
  Since $\mathtt{I}\{s_i^s\} \sim_{\mathrm{H}, (\!|.|\!)^\iota} s_i^\iota$, we know that $s_i^\iota.\mathtt{P} = (\!|\mathrm{T}[s_i^s.\mathbf{pc}]|\!)^\iota$. Based on the translation rules in Fig. 5.5, we get that $s_i^\iota.\mathtt{P} = \mathtt{new}\ x^s;\ (\!|\mathrm{T}'|\!)^\iota$. Also, by construction of the symbolic execution tree we know that there exist T_0 and T_1 such that $\mathrm{T} = \mathrm{T}_0 :: (\mathbf{pc}, \mathrm{Fr}(x^s)) :: (\mathbf{pc}', ev) :: \mathrm{T}_1$.

  Therefore, based on the operational semantics of IML processes [7, p. 23] and translation rules in Fig. 5.5, we get that $s_i^\iota \xrightarrow{\mathtt{fr(b)}}\!\!\!\!\rightarrow\!\!\!\!\rightarrow_{\frac{1}{2^n}} s_j^\iota$ and $s_j^\iota = (\eta'[x^s \mapsto \mathtt{b}], (\!|\mathrm{T}[s_j^s.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$. Based on the semantics of $\mathtt{IML}_{\mathrm{SB}}$, Fig.6.2, we find $\mathtt{I}\{s_j^s\} = (\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}'), \mathcal{Q}'^{\times s}$ that is in the relation $\mathtt{I}\{s_i^s\} \xrightarrow{o'^{\times s}}\!\!\!\!\rightarrow\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$ such that $o'^{\times s} = \mathrm{Fr}(x^s)$ and $\eta^{s\prime\prime} = \eta^{s\prime}[r_0 \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta^s[\mathbf{r_k}] + l]$ where $(\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap, Mem, x^s, 128)$ and $x^s = \mathfrak{R}^s(\eta^s, n)$.

  Finally, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(x^s) = \mathtt{b}$. Therefore, we can conclude that $(\Pi^s, \eta^{s\prime}[r_0 \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta^s[\mathbf{r_k}] + l], \mathbf{pc}'), \mathcal{Q}'^{\times s} \sim_{\mathrm{H}', (\!|.|\!)^\iota} (\eta'[x^s \mapsto \mathtt{b}], (\!|\mathrm{T}[s_i^s.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$ and $\mathtt{fr(b)} =_{\mathrm{H}'} \mathrm{Fr}(x^s)$.

- $\mathbf{pc_1} \in \mathbf{L}_{\mathcal{A}_s} \wedge \mathbf{pc_2} \in \mathbf{L}_{\mathcal{A}_r}$
  In this case, since $\mathbf{pc_1} \in \mathbf{L}_{\mathcal{A}_s}$, we have an IML output process `out` in the IML state s_i^ι. Based on IOut rule in the operational semantics of IML processes [7, p. 23], we get that there is an IML input process `in` in the multiset of executing input processes that is ready to receive input from the channel. Therefore, there exists a node in our symbolic execution tree such that its program counter is in the label set $\mathbf{L}_{\mathcal{A}_r}$.

  More formally, since $\mathtt{I}\{s_i^s\} \sim_{\mathrm{H}, (\!|.|\!)^\iota} s_i^\iota$, we know that $s_i^\iota.\mathtt{P} = (\!|\mathrm{T}[s_i^s.\mathbf{pc_1}]|\!)^\iota$. Based on the translation rules in Fig. 5.5, we get that $s_i^\iota.\mathtt{P} = \mathtt{out}(\mathtt{c}[e_1^s, \ldots, e_m^s], e^s); (\!|\mathrm{T}'|\!)^\iota$. Based on the operational semantics of IML processes [7, p. 23], we get that $s_i^\iota \rightarrow\!\!\!\!\rightarrow_1 s_j^\iota$ and $s_j^\iota = (\eta'[v^s \mapsto \mathtt{b}'], \mathtt{P}'), \mathcal{Q} \uplus \mathcal{Q}' \setminus \mathcal{Q}''$ such that $\mathcal{Q}'' = \{(\eta', \mathtt{in}(\mathtt{c}[e_1^{s\prime}, \ldots, e_m^{s\prime}], v^s); \mathtt{P}')\}$.

  Therefore, there exist T_0 and T_1 such that $\mathrm{T}' = \mathrm{T}_0 :: (\mathbf{pc_2}, \mathrm{In}(v^s, (e_1^{s\prime}, \ldots, e_m^{s\prime}))) :: (\mathbf{pc}', ev) :: \mathrm{T}_1$ and it holds that $\mathtt{in}(\mathtt{c}[e_1^{s\prime}, \ldots, e_m^{s\prime}], v^s); \mathtt{P}' = (\!|\mathrm{In}(v^s, (e_1^{s\prime}, \ldots, e_m^{s\prime}))|\!)^\iota; (\!|ev|\!)^\iota$.

Since we have an event node $(\mathbf{pc}_2, \mathtt{In}(\mathrm{v}^s, (\mathrm{e}_1^s{}', \ldots, \mathrm{e}_m^s{}')))$, we get that there exist an intermediate state $\mathtt{I}\{s_1^s\} \in Q^{\iota\times s}$ such that $\mathtt{I}\{s_1^s\} = \{((\Pi^s, \eta^s{}', \mathbf{pc}_2), \mathtt{c}[\mathrm{e}_1^s{}', \ldots, \mathrm{e}_m^s{}'])\}$ and $\mathbf{pc}_2 \in \mathbf{L}_{\mathcal{A}_r}$.

Based on the semantics of $\mathtt{IML_{SB}}$, Fig.6.2, we find $\mathtt{I}\{s_j^s\} = ((\Pi^s, \eta^s{}'', \mathbf{pc}'), \mathtt{c}[\mathrm{e}_1^s{}', \ldots, \mathrm{e}_m^s{}'])$, $Q^{\iota\times s} \uplus Q^{\iota\times s}{}' \setminus Q^{\iota\times s}{}''$ that is in the transition relation $\mathtt{I}\{s_i^s\} \xrightarrow{o^{\iota\times s}} {}^{+}_{1,\mathrm{H}}\ \mathtt{I}\{s_j^s\}$ such that $o^{\iota\times s} = \mathtt{Out}(\mathrm{e}^s, (\mathrm{e}_1^s, \ldots, \mathrm{e}_m^s)); \ldots; \mathtt{In}(\mathrm{v}^s, (\mathrm{e}_1^s{}', \ldots, \mathrm{e}_m^s{}'))$ and $\eta^s{}'' = \eta^s{}'\ [r_0 \mapsto \mathbf{a}]$ where $(\eta^s{}', \mathbf{a}) = \mathit{mstore}(\eta^s{}', \mathit{heap}_{\mathcal{A}}, \mathit{Mem}_{\mathcal{A}}, \mathrm{v}^s, 128)$ and $\mathbf{pc}' = \mathit{ret}(\Pi^s, \eta^s{}', \mathbf{pc}_2)$.

Finally, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(\mathrm{v}^s) = \mathrm{b}'$. Thus, we can conclude that $((\Pi^s, \eta^s{}''[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathtt{c}[\mathrm{e}_1^s{}', \ldots, \mathrm{e}_m^s{}'])$, $Q^{\iota\times s} \uplus Q^{\iota\times s}{}' \setminus Q^{\iota\times s}{}'' \sim_{\mathrm{H}', (\!|.|\!)^\iota} (\eta'[\mathrm{v}^s \mapsto \mathrm{b}'], (\!|\mathrm{T}[s_i^s.\mathbf{pc}']|\!)^\iota), Q \uplus Q' \setminus Q''$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{N}} \wedge \mathbf{pc} \notin \mathbf{L}_{\mathcal{L}}$
  We show this case by a case split based on $\mathbf{P}(\mathtt{I}\{s_i^s\}.\mathbf{pc})$.

  - $\mathbf{P}(\mathtt{I}\{s_i^s\}.\mathbf{pc}) \neq \mathrm{Branch}(\phi, \mathrm{T}_1, \mathrm{T}_2)$
    Since $\mathtt{I}\{s_i^s\} \sim_{\mathrm{H}, (\!|.|\!)^\iota} s_i^\iota$, we know that $s_i^\iota.\mathtt{P} = (\!|\mathrm{T}[s_i^s.\mathbf{pc}]|\!)^\iota$. By construction of the symbolic execution tree we know that there exist T_0 and $\mathrm{T}_1{}'$ such that $\mathrm{T} = \mathrm{T}_0 :: (\mathbf{pc}, \tau) :: (\mathbf{pc}', \mathrm{ev}) :: \mathrm{T}_1{}'$.

    Therefore, based on the operational semantics of $\mathtt{IML}$ processes [7, p. 23] and translation rules in Fig.5.5, we get that $s_j^\iota = (\eta^\iota{}', (\!|\mathrm{T}[s_j^s.\mathbf{pc}']|\!)^\iota), Q$ and $s_i^\iota \longrightarrow\!\!\!\!\!\rightarrow_1 s_j^\iota$. Based on the semantics of $\mathtt{IML_{SB}}$, Fig. 6.2, we find $\mathtt{I}\{s_j^s\} = (\Pi^s{}', \eta^s{}', \mathbf{pc}'), Q^{\iota\times s}$ that is in the relation $\mathtt{I}\{s_i^s\} \longrightarrow\!\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$.

    Finally, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(\mathtt{I}\{s_j^s\}.\eta^s) = s_j^\iota.\eta^\iota$. Therefore, we can conclude that $(\Pi^s{}', \eta^s{}', \mathbf{pc}'), Q^{\iota\times s} \sim_{\mathrm{H}', (\!|.|\!)^\iota} (\eta^\iota{}', (\!|\mathrm{T}[s_i^s.\mathbf{pc}']|\!)^\iota), Q$.

  - $\mathbf{P}(\mathtt{I}\{s_i^s\}.\mathbf{pc}) = \mathrm{Branch}(\phi, \mathrm{T}_1, \mathrm{T}_2)$ case *True* (case *False*)
    Since $\mathtt{I}\{s_i^s\} \sim_{\mathrm{H}, (\!|.|\!)^\iota} s_i^\iota$, we know that $s_i^\iota.\mathtt{P} = (\!|\mathrm{T}[s_i^s.\mathbf{pc}]|\!)^\iota$. Based on the translation rules in Fig. 5.5, we get that $s_i^\iota.\mathtt{P} = \mathtt{if}\ (\!|\phi|\!)^\iota\ \mathtt{then}\ (\!|\mathrm{T}_1|\!)^\iota\ \mathtt{else}\ (\!|\mathrm{T}_2|\!)^\iota$. Also, by construction of the symbolic execution tree we know that there exist T_0, $\mathrm{T}_1{}'$, and $\mathrm{T}_2{}'$ s.t. $\mathrm{T} = \mathrm{T}_0 :: (\mathbf{pc}, \mathrm{Branch}(\phi, ((\mathbf{pc}', -) :: \mathrm{T}_1{}'), \mathrm{T}_2))$ $(\mathrm{T} = \mathrm{T}_0 :: (\mathbf{pc}, \mathrm{Branch}(\phi, \mathrm{T}_1, ((\mathbf{pc}'', -) :: \mathrm{T}_2{}'))))$.

    Therefore, based on the operational semantics of $\mathtt{IML}$ processes [7, p. 23] and translation rules in Fig. 5.5, we get that $s_i^\iota \longrightarrow\!\!\!\!\!\rightarrow_1 s_j^\iota$ and $s_j^\iota = (\eta^\iota, (\!|\mathrm{T}[s_j^s.\mathbf{pc}']|\!)^\iota)$, $Q\ \Big(s_j^\iota = (\eta^\iota, (\!|\mathrm{T}[s_j^s.\mathbf{pc}'']|\!)^\iota), Q\Big)$.

    Based on the semantics of $\mathtt{IML_{SB}}$, Fig. 6.2, we find $s_j^s = (\Pi^s, \eta^s, \mathbf{pc}')$ $\Big(s_j^s = (\Pi^s, \eta^s, \mathbf{pc}'')\Big)$ that is in the transition relation $\mathtt{I}\{s_i^s\} \longrightarrow\!\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$. There-

fore, we can conclude that $(\Pi^s, \eta^s, \mathbf{pc}'), \mathcal{Q}^{\iota\times s} \sim_{H',(\!|.|\!)^\iota} (\eta^\iota, (\!|\mathtt{T}[s_i^s.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$ $\Big((\Pi^s, \eta^s, \mathbf{pc}''), \mathcal{Q}^{\iota\times s} \sim_{H',(\!|.|\!)^\iota} (\eta^\iota, (\!|\mathtt{T}[s_i^s.\mathbf{pc}'']|\!)^\iota), \mathcal{Q}\Big)$.

- $\mathbf{P}(\mathtt{I}\{s_i^s\}.\mathbf{pc}) \in \mathbf{L}_{\mathcal{L}}$
  Since $\mathtt{I}\{s_i^s\} \sim_{H,(\!|.|\!)^\iota} s_i^\iota$, we know that $s_i^\iota.\mathtt{P} = (\!|\mathtt{T}[s_i^s.\mathbf{pc}]|\!)^\iota$. Based on the translation rules in Fig. 5.5, we get that $s_i^\iota.\mathtt{P} = !^{t^s \leq \mathtt{m}}\ \mathit{LoopProc}(s_i^s.\mathbf{pc})$. Also, by construction of the symbolic execution tree we know that there exist $\mathtt{T}_0$, and $\mathtt{T}_1$ such that $\mathtt{T} = \mathtt{T}_0 :: (\mathbf{pc}, \mathtt{loop}(t^s)) :: (\mathbf{pc}', \mathtt{ev}) :: \mathtt{T}_1$ such that $\mathbf{pc}' = \mathit{exit}(\mathbf{pc})$.

  Therefore, based on the operational semantics of IML processes [7, p. 23] and translation rules in Fig. 5.5, we get $s_j^\iota = (\eta', (\!|\mathtt{T}[s_j^s.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$ and $s_i^\iota \twoheadrightarrow_1 s_j^\iota$ such that $\eta'[t^s] = \mathtt{b}$. Based on the semantics of $\mathtt{IML}_{SB}$, Fig. 6.2 and the computed loop summary, we find $\mathtt{I}\{s_j^s\} = (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), \mathcal{Q}^{\iota\times s}$ that is in the transition relation $\mathtt{I}\{s_i^s\} \xrightarrow{o^{\iota\times s}} \!\!\!\!\twoheadrightarrow_{1,H} \mathtt{I}\{s_j^s\}$, $o^{\iota\times s} = \mathtt{loop}(t^s)$, and $\mathbf{pc}' = \mathit{exit}(\mathbf{pc})$.

  Finally, for every IML variable $\mathtt{x}$ and symbolic BIR variable x^s that $\mathtt{I}\{s_j^s\}.\eta^s[x^s] \neq \mathtt{I}\{s_i^s\}.\eta^s[x^s]$ and $s_j^\iota.\eta^\iota[\mathtt{x}] \neq s_i^\iota.\eta^\iota[\mathtt{x}]$, we choose H' by extending H according to the executed operation such that $H'(\mathtt{I}\{s_j^s\}.\eta^s[x^s]) = s_j^\iota.\eta^\iota[\mathtt{x}]$. Moreover, for the symbolic counter t^s which represents the number of iterations of the loop in symbolic execution and the number of the process replication $\mathtt{b}$, we have $H'(t^s) = \mathtt{b}$. Therefore, we can conclude that $(\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), \mathcal{Q}^{\iota\times s} \sim_{H',(\!|.|\!)^\iota} (\eta', (\!|\mathtt{T}[s_j^s.\mathbf{pc}']|\!)^\iota), \mathcal{Q}$.

□

We then show the translation's soundness by extending the simulation relation to execution traces, i.e., $\sim_{H,(\!|.|\!)^\iota,k} \subseteq \mathcal{R}^{\iota\times s} \times \mathcal{R}^\iota$, w.r.t an upper bound $k \in \mathbb{N}$ on the number of RNG steps of the execution $rng : \mathcal{R} \to \mathbb{N}$. The bound k is needed because of BIR's finite memory model. That is, $R^{\iota\times s} \sim_{H,(\!|.|\!)^\iota,k} \mathtt{R}^\iota$ holds, iff, $rng(R^{\iota\times s}) \leq k$ and for all $\mathtt{I}\{s^s\}$ and $o^{\iota\times s} \in R^{\iota\times s}$ there exist $s^\iota, \mathtt{o}^\iota \in \mathtt{R}^\iota$, and H s.t. $\mathtt{I}\{s^s\} \sim_{H,(\!|.|\!)^\iota} s^\iota$ and $o^{\iota\times s} =_H \mathtt{o}^\iota$.

Finally, we show that executions of the mixed IML and symbolic execution and IML preserve the simulation relation. Note that, in the following, we assume a single BIR program that implements different protocol participants with distinct sets of program counters. The results can be extended to multiple BIR programs, as presented in Sec. 6.1.4.

Theorem 6.3 ($\mathtt{IML}_{SB}$-IML Trace Inclusion). *Let* $\mathbf{P}$ *be a* BIR *program,* $\mathtt{I}$ *be an* IML *process, and* $k \in \mathbb{N}$ *is any upper bound on the number of RNG steps, then, for all mixed* IML *and symbolic execution traces* $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(\mathtt{I}\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm}_k^s, \mathbf{r_k} \mapsto 0])$ *s.t.* $rng(R^{\iota\times s}) \leq k$, *there are an* IML *trace* $\mathtt{R}^\iota \in \mathcal{R}^\iota(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\})$ *and an* H *s.t.* $R^{\iota\times s} \sim_{H,(\!|.|\!)^\iota,k} \mathtt{R}^\iota$.

Proof. The goal is to show that for all $\mathtt{IML}_{\mathrm{SB}}$ traces, there is an equivalent $\mathtt{IML}$ trace that are in the simulation relation through the interpretation H. We prove the theorem by induction on the length of the execution traces:

- **Base case**. Follows from Theorem 6.1.
- **Inductive case**. Follows from Theorem 6.2.

□

Theorem 6.3 is the first step to relating the properties we verify for the $\mathtt{IML}$ model to the actual binary of the protocol. We showed that the $\mathtt{IML}$ model resulting from translation covers all behaviors in the $\mathtt{IML}_{\mathrm{SB}}$ semantics. Recall that we have to talk about behavioral properties in the mixed semantics (as opposed to the pure **BIR** semantics) as protocol properties typically concern more than one party. Next, we show that these symbolic behaviors cover all concrete behaviors.

6.1.2 Symbolic Execution

To ensure that the extracted $\mathtt{IML}$ model preserves the semantics of the protocol's binary, we have to prove further that our symbolic execution is behaviorally equivalent to the transpiled **BIR** code. To show this, we construct a mixed $\mathtt{IML}$-**BIR** execution semantics, hereafter $\mathtt{IML}_{\mathbf{B}}$, that allows the **BIR** program to communicate with the same $\mathtt{IML}$ attacker at the $\mathtt{IML}_{\mathrm{SB}}$ level. Fig.6.3 presents the $\mathtt{IML}$-**BIR** mixed execution semantics and the rules are similar and have the same meaning as those defined for $\mathtt{IML}_{\mathrm{SB}}$.

Our proof strategy to show the behavioral equivalence of $\mathtt{IML}_{\mathbf{B}}$ and $\mathtt{IML}_{\mathrm{SB}}$ is similar to our technique to prove the soundness of the $\mathtt{IML}$ translation. That is, we first show the state/event equivalence between the two abstractions and then use this to prove the trace inclusion of $\mathtt{IML}_{\mathbf{B}}$ in $\mathtt{IML}_{\mathrm{SB}}$.

We show the state/event equivalence by extending the simulation relation of Property 1 to a relation $\sim_{\mathrm{H}} \subseteq S^{\iota^b} \times S^{\iota^s}$ between $\mathtt{IML}_{\mathbf{B}}$ and $\mathtt{IML}_{\mathrm{SB}}$. The relation $\mathtt{I}\{s_{\mathrm{i}}^b\} \sim_{\mathrm{H}} \mathtt{I}\{s_{\mathrm{i}}^s\}$ checks that $\mathtt{I}\{s_{\mathrm{i}}^s\}.\mathbf{pc} = \mathtt{I}\{s_{\mathrm{i}}^b\}.\mathbf{pc}$, and for all $\mathbf{x} \in dom(\mathtt{I}\{s_{\mathrm{i}}^b\}.\eta^b)$ there exist $\mathrm{x}^s \in dom(\mathtt{I}\{s_{\mathrm{i}}^s\}.\eta^s)$ and an interpretation H s.t. $\mathrm{H}(\mathtt{I}\{s_{\mathrm{i}}^s\}.\eta^s[\mathrm{x}^s]) = \mathtt{I}\{s_{\mathrm{i}}^b\}.\eta^b[\mathbf{x}]$. The first step in showing this simulation relation between the two layers is to prove that the initial states are in the relation using Theorem 6.4:

Lemma 6.4 ($\mathtt{IML}_{\mathbf{B}}$-$\mathtt{IML}_{\mathrm{SB}}$ Initial State Equivalence)**.** *For a* **BIR** *program* $\mathbf{P}$*, an* $\mathtt{IML}$ *process* $\mathtt{I}$ *and any upper bound* $k \in \mathbb{N}$ *on the number of RNG steps, let* $\mathtt{I}\{s_0^b\} = (\eta_0^b[\mathbf{RM} \mapsto \mathtt{rm}_{\mathrm{k}}, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$ *be an initial* **BIR** *state in* $\mathtt{IML}_{\mathbf{B}}$ *and* $\mathtt{I}\{s_0^s\} = (True, \eta_0^s[\mathbf{RM} \mapsto \mathrm{rm}_{\mathrm{k}}^s, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$ *be the corresponding initial state in* $\mathtt{IML}_{\mathrm{SB}}$*. Then,* $\mathtt{I}\{s_0^b\} \sim_{\mathrm{H}} \mathtt{I}\{s_0^s\}$ *for all* H*.*

$$\frac{\mathbf{pc} \in \mathbf{L}_{\mathcal{N}} \uplus \xi_C \uplus \xi_{\mathcal{R}} \uplus \xi_{\mathcal{E}} \qquad (\eta^b, \mathbf{pc}) \xrightarrow{\mathrm{o}^b} (\eta^{b\prime}, \mathbf{pc}')}{\mathbf{P} \vdash (\eta^b, \mathbf{pc}), Q^{\iota \times b} \xrightarrow{\mathrm{o}^b}\!\!\!\!\!\rightarrow_1 (\eta^{b\prime}, \mathbf{pc}'), Q^{\iota \times b}}\ normal_{\mathrm{IB}}$$

$$\frac{\forall j \leq m : [\![\mathrm{y_j}]\!]_{\eta^\iota} = \mathrm{b_j} \neq \bot \quad \forall j \leq m : (\eta^b_{j+1}, \mathbf{a_j}) = mstore(\eta^b_j, heap, Mem, \mathrm{b_j}, 128)}{\mathbf{P} \vdash (\eta^\iota, \mathtt{run}(\mathbf{pc}, (\mathrm{y_1}, ..., \mathrm{y_m}))), Q^{\iota \times b} \xrightarrow{}\!\!\!\!\!\rightarrow_1 (\eta^b_{m+1}[r_0 \mapsto \mathbf{a_1}, ..., r_{m-1} \mapsto \mathbf{a_m}], \mathbf{pc}), Q^{\iota \times b}}\ run_{\mathrm{IB}}$$

$$\frac{\begin{array}{c}\mathbf{pc} \in \xi_{\mathcal{A}_s} \quad \mathbf{pc}' = ret(\eta^b, \mathbf{pc}) \quad Q^{\iota \times b\prime} = \{(\eta^b, \mathbf{pc}')\} \quad \mathbf{a} = \eta^b[r_0] \quad \mathrm{b} = mload(\eta^b, \mathbf{a}) \\ \forall j \leq m : \eta^b[\mathbf{e_j}] = \mathrm{b_j} \neq \bot \quad \mathrm{b}' = truncate(\mathrm{b}, maxlen(\mathtt{c})) \\ \exists!(\eta^\iota, \mathtt{Q}) \in Q^{\iota \times b} : (\mathtt{Q} = \mathtt{in}(\mathtt{c}[\mathrm{e_1}', ..., \mathrm{e_m}'], \mathrm{x}); \mathtt{P} \wedge \forall j \leq m : [\![\mathrm{e_j}']\!]_{\eta^\iota} = \mathrm{b_j} \neq \bot)\end{array}}{\mathbf{P} \vdash ((\eta^b, \mathbf{pc}), \mathtt{c}[\mathbf{e_1}, ..., \mathbf{e_m}]), Q^{\iota \times b} \xrightarrow{\mathrm{Out}(\mathrm{b}, (\mathbf{e_1}, ..., \mathbf{e_m}))}\!\!\!\!\!\rightarrow_1 (\eta^\iota[\mathrm{x} \mapsto \mathrm{b}'], \mathtt{P}), Q^{\iota \times b} \uplus Q^{\iota \times b\prime} \setminus \{(\eta^\iota, \mathtt{Q})\}}\ BtoI$$

$$\frac{\begin{array}{c}[\![\mathrm{e}]\!]_{\eta^\iota} = \mathrm{b} \neq \bot \quad \mathrm{b}' = truncate(\mathrm{b}, maxlen(\mathtt{c})) \quad \forall j \leq m : [\![\mathrm{e_j}]\!]_{\eta^\iota} = \mathrm{b_j} \neq \bot \quad Q^{\iota \times b\prime} = reduce(\{(\eta^\iota, \mathtt{Q})\}) \\ \exists!((\eta^b, \mathbf{pc}), \mathtt{c}[\mathbf{e_1}', ..., \mathbf{e_m}']) \in Q^{\iota \times b} : (\mathbf{pc} \in \xi_{\mathcal{A}_r} \wedge \forall j \leq m : \eta^b[\mathbf{e_j}'] = \mathrm{b_j} \neq \bot) \quad \mathbf{pc}' = ret(\eta^b, \mathbf{pc}) \\ (\eta^{b\prime}, \mathbf{a}) = mstore(\eta^b, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, \mathrm{b}', 128) \quad \eta^{b\prime\prime} = \eta^{b\prime}[r_0 \mapsto \mathbf{a}] \quad Q^{\iota \times b\prime\prime} = \{((\eta^b, \mathbf{pc}), \mathtt{c}[\mathbf{e_1}', ..., \mathbf{e_m}'])\}\end{array}}{\mathbf{P} \vdash (\eta^\iota, \mathtt{out}(\mathtt{c}[\mathrm{e_1}, ..., \mathrm{e_m}], \mathrm{e}); \mathtt{Q}), Q^{\iota \times b} \xrightarrow{\mathrm{In}(\mathrm{b}', (\mathbf{e_1}', ..., \mathbf{e_m}'))}\!\!\!\!\!\rightarrow_1 ((\eta^{b\prime\prime}, \mathbf{pc}'), \mathtt{c}[\mathbf{e_m}', ..., \mathbf{e_m}']), Q^{\iota \times b} \uplus Q^{\iota \times b\prime} \setminus Q^{\iota \times b\prime\prime}}\ ItoB$$

Figure 6.3: The mixed semantics of **BIR** and IML shown by $\mathtt{IML_B}$.

Proof. We choose H such that for $0 < i \leq k$, $\mathrm{H}(\mathrm{I}\{s_0^s\}.\eta^s.\mathbf{RM}[\mathrm{x_i^s}]) = \mathrm{I}\{s_0^b\}.\eta^b.\mathbf{RM}[\mathbf{x_i}]$. Finally, we can conclude that $(\eta_0^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0], \mathbf{pc_0}) \sim_{\mathrm{H}} (True, \eta_0^s[\mathbf{RM} \mapsto \mathrm{rm_k^s}, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$. □

We then prove that the single-step transitions of $\mathtt{IML_B}$ and $\mathtt{IML_{SB}}$ preserve the simulation relation using Theorem 6.5.

Lemma 6.5 ($\mathtt{IML_B}$-$\mathtt{IML_{SB}}$ State/Event Equivalence). *Let* **P** *be a* **BIR** *program and* I *be an* IML *process, then, for all* $\mathrm{I}\{s_i^s\}, \mathrm{I}\{s_i^b\}, \mathrm{I}\{s_j^b\}$ *and* H *s.t.* $\mathrm{I}\{s_i^b\} \sim_{\mathrm{H}} \mathrm{I}\{s_i^s\}$ *and* $\mathrm{I}\{s_i^b\} \xrightarrow{o^{\iota \times b}}\!\!\!\!\!\rightarrow_{\mathrm{p}}^{+} \mathrm{I}\{s_j^b\}$, *there exist an* H′ *and* $\mathrm{I}\{s_j^s\}$ *s.t.* $\mathrm{H} \subseteq \mathrm{H}'$, $\mathrm{I}\{s_i^s\} \xrightarrow{o^{\iota \times s}}\!\!\!\!\!\rightarrow_{\mathrm{p},\mathrm{H}'}^{+} \mathrm{I}\{s_j^s\}$, $\mathrm{I}\{s_j^b\} \sim_{\mathrm{H}'} \mathrm{I}\{s_j^s\}$ *and* $o^{\iota \times b} =_{\mathrm{H}'} o^{\iota \times s}$.

Proof of Theorem 6.5 is done by a case split on the type of label sets in the **BIR** program **P**.

Proof. We prove the statement by a case split based on the type of the program counter $\mathrm{I}\{s_i^b\}.\mathbf{pc}$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{E}}$
  Based on the mixed semantics of IML and **BIR**, Fig. 6.3, we get that $\mathrm{I}\{s_j^b\} = (\eta^{b\prime}, \mathbf{pc}'), Q^{\iota \times b}$ and $\mathrm{I}\{s_i^b\} \xrightarrow{\mathrm{o}^{\iota \times b}}\!\!\!\!\!\rightarrow_1 \mathrm{I}\{s_j^b\}$. Based on the definition of event functions in Sec. 5.1.4, we have $\mathbf{pc}' = ret(\eta^b, \mathbf{pc})$, $o^{\iota \times b} = \mathbf{Ev}(\mathrm{b_1}, \ldots, \mathrm{b_m})$ and $\eta^{b\prime} = \eta^b$. Since

$\mathtt{I}\{s_i^b\} \sim_{\mathrm{H}} \mathtt{I}\{s_i^s\}$, we get that $\mathtt{I}\{s_i^s\}.\mathbf{pc} \in \mathbf{L}_{\mathcal{E}}$. Based on the semantics of $\mathtt{IML}_{\mathrm{SB}}$, Fig.6.2, we find $\mathtt{I}\{s_j^s\} = (\Pi^s, \eta^s, \mathbf{pc}'), Q^{\iota \times s}$ that is in the transition relation $\mathtt{I}\{s_i^s\} \xrightarrow{o^{\iota \times s}} \!\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$ such that $o^{\iota \times s} = \mathrm{Ev}(d_1^s, \ldots, d_m^s)$.

Therefore, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(d_j^s) = b_j$ for $0 < j \leq m$. Moreover, since the environment of both s_j^b and s_j^s are the same as s_i^b and s_i^s, respectively, we get that $\mathtt{I}\{s_j^s\}.\eta^s =_{\mathrm{H}'} \mathtt{I}\{s_j^b\}.\eta^b$. Finally, we can conclude that $(\eta^b, \mathbf{pc}'), Q^{\iota \times b} \sim_{\mathrm{H}'} (\Pi^s, \eta^s, \mathbf{pc}'), Q^{\iota \times s}$ and $\mathbf{Ev}(b_1, \ldots, b_m) =_{\mathrm{H}'} \mathrm{Ev}(d_1^s, \ldots, d_m^s)$.

- $\mathbf{pc} \in \mathbf{L}_C$

  Based on the mixed semantics of $\mathtt{IML}$ and **BIR**, Fig. 6.3, we get that $\mathtt{I}\{s_j^b\} = (\eta^{b\prime\prime}, \mathbf{pc}'), Q^{\iota \times b}$ and $\mathtt{I}\{s_i^b\} \xrightarrow{o^{\iota \times b}} \!\!\!\!\!\rightarrow_1 \mathtt{I}\{s_j^b\}$. Based on the definition of cryptographic calls in Sec. 5.1.3, we have $o^{\iota \times b} = \mathbf{Cr}(v)$, $\mathbf{pc}' = ret(\eta^b, \mathbf{pc})$ and $\eta^{b\prime\prime} = \eta^{b\prime}[r_0 \mapsto \mathbf{a}]$ where $(\eta^{b\prime}, \mathbf{a}) = mstore(\eta^b, heap_{\mathrm{Op}}, Mem_{\mathrm{Op}}, v, 128)$.

  Since $\mathtt{I}\{s_i^b\} \sim_{\mathrm{H}} \mathtt{I}\{s_i^s\}$, we get that $\mathtt{I}\{s_i^s\}.\mathbf{pc} \in \mathbf{L}_C$. Based on the semantics of $\mathtt{IML}_{\mathrm{SB}}$, Fig. 6.2, we find $\mathtt{I}\{s_j^s\} = (\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}'), Q^{\iota \times s}$ that is in the transition relation $\mathtt{I}\{s_i^s\} \xrightarrow{o^{\iota \times s}} \!\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$ such that $o^{\iota \times s} = \mathrm{Cr}(v^s)$ and $\eta^{s\prime\prime} = \eta^{s\prime}[r_0 \mapsto \mathbf{a}]$ where $(\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap_{\mathrm{Op}}, Mem_{\mathrm{Op}}, v^s, 128)$.

  Therefore, we choose H' by extending H according to the executed operation and such that $\mathrm{H}'(v^s) = v$. Hence, we get that $\mathtt{I}\{s_j^s\}.\eta^s =_{\mathrm{H}'} \mathtt{I}\{s_j^b\}.\eta^b$. Finally, we can conclude that $(\eta^{b\prime\prime}, \mathbf{pc}'), Q^{\iota \times b} \sim_{\mathrm{H}'} (\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}'), Q^{\iota \times s}$ and $\mathbf{Cr}(v) =_{\mathrm{H}'} \mathrm{Cr}(v^s)$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{R}}$

  Based on the mixed semantics of $\mathtt{IML}$ and **BIR**, Fig. 6.3, we get that $\mathtt{I}\{s_j^b\} = (\eta^{b\prime\prime}, \mathbf{pc}'), Q^{\iota \times b}$ and $\mathtt{I}\{s_i^b\} \xrightarrow{o^{\iota \times b}} \!\!\!\!\!\rightarrow_1 \mathtt{I}\{s_j^b\}$. Based on the definition of RNG in Sec. 5.1.1, we have $o^{\iota \times b} = \mathbf{Fr}(x_d)$, $d = \left\lfloor \frac{\eta^b[r_k]}{l} \right\rfloor + 1$, $\eta^{b\prime\prime} = \eta^{b\prime}[r_0 \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta^b[\mathbf{r_k}] + l]$ where $(\eta^{b\prime}, \mathbf{a}) = mstore(\eta^b, heap, Mem, x_d, 128)$, $x_d = \mathfrak{R}(\eta^b, n)$ and $\mathbf{pc}' = ret(\eta^b, \mathbf{pc})$.

  Since $\mathtt{I}\{s_i^b\} \sim_{\mathrm{H}} \mathtt{I}\{s_i^s\}$, we get that $\mathtt{I}\{s_i^s\}.\mathbf{pc} \in \mathbf{L}_{\mathcal{R}}$. Based on the semantics of $\mathtt{IML}_{\mathrm{SB}}$, Fig. 6.2, we find $\mathtt{I}\{s_j^s\} = (\Pi^s, \eta^{s\prime\prime}, \mathbf{pc}'), Q^{\iota \times s}$ that is in the relation $\mathtt{I}\{s_i^s\} \xrightarrow{o^{\iota \times s}} \!\!\!\!\!\rightarrow_{1,\mathrm{H}} \mathtt{I}\{s_j^s\}$ such that $o^{\iota \times s} = \mathrm{Fr}(x^s_{d'})$, $d' = \left\lfloor \frac{\eta^s[r_k]}{l} \right\rfloor + 1$ and $\eta^{s\prime\prime} = \eta^{s\prime}[r_0 \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta^s[\mathbf{r_k}] + l]$ where $(\eta^{s\prime}, \mathbf{a}) = mstore(\eta^s, heap, Mem, x^s_{d'}, 128)$ and $x^s_{d'} = \mathfrak{R}^s(\eta^s, n)$.

  Since $\mathtt{I}\{s_i^b\} \sim_{\mathrm{H}} \mathtt{I}\{s_i^s\}$ and $\mathtt{I}\{s_i^s\}.\eta^s =_{\mathrm{H}} \mathtt{I}\{s_i^b\}.\eta^b$, we get that $d' = d$. Finally, we choose H' by extending H according to the executed operation and such that

$\mathtt{H}'(\mathtt{x}^s_\mathtt{d}) = \mathtt{x}_\mathtt{d}$. Hence, we get that $\mathtt{I}\{s^s_\mathtt{j}\}.\eta^s =_{\mathtt{H}'} \mathtt{I}\{s^b_\mathtt{j}\}.\eta^b$. Therefore, we can conclude that $(\eta^b{''}, \mathbf{pc}'), Q^{\iota\times_b} \sim_{\mathtt{H}'} (\Pi^s, \eta^s{''}, \mathbf{pc}'), Q^{\iota\times_s}$ and $\mathbf{Fr}(\mathtt{x}_\mathtt{d}) =_{\mathtt{H}'} \mathtt{Fr}(\mathtt{x}^s_\mathtt{d})$.

- $\mathbf{pc}_1 \in \mathbf{L}_{\mathcal{A}_\mathtt{s}} \wedge \mathbf{pc}_2 \in \mathbf{L}_{\mathcal{A}_\mathtt{r}}$

  In this case, $\mathtt{I}\{s^b_\mathtt{i}\} \xrightarrow{o^{\iota\times_b}}\!\!\!\!\!\rightarrow^+_\mathtt{p} \mathtt{I}\{s^b_\mathtt{j}\}$ amounts to 3 sub-transitions as follows. First, using the **B***to*I rule in Fig.6.3, we have the transition relation $\mathtt{I}\{s^b_\mathtt{i}\} \xrightarrow{\mathbf{Out}(\mathtt{b},(\mathbf{e}_1,\ldots,\mathbf{e}_\mathtt{m}))}\!\!\!\!\!\rightarrow_1 \mathtt{I}\{s^\iota_\mathtt{x}\}$ such that $\mathtt{I}\{s^b_\mathtt{i}\}.\mathbf{pc}_1 \in \mathbf{L}_{\mathcal{A}_\mathtt{s}}$.

  Then, based on the operational semantics of IML output processes [7, p. 23], the transition $\mathtt{I}\{s^\iota_\mathtt{x}\} \rightarrow\!\!\!\!\!\rightarrow^+_\mathtt{p} \mathtt{I}\{s^\iota_\mathtt{y}\}$ takes place. Finally, we have $\mathtt{I}\{s^\iota_\mathtt{y}\} \xrightarrow{\mathbf{In}(\mathtt{b}',(\mathbf{e}'_1,\ldots,\mathbf{e}'_\mathtt{m}))}\!\!\!\!\!\rightarrow_1 \mathtt{I}\{s^b_\mathtt{j}\}$ using I*to*B rule in Fig.6.3 such that there exist a **BIR** state $((\eta^b, \mathbf{pc}_2), \mathtt{c}[\mathbf{e}'_1, \ldots, \mathbf{e}'_\mathtt{m}])$ in the multiset of executing states $\mathtt{I}\{s^\iota_\mathtt{y}\}.Q^{\iota\times_b}$ such that $\mathbf{pc}_2 \in \xi_{\mathcal{A}_\mathtt{r}}$.

  Since $\mathtt{I}\{s^b_\mathtt{i}\} \sim_\mathtt{H} \mathtt{I}\{s^s_\mathtt{i}\}$, we need to find a mixed symbolic and IML state $\mathtt{I}\{s^s_\mathtt{j}\}$ such that $\mathtt{I}\{s^b_\mathtt{j}\} \sim_\mathtt{H} \mathtt{I}\{s^s_\mathtt{j}\}$ and it is reachable from intermediate states $\mathtt{I}\{s^\iota_\mathtt{x}\}$ and $\mathtt{I}\{s^\iota_\mathtt{y}\}$ using SB*to*I and I*to*SB rules in Fig. 6.2, respectively.

  Therefore, since $\mathbf{pc}_1 \in \mathbf{L}_{\mathcal{A}_\mathtt{s}}$, based on the mixed semantics of IML and **BIR**, Fig.6.3, we get that there exist an IML state $\mathtt{I}\{s^\iota_\mathtt{x}\}$ such that $\mathtt{I}\{s^b_\mathtt{i}\} \xrightarrow{\mathbf{Out}(\mathtt{b},(\mathbf{e}_1,\ldots,\mathbf{e}_\mathtt{m}))}\!\!\!\!\!\rightarrow_1 \mathtt{I}\{s^\iota_\mathtt{x}\}$ and $\mathtt{I}\{s^\iota_\mathtt{x}\} = (\eta^\iota[\mathtt{x} \mapsto \mathtt{b}'], \mathtt{P}), Q^{\iota\times_b} \setminus Q^{\iota\times_b\prime}$ such that $\mathtt{b}' = \mathit{truncate}(\mathtt{b}, \mathit{maxlen}(\mathtt{c}))$.

  Since $\mathtt{I}\{s^b_\mathtt{i}\} \sim_\mathtt{H} \mathtt{I}\{s^s_\mathtt{i}\}$, we get that $\mathtt{I}\{s^s_\mathtt{i}\}.\mathbf{pc}_1 \in \mathbf{L}_{\mathcal{A}_\mathtt{s}}$. Based on the semantics of $\mathtt{IML}_\mathtt{SB}$, Fig.6.2, we find $\mathtt{I}\{s^\iota_\mathtt{x}\} = (\eta^\iota[\mathtt{x} \mapsto \mathtt{b}'], \mathtt{P}), Q^{\iota\times_s} \setminus Q^{\iota\times_s\prime}$ that is in the transition relation $\mathtt{I}\{s^s_\mathtt{i}\} \xrightarrow{o^{\iota\times_s}}\!\!\!\!\!\rightarrow_{1,\mathtt{H}} \mathtt{I}\{s^\iota_\mathtt{x}\}$ such that $o^{\iota\times_s} = \mathtt{Out}(\mathtt{e}^s, (\mathtt{e}^s_1, \ldots, \mathtt{e}^s_\mathtt{m}))$.

  Since $\mathbf{pc}_2 \in \mathbf{L}_{\mathcal{A}_\mathtt{r}}$, based on the mixed semantics of IML and **BIR**, Fig. 6.3, we get that there exist an IML state $s^\iota_\mathtt{y}$ such that $\mathtt{I}\{s^\iota_\mathtt{y}\} \xrightarrow{\mathbf{In}(\mathtt{b}',(\mathbf{e}'_1,\ldots,\mathbf{e}'_\mathtt{m}))}\!\!\!\!\!\rightarrow_1 \mathtt{I}\{s^b_\mathtt{j}\}$ and $\mathtt{I}\{s^b_\mathtt{j}\} = ((\eta^b{'}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathtt{c}[\mathbf{e}'_1, \ldots, \mathbf{e}'_\mathtt{m}]), Q^{\iota\times_b} \uplus Q^{\iota\times_b\prime} \setminus Q^{\iota\times_b\prime\prime}$ such that $(\eta^b{'}, \mathbf{a}) = \mathit{mstore}(\eta^b, \mathit{heap}_\mathcal{A}, \mathit{Mem}_\mathcal{A}, \mathtt{b}', 128)$ and $\mathbf{pc}' = \mathit{ret}(\eta^b, \mathbf{pc}_2)$.

  Since $\mathtt{I}\{s^b_\mathtt{i}\} \sim_\mathtt{H} \mathtt{I}\{s^s_\mathtt{i}\}$, we get that $\mathtt{I}\{s^s_\mathtt{i}\}.\mathbf{pc}_2 \in \mathbf{L}_{\mathcal{A}_\mathtt{r}}$. Based on the semantics of $\mathtt{IML}_\mathtt{SB}$, Fig.6.2, we find $\mathtt{I}\{s^s_\mathtt{j}\} = ((\Pi^s, \eta^s{'}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathtt{c}[\mathtt{e}^s_1{'}, \ldots, \mathtt{e}^s_\mathtt{m}{'}]), Q^{\iota\times_s} \uplus Q^{\iota\times_s\prime} \setminus Q^{\iota\times_s\prime\prime}$ that is in the relation $\mathtt{I}\{s^\iota_\mathtt{y}\} \xrightarrow{o^{\iota\times_s}}\!\!\!\!\!\rightarrow_{1,\mathtt{H}} \mathtt{I}\{s^s_\mathtt{j}\}$ such that $\mathbf{pc}' = \mathit{ret}(\Pi^s, \eta^s, \mathbf{pc}_2)$, $o^{\iota\times_s} = \mathtt{In}(\mathtt{e}^s{'}, (\mathtt{e}^s_1{'}, \ldots, \mathtt{e}^s_\mathtt{m}{'}))$ and $(\eta^s{'}, \mathbf{a}) = \mathit{mstore}(\eta^s, \mathit{heap}_\mathcal{A}, \mathit{Mem}_\mathcal{A}, \mathtt{e}^s{'}, 128)$.

  Moreover, we choose $\mathtt{H}'$ by extending $\mathtt{H}$ according to the executed operation and such that $\mathtt{H}'(\mathtt{e}^s) = \mathtt{b}$, $\mathtt{H}'(\mathtt{e}^s{'}) = \mathtt{b}'$, $\mathtt{H}'(\mathtt{e}^s_\mathtt{j}) = \mathbf{e}_\mathtt{j}$ and $\mathtt{H}'(\mathtt{e}^s_\mathtt{j}{'}) = \mathbf{e}_\mathtt{j}{'}$ for $1 \leq j \leq m$. Hence, we get that $\mathtt{I}\{s^s_\mathtt{j}\}.\eta^s =_{\mathtt{H}'} \mathtt{I}\{s^b_\mathtt{j}\}.\eta^b$. Therefore, we can conclude that $((\eta^b{'}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathtt{c}[\mathbf{e}'_1, \ldots, \mathbf{e}'_\mathtt{m}]), Q^{\iota\times_b} \uplus Q^{\iota\times_b\prime} \setminus Q^{\iota\times_b\prime\prime} \sim_{\mathtt{H}'} ((\Pi^s, \eta^s{'}[r_0 \mapsto \mathbf{a}], \mathbf{pc}'), \mathtt{c}[\mathtt{e}^s_1{'}, \ldots, \mathtt{e}^s_\mathtt{m}{'}]), Q^{\iota\times_s} \uplus Q^{\iota\times_s\prime} \setminus Q^{\iota\times_s\prime\prime}$, $\mathbf{Out}(\mathtt{b}, (\mathbf{e}_1, \ldots, \mathbf{e}_\mathtt{m})) =_{\mathtt{H}'} \mathtt{Out}(\mathtt{e}^s, (\mathtt{e}^s_1, \ldots, \mathtt{e}^s_\mathtt{m}))$, and $\mathbf{In}(\mathtt{b}', (\mathbf{e}'_1, \ldots, \mathbf{e}'_\mathtt{m})) =_{\mathtt{H}'} \mathtt{In}(\mathtt{e}^s{'}, (\mathtt{e}^s_1{'}, \ldots, \mathtt{e}^s_\mathtt{m}{'}))$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{N}} \wedge \mathbf{pc} \notin \mathbf{L}_{\mathcal{L}}$
  Based on the mixed semantics of IML and BIR, Fig. 6.3, we get that $\mathtt{I}\{s_j^b\} = (\eta^{b\prime}, \mathbf{pc}'), Q^{\iota\times b}$ and $\mathtt{I}\{s_i^b\} \xrightarrow{o^{\iota\times b}}\!\!\!\!\!\rightarrow_1 \mathtt{I}\{s_j^b\}$. Since $\mathtt{I}\{s_i^b\} \sim_H \mathtt{I}\{s_i^s\}$, we get that $\mathtt{I}\{s_i^s\}.\mathbf{pc} \in \mathbf{L}_{\mathcal{N}}$. Based on the semantics of $\mathtt{IML}_{\mathtt{SB}}$, Fig. 6.2, we find $\mathtt{I}\{s_j^s\} = (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), Q^{\iota\times s}$ that is in the transition relation $\mathtt{I}\{s_i^s\} \rightarrow\!\!\!\!\!\rightarrow_{1,H} \mathtt{I}\{s_j^s\}$.

  Based on Lindner's work [116] (see Property 1), there exist an interpretation $H' \supseteq H$ such that $\mathtt{I}\{s_j^s\}.\eta^s =_{H'} \mathtt{I}\{s_j^b\}.\eta^b$. Therefore, we can conclude that $(\eta^{b\prime}, \mathbf{pc}'), Q^{\iota\times b} \sim_{H'} (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), Q^{\iota\times s}$.

- $\mathbf{pc} \in \mathbf{L}_{\mathcal{L}}$
  Based on the mixed semantics of IML and BIR, Fig. 6.3, we get that $\mathtt{I}\{s_j^b\} = (\eta^{b\prime}, \mathbf{pc}'), Q^{\iota\times b}$ and $\mathtt{I}\{s_i^b\} \xrightarrow{\mathbf{loop(t)}}\!\!\!\!\!\rightarrow_1^+ \mathtt{I}\{s_j^b\}$ and $\mathbf{pc}' = \mathit{exit}(\mathbf{pc})$. Since $\mathtt{I}\{s_i^b\} \sim_H \mathtt{I}\{s_i^s\}$, we get that $\mathtt{I}\{s_i^s\}.\mathbf{pc} \in \mathbf{L}_{\mathcal{L}}$. Based on the semantics of $\mathtt{IML}_{\mathtt{SB}}$, Fig. 6.2 and the computed loop summary, we find $\mathtt{I}\{s_j^s\} = (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), Q^{\iota\times s}$ that is in the transition relation $\mathtt{I}\{s_i^s\} \xrightarrow{\mathtt{loop(t^s)}}\!\!\!\!\!\rightarrow_{1,H}^+ \mathtt{I}\{s_j^s\}$ such that $\mathbf{pc}' = \mathit{exit}(\mathbf{pc})$. Therefore, we get that both loops in $\mathtt{I}\{s_j^b\}$ and $\mathtt{I}\{s_j^s\}$ terminate at the same position $\mathbf{pc}'$.

  Finally, for every BIR variable $\mathbf{x}$ and symbolic BIR variable $\mathtt{x}^s$ that $\mathtt{I}\{s_j^s\}.\eta^s[\mathtt{x}^s] \neq \mathtt{I}\{s_i^s\}.\eta^s[\mathtt{x}^s]$ and $\mathtt{I}\{s_j^b\}.\eta^b[\mathbf{x}] \neq \mathtt{I}\{s_i^b\}.\eta^b[\mathbf{x}]$, we choose H' by extending H according to the executed operation such that $H'(\mathtt{I}\{s_j^s\}.\eta^s[\mathtt{x}^s]) = \mathtt{I}\{s_j^b\}.\eta^b[\mathbf{x}]$. Moreover, for the symbolic counter $\mathtt{t}^s$ which represents the number of iterations of the loop in symbolic execution and the concrete counter $\mathbf{t}$, we have $H'(\mathtt{t}^s) = \mathbf{t}$. Therefore, we can conclude that $(\eta^{b\prime}, \mathbf{pc}'), Q^{\iota\times b} \sim_{H'} (\Pi^{s\prime}, \eta^{s\prime}, \mathbf{pc}'), Q^{\iota\times s}$ and $\mathtt{loop}(\mathtt{t}^s) =_{H'} \mathbf{loop(t)}$.

□

We show the behavioral equivalence between the two layers by extending the simulation relation to execution traces $\sim_{H,k} \subseteq \mathcal{R}^{\iota\times b} \times \mathcal{R}^{\iota\times s}$ w.r.t an upper bound $k \in \mathbb{N}$ on the number of RNG steps. That is, $R^{\iota\times b} \sim_{H,k} R^{\iota\times s}$ holds, iff, $rng(R^{\iota\times b}) \leq k$, and for all $\mathtt{I}\{s^b\}, o^{\iota\times b} \in R^{\iota\times b}$ there exist $\mathtt{I}\{s^s\}, o^{\iota\times s} \in R^{\iota\times s}$ and an H s.t. $\mathtt{I}\{s^b\} \sim_H \mathtt{I}\{s^s\}$ and $o^{\iota\times b} =_H o^{\iota\times s}$.

Theorem 6.6 ($\mathtt{IML}_{\mathbf{B}}$-$\mathtt{IML}_{\mathtt{SB}}$ Trace Inclusion). *Let* $\mathbf{P}$ *be a* BIR *program,* $\mathtt{I}$ *be an* IML *process, and* $k \in \mathbb{N}$ *is any upper bound on RNG steps, then, for all* $\mathtt{IML}_{\mathbf{B}}$ *traces* $R^{\iota\times b} \in \mathcal{R}^{\iota\times b}(\mathtt{I}\{\mathbf{P}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm}_k, \mathbf{r_k} \mapsto 0])$ *s.t.* $rng(R^{\iota\times b}) \leq k$, *there are an* $\mathtt{IML}_{\mathtt{SB}}$ *trace* $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(\mathtt{I}\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm}_k^s, \mathbf{r_k} \mapsto 0])$ *and an* H *s.t.* $R^{\iota\times b} \sim_{H,k} R^{\iota\times s}$.

Proof. Theorem 6.6 shows that for all $\mathtt{IML}_{\mathbf{B}}$ traces, there is an equivalent $\mathtt{IML}_{\mathtt{SB}}$ trace through a properly chosen interpretation H. We prove Theorem 6.6 by induction on the length of the traces.

- **Base case**. The base case can be proved using Theorem 6.4.
- **Inductive case**. The inductive step can be proved using Theorem 6.5.

□

Theorem 6.6 shows that, for an appropriately chosen interpretation and random memory, symbolic and concrete executions of a **BIR** program are behaviorally equivalent. This holds in the mixed IML-(S)**BIR** semantics, i.e., when coupled with the same IML attacker and protocol partners.

6.1.3 Security Properties

From the simulation results between concrete **BIR**, symbolic **BIR** and extracted IML, we will now conclude our target result, which argues that probabilistic security results translate across these levels of abstraction. The security properties we consider, i.e., authentication and weak secrecy, are safety properties over event traces. Specifically, we consider a security property ψ as a polynomially decidable prefix-closed set of event traces such that $\{t \in \psi \mid \forall i \in \mathbb{N} : t[..i] \in \psi \implies (\exists j \in \mathbb{N} : j < i \ \wedge \ t[..j] \in \psi)\}$. Here, we use the following notation. Given a trace $t = o_0 o_1 \ldots$ and index i, we define $t[i] = o_i$ and $t[..i] = o_0 o_1 \ldots o_i$.

Example 6.1. For SSH, we show authentication between the events $Accept_S(PK_S, PK_C)$ (in the server model based on the binary from one of our case studies, TinySSH) and $Accept_C(PK_S, PK_C)$ (in the client model implemented based on the SSH specification) where PK_S and PK_C are the server's public key and the client's public key, respectively.

$$\begin{aligned}\textbf{Auth} = \{t \in \psi \mid \forall i \in \mathbb{N} : t[i] = Accept_S(PK_S, PK_C) \\ \implies (\exists j \in \mathbb{N} : j < i \ \wedge \ t[j] = Accept_C(PK_S, PK_C))\}\end{aligned}$$

We quantify the probability of a protocol remaining secure by considering the complementary probability: the sum of the probabilities of each violation. To avoid double counting, we only sum over the set of shortest violating prefixes, i.e., $\psi_\neg = \{t \notin \psi | \forall t'.t' \text{ is prefix of } t \implies t' \in \psi\}$. As security properties are prefix-closed, this captures the probability of a violation. The system we analyze consists of the protocol implementations in **BIR**. Say $\mathfrak{T}^\alpha$ denotes a set of event traces obtained from the respective set of execution traces $\mathcal{R}^\alpha \downarrow_{events}$, where for each execution trace in $\mathcal{R}^\alpha$, $tr(.)$ returns a corresponding event trace in $\mathfrak{T}^\alpha$, pr is a probability distribution function that computes the probability of an event trace and BS_n^k is a set of bit strings for generating k random numbers of length n, then:

Definition 6.1 (BIR Insecurity)**.** For a BIR program $\mathbf{P}$, an IML process $\mathtt{I}$, a security parameter $n \in \mathbb{N}$, and $k \in \mathbb{N}$ the size of BIR's random memory, the insecurity of $\mathtt{I}\{\mathbf{P}\}$ w.r.t. ψ is: $\mathbf{insec}(\mathtt{I}\{\mathbf{P}\}, n, k, \psi) = 2^{-n \cdot k} \cdot \sum_{\substack{\mathtt{rm_k} \in \mathtt{BS}_n^k \\ \mathtt{t}^{\iota \times b} \in \mathfrak{T}^{\iota \times b}_{\mathtt{rm_k}} \cap \psi_\neg}} pr(\mathtt{t}^{\iota \times b})$ where $\mathfrak{T}^{\iota \times b}_{\mathtt{rm_k}} = \mathfrak{T}^{\iota \times b}(\mathtt{I}\{\mathbf{P}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0])$.

After translating $\mathbf{P}$ into $(\!|\mathbf{P}|\!)^\iota$, we define insecurity in terms of IML's probabilistic semantics.

Definition 6.2 (IML Insecurity)**.** The insecurity of an IML process $\mathtt{I}$ w.r.t a trace property ψ and a security parameter n is $\mathbf{insec}(\mathtt{I}, n, \psi) = \sum_{\mathtt{t} \in \mathfrak{T}(\mathtt{I}, n) \cap \psi_\neg} pr(\mathtt{t})$ where $pr(s_0^\iota \xrightarrow{\twoheadrightarrow}_{\mathtt{p}_1} s_1^\iota \cdots s_{\mathtt{n}-1}^\iota \xrightarrow{\twoheadrightarrow}_{\mathtt{p}_\mathtt{n}} s_\mathtt{n}^\iota) = \prod_{1 \le i \le n} \mathtt{p}_\mathtt{i}$.

Note that definitions 6.1 and 6.2 coincide on $\mathtt{IML}_\mathbf{B}$ processes that do not contain the run-construct, as in this case, the RNG rule (like any other BIR rule) can never be applied and thus k be chosen to be 0. This applies to the $\mathtt{IML}_\mathbf{B}$ processes resulting from our translation.

Via [7, Thm. 4.3, Thm. 5.2] we obtain a bound for $\mathbf{insec}(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\}, n, \psi)$ from either of the backends, ProVerif or CryptoVerif. In cryptography, probability bounds are expressed as asymptotic functions in the security parameter. CryptoVerif provides a symbolic expression of such a probability bound and, furthermore, proves that the bound is negligible, i.e., it decreases faster than the inverse of any polynomial. On the other hand, ProVerif only confirms the existence of a negligible bound. In both cases, the existence of this negligible upper bound ensures $\mathbf{insec}(\mathtt{I}\{\mathbf{P}\}, n, k, \psi)$ is negligible.

Theorem 6.12 shows the translation is sound. Note that $\mathtt{I}$ contains BIR programs (via the run construct), but also IML processes that represent communication partners and the network attacker. In the following, we present the proof of Theorem 6.12. Based on definitions 6.1 and 6.2, we calculate the probability distribution in both IML and $\mathtt{IML}_\mathbf{B}$. While the probability of all transitions except for random number generation is 1, we need to demonstrate other requirements, such as extra randomness, injective event trace inclusion, etc. To this end, we present the following lemmas to show these requirements, which is necessary to prove Theorem 6.12.

Lemma 6.7 (Trace Contains Randomness)**.** *For a BIR program $\mathbf{P}$, an IML process $\mathtt{I}$, any upper bound $k \in \mathbb{N}$ on the number of RNG steps, all IML traces $\mathtt{R}^\iota \in \mathcal{R}^\iota(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\})$, all interpretations $\mathtt{H}$ and $\mathtt{H}'$, and all mixed IML and symbolic execution traces $R^{\iota \times s} \in \mathcal{R}^{\iota \times s}(\mathtt{I}\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm}_\mathtt{k}^s, \mathbf{r_k} \mapsto 0])$ with d the number of RNG steps in $R^{\iota \times s}$ such that $\mathtt{rm}_\mathtt{d}^s$ be the first d random symbolic values in $\mathbf{RM}$ and $d \le k$, then,*

$$R^{\iota \times s} \sim_{\mathtt{H}, k, (\!|\cdot|\!)^\iota} \mathtt{R}^\iota \wedge R^{\iota \times s} \sim_{\mathtt{H}', k, (\!|\cdot|\!)^\iota} \mathtt{R}^\iota \implies \mathtt{H}|_{\mathtt{rm}_\mathtt{d}^s} = \mathtt{H}'|_{\mathtt{rm}_\mathtt{d}^s}$$

Proof. Since the relation $R^{\iota\times s} \sim_{H,k,(|\cdot|)^\iota} R^\iota$ holds, we know that $R^{\iota\times s} = I\{s_0^s\} \xrightarrow{o_1^{\iota\times s}}\!\!\!\!\rightarrow_{p_1,H} \dots \xrightarrow{o_i^{\iota\times s}}\!\!\!\!\rightarrow_{p_i,H} I\{s_i^s\} \xrightarrow{Fr(x_1^s)}\!\!\!\!\rightarrow_{1,H} I\{s_{i+1}^s\} \xrightarrow{o_{i+2}^{\iota\times s}}\!\!\!\!\rightarrow_{p_{i+2},H} \dots \xrightarrow{o_j^{\iota\times s}}\!\!\!\!\rightarrow_{p_j,H} I\{s_j^s\} \xrightarrow{Fr(x_d^s)}\!\!\!\!\rightarrow_{1,H} I\{s_{j+1}^s\} \xrightarrow{o_{j+2}^{\iota\times s}}\!\!\!\!\rightarrow_{p_{j+2},H} \dots \!\!\rightarrow_{p_m,H} I\{s_m^s\}$ where $o_i^{\iota\times s}$ does not come from a RNG(n) call and $x_1^s, \dots, x_d^s = rm_d^s$ and $R^\iota = s_\emptyset^\iota \xrightarrow{o_1^\iota}\!\!\!\!\rightarrow_{p_1} \dots \xrightarrow{o_i^\iota}\!\!\!\!\rightarrow_{p_i} s_i^\iota \xrightarrow{fr(b_1)}\!\!\!\!\rightarrow_{\frac{1}{2^n}} s_{i+1}^\iota \dots \xrightarrow{o_j^\iota}\!\!\!\!\rightarrow_{p_j} s_j^\iota \xrightarrow{fr(b_d)}\!\!\!\!\rightarrow_{\frac{1}{2^n}} s_{j+1}^\iota \dots \!\!\rightarrow_{p_m} s_m^\iota$. For all $0 < i \le d$, then, $b_i = H(x_i^s)$ (†). Since $R^{\iota\times s} \sim_{H',k,(|\cdot|)^\iota} R^\iota$, for the random symbolic values $x_1^s, \dots, x_d^s$, we get that $b_i = H'(x_i^s)$ for $0 < i \le d$ (‡). From (†) and (‡), we conclude that $H|_{rm_d^s} = H'|_{rm_d^s}$.

□

Lemma 6.8 (Same Random Steps). *For a* **BIR** *program* **P**, *an* IML *process* I, *any upper bound* $k \in \mathbb{N}$ *on the number of RNG steps, a security parameter* $n \in \mathbb{N}$, *all* IML *traces* $R^\iota \in \mathcal{R}^\iota(I\{(|\mathbf{P}|)^\iota\})$, *all interpretations* H, *all mixed* IML *and symbolic execution traces* $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(I\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto rm_k^s, \mathbf{r_k} \mapsto 0])$, *if* $R^{\iota\times s} \sim_{H,k,(|\cdot|)^\iota} R^\iota$, *then, the number of* New *steps in* R^ι *and the number of RNG plus the number of* New *steps in* $R^{\iota\times s}$ *are equal. Moreover,* $pr(tr(R^\iota)) = pr(tr(R^{\iota\times s})) \cdot 2^{-n\cdot rng(R^{\iota\times s})}$.

Proof. Since $R^{\iota\times s} \sim_{H,k,(|\cdot|)^\iota} R^\iota$, we know that $Fr(x_i^s)$ or $fr(b)$ steps in $R^{\iota\times s}$ maps to $fr(H(x_i^s))$ or $fr(b)$ steps in R^ι, respectively, for $i \le rng(R^{\iota\times s})$. The first statement is as follows:

Based on the rule New in the operational semantics of IML output processes [7, p. 23], we get that the probability of generating the random number b based on the IML transition relation (i.e. $\rightarrow\!\!\!\!\rightarrow_{\frac{1}{2^n}}$ semantics) with respect to security parameter n is $\frac{1}{2^n}$. Therefore, the $fr(b)$ step has the probability $\frac{1}{2^n}$ in both $R^{\iota\times s}$ and R^ι. Hence, for $fr(b)$ steps, we have $pr(tr(R^\iota)) = pr(tr(R^{\iota\times s}))$.

Based on the rule RNG(n) in the semantics of IML_{SB}, Fig. 6.2, we get that the probability of generating the symbolic random number x^s based on the mixed IML and symbolic transition relation (i.e., $\rightarrow\!\!\!\!\rightarrow_{1,H}$ semantics) with respect to security parameter n is 1 but we translate $Fr(x^s)$ to $fr(H(x^s))$ using interpretation H which have the probability $1 \cdot \frac{1}{2^n}$. The number of $Fr(x^s)$ steps in $R^{\iota\times s}$ are $rng(R^{\iota\times s})$, hence, the probability of $Fr(x^s)$ steps is $pr(tr(R^{\iota\times s})) \cdot 2^{-n\cdot rng(R^{\iota\times s})}$.

Therefore, we can conclude that $pr(tr(R^\iota)) = pr(tr(R^{\iota\times s})) \cdot 2^{-n\cdot rng(R^{\iota\times s})}$.

□

Lemma 6.9 (Extra Randomness). *For a* **BIR** *program* **P**, *an* IML *process* I, *any upper bound* $k \in \mathbb{N}$ *on the number of RNG steps, a security parameter* $n \in \mathbb{N}$, *all* IML *traces* $R^\iota \in \mathcal{R}^\iota(I\{(|\mathbf{P}|)^\iota\})$, *all interpretations* H, *all mixed* IML *and symbolic execution traces* $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(I\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto rm_k^s, \mathbf{r_k} \mapsto 0])$ *with* $k = rng(R^{\iota\times s})$, *and for any* $l \ge k$, *if* $R^{\iota\times s} \sim_{H,k,(|\cdot|)^\iota} R^\iota$, *then, for all* H′ *with* $dom(H') \subseteq rm_l^s \setminus rm_k^s$, *we have,* $R^{\iota\times s} \sim_{H'|_{rm_k^s},k,(|\cdot|)^\iota} R^\iota$.

Proof. Since $R^{\iota\times s} \sim_{H,k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$, we know that the number of RNG steps in $R^{\iota\times s}$ is $rng(R^{\iota\times s}) \leq k$. Based on the semantics of $\mathtt{IML}_{\mathrm{SB}}$, Fig. 6.2, we know that random symbolic values $x_i^s \in rm_k^s$ for $0 < i \leq k$ are used in RNG(n) rule by order of i. The random symbolic values $x_j^s \in rm_l^s$ for $l \geq j \geq k$ are not generated by RNG(n) rule in Fig. 6.2. Therefore, the random symbolic values $x_j^s \in rm_l^s$ for $l \geq j \geq k$ are not part of the mixed IML and symbolic state after RNG(n) call. Hence, we can conclude that modification of H does not affect trace equivalence and we have $R^{\iota\times s} \sim_{H',k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$ such that $H \subseteq H'$ and $dom(H') \subseteq rm_l^s \setminus rm_k^s$.

□

Lemma 6.10 ($\mathtt{IML}_{\mathrm{SB}}$-IML Injective Event Trace Inclusion). *For a* **BIR** *program* **P**, *an* IML *process* I, *any security parameter* $n \in \mathbb{N}$, *an upper bound on the number of nonces* $k \in \mathbb{N}$, *and for* $\mathfrak{T}^{\iota\times s} = \mathfrak{T}^{\iota\times s}(\mathtt{I}\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto rm_k^s, \mathbf{r_k} \mapsto 0])$, *there is an injective function* ζ *from*

$$\left\{ \begin{pmatrix} t^{\iota\times s} \in \mathfrak{T}^{\iota\times s}, \\ H : rm_k^s \to \mathsf{BS}_n^k \end{pmatrix} \;\middle|\; \begin{array}{l} rng(t^{\iota\times s}) = k \wedge \\ \exists H'.\ H'|_{rm_k^s} = H \end{array} \right\}$$

to $\mathfrak{T}(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\}, n)$ *such that*

$$\zeta(t^{\iota\times s}, H) = t \implies \begin{pmatrix} \exists H', R^{\iota\times s}, \mathsf{R}^\iota. \\ R^{\iota\times s} \sim_{H',k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota \wedge \\ tr(R^{\iota\times s}) = t^{\iota\times s} \;\wedge\; tr(\mathsf{R}^\iota) = t \end{pmatrix}$$

Proof. From the skolemization of Theorem 6.3, we define $\zeta' : \mathcal{R}^{\iota\times s} \times \mathcal{H} \to \mathcal{R}^\iota$ such that $\zeta'(R^{\iota\times s}, H') = \mathsf{R}^\iota$ implies $R^{\iota\times s} \sim_{H',k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$. We define $\zeta : \mathfrak{T}^{\iota\times s} \times \mathcal{H} \to \mathfrak{T}$ such that $\zeta(t^{\iota\times s}, H) = t$. Hence, we choose arbitrary $R^{\iota\times s}$ and H' such that $tr(R^{\iota\times s}) = t^{\iota\times s}$ and $tr(\zeta'(R^{\iota\times s}, H')) = tr(\mathsf{R}^\iota) = t$.

Then, we prove the function ζ is injective by contradiction and assume for arbitrary $t^{\iota\times s}$, $t^{\iota\times s\prime}$, H, and H' such that $t^{\iota\times s} \neq t^{\iota\times s\prime} \vee H \neq H'$, we have $\zeta(t^{\iota\times s}, H) = t = \zeta(t^{\iota\times s\prime}, H')$. Therefore, $H(t^{\iota\times s}) = H'(t^{\iota\times s\prime})$ and for any event $o^{\iota\times s}$ where $t^{\iota\times s}$ and $t^{\iota\times s\prime}$ are different, we have $H(o^{\iota\times s}) = H'(o^{\iota\times s\prime})$. Hence, there exist two cases:

1. $o^{\iota\times s} \neq o^{\iota\times s\prime}$

   (a) $H = H'$: We assume that there exists a symbolic value d^s such that $H(Ev(d^s)) \neq H(Fr(d^s))$. Based on $R^{\iota\times s} \sim_{H,k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$, we have $\mathtt{ev}(H(d^s)) \in t$ at the position j and $Ev(d^s) \in t^{\iota\times s}$ at the same position j. Similarly, we have $\mathtt{fr}(H(d^s)) \in t$ at the position j and $Fr(d^s) \in t^{\iota\times s\prime}$ at the same position j. Hence, we get that we have two different t (i.e., at the position j). Since $o^{\iota\times s} \neq o^{\iota\times s\prime}$ gives a contradiction, we deduce that $o^{\iota\times s} = o^{\iota\times s\prime}$.

(b) $\mathrm{H} \neq \mathrm{H}'$: We assume that there exists a symbolic value d^s such that $\mathrm{H}(\mathrm{Ev}(\mathrm{d}^s)) \neq \mathrm{H}'(\mathrm{Fr}(\mathrm{d}^s))$. Based on $R^{\iota\times s} \sim_{\mathrm{H},k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$, we have $\mathtt{ev}(\mathrm{H}(\mathrm{d}^s)) \in \mathsf{t}$ at the position j and $\mathrm{Ev}(\mathrm{d}^s) \in \mathrm{t}^{\iota\times s}$ at the same position j. Based on $R^{\iota\times s\prime} \sim_{\mathrm{H}',k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$, we have $\mathtt{fr}(\mathrm{H}'(\mathrm{d}^s)) \in \mathsf{t}$ at the position j and $\mathrm{Fr}(\mathrm{d}^s) \in \mathrm{t}^{\iota\times s\prime}$ at the same position j. Hence, we get that we have two different t (i.e., at the position j). Since $o^{\iota\times s} \neq o^{\iota\times s\prime}$ gives a contradiction, we deduce that $o^{\iota\times s} = o^{\iota\times s\prime}$.

2. $\mathrm{H} \neq \mathrm{H}' \wedge o^{\iota\times s} = o^{\iota\times s\prime}$

(a) $o^{\iota\times s}$ is a $\mathrm{Fr}(\cdot)$ event: We assume that there exists a random symbolic value $\mathrm{x}_\mathrm{i}^s \in \mathrm{rm}_\mathrm{k}^s$ for $0 < i \leq k$ such that $\mathrm{H}(\mathrm{Fr}(\mathrm{x}_\mathrm{i}^s)) \neq \mathrm{H}'(\mathrm{Fr}(\mathrm{x}_\mathrm{i}^s))$. Based on $R^{\iota\times s} \sim_{\mathrm{H},k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$, we have $\mathtt{fr}(\mathrm{H}(\mathrm{x}_\mathrm{i}^s)) \in \mathsf{t}$ at the position j and $\mathrm{Fr}(\mathrm{x}_\mathrm{i}^s) \in \mathrm{t}^{\iota\times s}$ at the same position j. Similarly, for $\mathrm{t}^{\iota\times s\prime}$ and H' and the same position j. Therefore, $\mathrm{H}(\mathrm{x}_\mathrm{i}^s) = \mathrm{H}'(\mathrm{x}_\mathrm{i}^s)$. Hence, from $\mathrm{x}_\mathrm{i}^s \in \mathrm{rm}_\mathrm{k}^s$, we get that $\mathrm{H}(\mathrm{Fr}(\mathrm{x}_\mathrm{i}^s)) = \mathrm{H}'(\mathrm{Fr}(\mathrm{x}_\mathrm{i}^s))$. Since $\mathrm{H} \neq \mathrm{H}'$ gives a contradiction, we deduce that $\mathrm{H} = \mathrm{H}'$.

(b) $o^{\iota\times s}$ is a $\mathrm{Ev}(\cdot)$ event: We assume that there exists a symbolic value d^s such that $\mathrm{H}(\mathrm{Ev}(\mathrm{d}^s)) \neq \mathrm{H}'(\mathrm{Ev}(\mathrm{d}^s))$. Based on $R^{\iota\times s} \sim_{\mathrm{H},k,(\!|\cdot|\!)^\iota} \mathsf{R}^\iota$, we have $\mathtt{ev}(\mathrm{H}(\mathrm{d}^s)) \in \mathsf{t}$ at the position j and $\mathrm{Ev}(\mathrm{d}^s) \in \mathrm{t}^{\iota\times s}$ at the same position j. Similarly, for $\mathrm{t}^{\iota\times s\prime}$ and H' and the same position j. Because we generate symbolic values in a canonical form, $\mathrm{H}(\mathrm{d}^s) = \mathrm{H}'(\mathrm{d}^s)$. Therefore, we get that $\mathrm{H}(\mathrm{Ev}(\mathrm{d}^s)) = \mathrm{H}'(\mathrm{Ev}(\mathrm{d}^s))$. Since $\mathrm{H} \neq \mathrm{H}'$ gives a contradiction, we deduce that $\mathrm{H} = \mathrm{H}'$.

Therefore, we can conclude that the function ζ is injective.

□

Lemma 6.11 ($\mathtt{IML}_\mathbf{B}$-$\mathtt{IML}_{\mathrm{SB}}$ Injective Event Trace Inclusion)**.** *For a* **BIR** *program* $\mathbf{P}$*, an* IML *process* $\mathtt{I}$*, any security parameter* $n \in \mathbb{N}$*, an upper bound on the number of nonces* $k \in \mathbb{N}$*, and for* $\mathfrak{T}^{\iota\times b} = \mathfrak{T}^{\iota\times b}(\mathtt{I}\{\mathbf{P}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm}_\mathrm{k}, \mathbf{r_k} \mapsto 0])$*, there is an injective function* ζ *from* $\{(\mathrm{t}^{\iota\times b}, \mathtt{rm}_\mathrm{k}) \mid \mathtt{rm}_\mathrm{k} \in \mathsf{BS}_n^k \wedge \mathrm{t}^{\iota\times b} \in \mathfrak{T}^{\iota\times b}(\mathtt{I}\{\mathbf{P}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm}_\mathrm{k}, \mathbf{r_k} \mapsto 0])\}$ *to*

$$\left\{ \begin{pmatrix} \mathrm{t}^{\iota\times s} \in \mathfrak{T}^{\iota\times s}, \\ \mathrm{H} : \mathrm{rm}_\mathrm{k}^s \to \mathsf{BS}_n^k \end{pmatrix} \;\middle|\; s.t.\ \exists \mathrm{H}'.\mathrm{H}'|_{\mathrm{rm}_\mathrm{k}^s} = \mathrm{H} \right\}$$

such that

$$\zeta(\mathrm{t}^{\iota\times b}, \mathtt{rm}_\mathrm{k}) = (\mathrm{t}^{\iota\times s}, \mathrm{H}) \implies \begin{pmatrix} pr(\mathrm{t}^{\iota\times b}) \cdot 2^{-n\cdot k} = pr(\mathrm{t}^{\iota\times s}) \cdot 2^{-n\cdot k} \wedge \\ tr(R^{\iota\times b}) = \mathrm{t}^{\iota\times b} \;\wedge\; tr(R^{\iota\times s}) = \mathrm{t}^{\iota\times s} \end{pmatrix}$$

Proof. From the skolemization of Theorem 6.6, we define $\zeta' : \mathcal{R}^{\iota\times b} \times \mathsf{BS}_n^k \to \mathcal{R}^{\iota\times s} \times \mathcal{H}$ such that $\zeta'(R^{\iota\times b}, \mathtt{rm}_\mathrm{k}) = (R^{\iota\times s}, \mathrm{H})$ implies $R^{\iota\times b} \sim_{\mathrm{H},k} R^{\iota\times s}$. We define $\zeta : \mathfrak{T}^{\iota\times b} \times \mathsf{BS}_n^k \to \mathfrak{T}^{\iota\times s} \times \mathcal{H}$

such that $\zeta(\mathsf{t}^{\iota\times_b}, \mathtt{rm}_k) = (\mathsf{t}^{\iota\times_s}, \mathrm{H})$. Hence, we choose arbitrary $R^{\iota\times_b}$ and $\mathtt{rm}_k$ such that $tr(R^{\iota\times_b}) = \mathsf{t}^{\iota\times_b}$ and $tr(\zeta'(R^{\iota\times_b}, \mathtt{rm}_k)) = tr(R^{\iota\times_s}, \mathrm{H}) = \mathsf{t}^{\iota\times_s}$.

Then, we prove the function ζ is injective by contradiction and assume for arbitrary $\mathsf{t}^{\iota\times_b}$, $\mathsf{t}^{\iota\times_b\prime}$, $\mathtt{rm}_k$, and $\mathtt{rm}'_k$ such that $\mathsf{t}^{\iota\times_b} \neq \mathsf{t}^{\iota\times_b\prime} \vee \mathtt{rm}_k \neq \mathtt{rm}'_k$, we have $\zeta(\mathsf{t}^{\iota\times_b}, \mathtt{rm}_k) = (\mathsf{t}^{\iota\times_s}, \mathrm{H}) = \zeta(\mathsf{t}^{\iota\times_b\prime}, \mathtt{rm}'_k)$. Therefore, we get that $(\mathsf{t}^{\iota\times_b}, \mathtt{rm}_k) = \mathrm{H}(\mathsf{t}^{\iota\times_s}) = (\mathsf{t}^{\iota\times_b\prime}, \mathtt{rm}'_k)$. Hence, there exist two cases:

1. $\mathtt{rm}_k \neq \mathtt{rm}'_k$
   We assume that there exists a random value $\mathtt{x}_i \in \mathtt{rm}_k$ and a random value $\mathtt{x}_i\prime \in \mathtt{rm}'_k$ for $0 < i \leq k$ such that $\mathtt{x}_i \neq \mathtt{x}_i\prime$. Based on $R^{\iota\times_b} \sim_{\mathrm{H},k} R^{\iota\times_s}$, we have $\mathrm{Fr}(\mathrm{x}^s_i) \in \mathsf{t}^{\iota\times_s}$ at the position j and $\mathbf{Fr}(\mathtt{x}_i) \in \mathsf{t}^{\iota\times_b}$ at the same position j such that $\mathrm{H}(\mathrm{x}^s_i) = \mathtt{x}_i$. Based on $R^{\iota\times_b\prime} \sim_{\mathrm{H},k} R^{\iota\times_s}$, we have $\mathrm{Fr}(\mathrm{x}^s_i) \in \mathsf{t}^{\iota\times_s}$ at the position j and $\mathbf{Fr}(\mathtt{x}_i\prime) \in \mathsf{t}^{\iota\times_b\prime}$ at the same position j such that $\mathrm{H}(\mathrm{x}^s_i) = \mathtt{x}_i\prime$. Therefore, we get that $\mathtt{x}_i = \mathrm{H}(\mathrm{x}^s_i) = \mathtt{x}_i\prime$. Since $\mathtt{rm}_k \neq \mathtt{rm}'_k$ gives a contradiction, we deduce that $\mathtt{rm}_k = \mathtt{rm}'_k$.

2. $\mathsf{t}^{\iota\times_b} \neq \mathsf{t}^{\iota\times_b\prime} \wedge \mathtt{rm}_k = \mathtt{rm}'_k$
   Let $o^{\iota\times_b}$ and $o^{\iota\times_b\prime}$ be the earliest mixed IML and **BIR** events where $\mathsf{t}^{\iota\times_b}$ and $\mathsf{t}^{\iota\times_b\prime}$ are different. Based on $R^{\iota\times_b} \sim_{\mathrm{H},k} R^{\iota\times_s}$, we have $o^{\iota\times_s} \in \mathsf{t}^{\iota\times_s}$ at the position j and $o^{\iota\times_b} \in \mathsf{t}^{\iota\times_b}$ at the same position j such that $\mathrm{H}(o^{\iota\times_s}) = o^{\iota\times_b}$. Based on $R^{\iota\times_b\prime} \sim_{\mathrm{H},k} R^{\iota\times_s}$, we have $o^{\iota\times_s} \in \mathsf{t}^{\iota\times_s}$ at the position j and $o^{\iota\times_b\prime} \in \mathsf{t}^{\iota\times_b\prime}$ at the same position j such that $\mathrm{H}(o^{\iota\times_s}) = o^{\iota\times_b\prime}$. Therefore, we get that $o^{\iota\times_b} = \mathrm{H}(o^{\iota\times_s}) = o^{\iota\times_b\prime}$. Since $\mathsf{t}^{\iota\times_b} \neq \mathsf{t}^{\iota\times_b\prime}$ gives a contradiction, we deduce that $\mathsf{t}^{\iota\times_b} = \mathsf{t}^{\iota\times_b\prime}$.

Therefore, we can conclude that the function ζ is injective. Since $R^{\iota\times_b} \sim_{\mathrm{H},k} R^{\iota\times_s}$, we know that $\mathrm{Fr}(\mathrm{x}^s_i)$ or $\mathtt{fr}(\mathtt{b})$ steps in $R^{\iota\times_s}$ maps to $\mathbf{Fr}(\mathrm{H}(\mathrm{x}^s_i))$ or $\mathtt{fr}(\mathtt{b})$ steps in $R^{\iota\times_b}$, respectively, for $i \leq k$. The first statement is as follows:

Based on the rule New in the operational semantics of IML output processes [7, p. 23], we get that the probability of generating the random number $\mathtt{b}$ based on the IML transition relation (i.e. $\twoheadrightarrow_{\frac{1}{2^n}}$ semantics) with respect to security parameter n is $\frac{1}{2^n}$. Therefore, the $\mathtt{fr}(\mathtt{b})$ step has the probability $\frac{1}{2^n}$ in both $\mathsf{t}^{\iota\times_b}$ and $\mathsf{t}^{\iota\times_s}$. Hence, for $\mathtt{fr}(\cdot)$ steps, we have $pr(\mathsf{t}^{\iota\times_b}) = pr(\mathsf{t}^{\iota\times_s})$.

Based on the rule RNG(n) in the semantics of $\mathrm{IML}_{\mathrm{SB}}$, Fig. 6.2, we get that the probability of generating the symbolic random number x^s based on the mixed IML and symbolic transition relation (i.e., $\twoheadrightarrow_{1,\mathrm{H}}$ semantics) with respect to security parameter n is 1 but we map $\mathrm{Fr}(\mathrm{x}^s)$ to $\mathbf{Fr}(\mathrm{H}(\mathrm{x}^s))$ using interpretation H which have the probability $1 \cdot \frac{1}{2^n}$. The number of $\mathrm{Fr}(\mathrm{x}^s)$ steps in $\mathsf{t}^{\iota\times_s}$ are k, hence, the probability of $\mathrm{Fr}(\mathrm{x}^s)$ steps is $pr(\mathsf{t}^{\iota\times_s}) \cdot 2^{-n\cdot k}$.

Based on the rule normal in the mixed semantics of IML and **BIR** Fig. 6.3 and Sec. 5.1.1, we get that the probability of generating the random number $\mathbf{x}$ based on the mixed IML

and BIR transition relation (i.e., $\twoheadrightarrow_1$ semantics) with respect to security parameter n is 1 but we extracting $\mathbf{x}$ from the random memory RM with length n which have the probability $1 \cdot \frac{1}{2^n}$. The number of $\mathbf{Fr}(\mathtt{x})$ steps in $\mathtt{t}^{\iota \times b}$ are k, hence, the probability of $\mathbf{Fr}(\mathtt{x})$ steps is $pr(\mathtt{t}^{\iota \times b}) \cdot 2^{-n \cdot k}$.

Therefore, we can conclude that $pr(\mathtt{t}^{\iota \times b}) \cdot 2^{-n \cdot k} = pr(\mathtt{t}^{\iota \times s}) \cdot 2^{-n \cdot k}$.

□

Theorem 6.12 (Translation Preserves Attacks). *Given a* BIR *program* $\mathbf{P}$*, an* IML *process* $\mathtt{I}$*, a security parameter* $n \in \mathbb{N}$*, a trace property* ψ *and an upper bound* $k \in \mathbb{N}$ *on the number of RNG steps in* $\mathfrak{T}^{\iota \times b}(\mathtt{I}\{\mathbf{P}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm}_\mathtt{k}, \mathbf{r_k} \mapsto 0])$*, we get that*

$$\mathbf{insec}(\mathtt{I}\{\mathbf{P}\}, n, k, \psi) \leq \mathbf{insec}(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\}, n, \psi).$$

Proof. Throughout the following proof, we use the following property of sums: if $\zeta : A \to B$ is an injection, then $\sum_{a \in A} f(\zeta(a)) \leq \sum_{b \in B} f(b)$. Recall also that, if $\zeta : A \to B$ is injective, then $\zeta|_{A'}$ is injective for any A'. Also, for brevity, let $\mathfrak{T}^{\iota \times s} = \mathfrak{T}^{\iota \times s}(\mathtt{I}\{\mathbf{P}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm}_\mathtt{k}^s, \mathbf{r_k} \mapsto 0])$ and $\mathfrak{T}^{\iota \times b} = \mathfrak{T}^{\iota \times b}(\mathtt{I}\{\mathbf{P}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm}_\mathtt{k}, \mathbf{r_k} \mapsto 0])$ in the follow up.

$$\mathbf{insec}(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\}, n, \psi)$$

$$\overset{\text{Def. 6.2}}{=} \sum_{\mathtt{t} \in \mathfrak{T}(\mathtt{I}\{(\!|\mathbf{P}|\!)^\iota\}, n) \cap \psi_\neg} pr(\mathtt{t})$$

Note that $R^{\iota \times s} \sim_{\mathtt{H}, k, (\!|\cdot|\!)^\iota} \mathtt{R}^\iota$ implies $\mathtt{H}(\mathtt{t}^{\iota \times s}) \in \psi_\neg \Leftrightarrow \mathtt{t} \in \psi_\neg$

$$\overset{\text{Lem. 6.10,Lem. 6.8}}{\geq} \sum_{\substack{\mathtt{t}^{\iota \times s} \in \mathfrak{T}^{\iota \times s} \\ \mathtt{H}:\mathtt{rm}_\mathtt{l}^s \to \mathtt{BS}_n^k \\ \text{s.t. } \exists \mathtt{H}'.\ \mathtt{H}'|_{\mathtt{rm}_\mathtt{l}^s} = \mathtt{H} \\ \wedge \mathtt{H}'(\mathtt{t}^{\iota \times s}) \in \psi_\neg \\ \wedge rng(\mathtt{t}^{\iota \times s}) = l}} 2^{-nl} pr(\mathtt{t}^{\iota \times s})$$

Let k be the maximal l in the sum. We extend the range of $\mathtt{H}$, which increases the range of the sum by $2^{n(k-l)}$:

$$= \sum_{\substack{\mathtt{t}^{\iota \times s} \in \mathfrak{T}^{\iota \times s} \\ \mathtt{H}:\mathtt{rm}_\mathtt{k}^s \to \mathtt{BS}_n^k \\ \text{s.t. } \exists \mathtt{H}'.\ \mathtt{H}'|_{\mathtt{rm}_\mathtt{l}^s} = \mathtt{H}|_{\mathtt{rm}_\mathtt{l}^s} \\ \wedge \mathtt{H}'(\mathtt{t}^{\iota \times s}) \in \psi_\neg \\ \wedge rng(\mathtt{t}^{\iota \times s}) = l}} \underbrace{\frac{2^{-nl}}{2^{n(k-l)}}}_{=2^{-nk}} \cdot pr(\mathtt{t}^{\iota \times s})$$

$$\overset{\text{Lem. 6.9}}{=} \sum_{\substack{\mathsf{t}^{\iota\times s}\in\mathfrak{T}^{\iota\times s}\\ \mathsf{H}:\mathsf{rm}_{\mathsf{k}}^{s}\to\mathsf{BS}_{n}^{k}\\ \text{s.t. }\exists\mathsf{H}'.\mathsf{H}'|_{\mathsf{rm}_{\mathsf{k}}^{s}}=\mathsf{H}\\ \wedge\mathsf{H}'(\mathsf{t}^{\iota\times s})\in\psi_{\neg}}} 2^{-nk}pr(\mathsf{t}^{\iota\times s})$$

$$=2^{-nk}\cdot \sum_{\substack{\mathsf{t}^{\iota\times s}\in\mathfrak{T}^{\iota\times s}\\ \mathsf{H}:\mathsf{rm}_{\mathsf{k}}^{s}\to\mathsf{BS}_{n}^{k}\\ \text{s.t. }\exists\mathsf{H}'.\mathsf{H}'|_{\mathsf{rm}_{\mathsf{k}}^{s}}=\mathsf{H}\\ \wedge\mathsf{H}'(\mathsf{t}^{\iota\times s})\in\psi_{\neg}}} pr(\mathsf{t}^{\iota\times s})$$

Note that $R^{\iota\times b} \sim_{\mathsf{H},k} R^{\iota\times s}$ implies $\mathsf{H}(\mathsf{t}^{\iota\times s}) \in \psi_{\neg} \Leftrightarrow \mathsf{t}^{\iota\times b} \in \psi_{\neg}$

$$\overset{\text{Lem. 6.11}}{\geq} 2^{-nk}\cdot \sum_{\substack{\mathsf{rm}_{\mathsf{k}}\in\mathsf{BS}_{n}^{k}\\ \mathsf{t}^{\iota\times b}\in\mathfrak{T}^{\iota\times b}\cap\psi_{\neg}}} pr(\mathsf{t}^{\iota\times b})$$

$$\overset{\text{Def. 6.1}}{=} \mathbf{insec}(\mathtt{I}\{\mathbf{P}\}, n, k, \psi)$$

□

6.1.4 Multi-Programs Proof

In this section, we extend our results for multiple programs by establishing theorems 6.13 to 6.15. Given an IML process $\mathtt{I}$ and implementations $\mathbf{P_1}, ..., \mathbf{P_m}$ of protocol participants in BIR, we denote $\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}$ which parties running in parallel with an IML attacker and communicate through a channel. For $\mathbf{P_1}, ..., \mathbf{P_m}$ which are symbolically executed and translated into IML processes $(\!|\mathbf{P_1}|\!)^{\iota}, ..., (\!|\mathbf{P_m}|\!)^{\iota}$, the IML process $\mathtt{I}\{(\!|\mathbf{P_1}|\!)^{\iota}, ..., (\!|\mathbf{P_m}|\!)^{\iota}\}$ describes the parallel composition of m parties in the presence of an attacker. The following theorem indicates that $\mathtt{IML}_{\mathtt{SB}}$ and IML preserve the simulation relation for m programs.

Theorem 6.13 ($\mathtt{IML}_{\mathtt{SB}}$-IML Trace Inclusion*). *Let $\mathbf{P_1}, ..., \mathbf{P_m}$ be BIR programs, $\mathtt{I}$ be an IML process and $k \in \mathbb{N}$ is any upper bound on RNG steps, then, for all mixed IML and symbolic execution traces $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^s[\mathbf{RM} \mapsto \mathsf{rm}_{\mathsf{k}}^s, \mathbf{r_k} \mapsto 0])$ such that $rng(R^{\iota\times s}) \leq k$, there exist an IML trace $\mathsf{R}^{\iota} \in \mathcal{R}^{\iota}(\mathtt{I}\{(\!|\mathbf{P_1}|\!)^{\iota}, ..., (\!|\mathbf{P_m}|\!)^{\iota}\})$ and an interpretation H such that $R^{\iota\times s} \sim_{\mathsf{H},(\!|.|\!)^{\iota},k} \mathsf{R}^{\iota}$.*

Proof. By Theorem 6.3, for each $0 < i \leq m$, we have that for all mixed IML and symbolic execution traces $R_i^{\iota\times s} \in \mathcal{R}_i^{\iota\times s}(\mathtt{I}\{\mathbf{P_i}\}, \eta_0^s[\mathbf{RM} \mapsto \mathsf{rm}_{\mathsf{k}}^s, \mathbf{r_k} \mapsto 0])$, exist an IML trace $\mathsf{R}_{\mathsf{i}}^{\iota} \in \mathcal{R}_{\mathsf{i}}^{\iota}(\mathtt{I}\{(\!|\mathbf{P_i}|\!)^{\iota}\})$ and an interpretation $\mathsf{H_i}$ s.t. $R_i^{\iota\times s} \sim_{\mathsf{H_i},(\!|.|\!)^{\iota},k} \mathsf{R}_{\mathsf{i}}^{\iota}$. Then, we can conclude that for all mixed IML and symbolic execution traces $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^s[\mathbf{RM} \mapsto \mathsf{rm}_{\mathsf{k}}^s, \mathbf{r_k} \mapsto 0])$, there exist an IML trace $\mathsf{R}^{\iota} \in \mathcal{R}^{\iota}(\mathtt{I}\{(\!|\mathbf{P_1}|\!)^{\iota}, ..., (\!|\mathbf{P_m}|\!)^{\iota}\})$ and a H such that $R^{\iota\times s} \sim_{\mathsf{H},(\!|.|\!)^{\iota},k} \mathsf{R}^{\iota}$ and $\mathsf{H_i} \subseteq \mathsf{H}$ for $0 < i \leq m$. □

Theorem 6.13 proves that the IML model resulting from the translation of m programs covers all behaviors in the mixed IML and symbolic execution semantics. To ensure that the extracted IML model for m protocol parties preserves the semantics of their implementations in binary, we have to show Theorem 6.14.

Theorem 6.14 ($\mathtt{IML_B}$-$\mathtt{IML_{SB}}$ Trace Inclusion*). *Let* $\mathbf{P_1}, ..., \mathbf{P_m}$ *be* **BIR** *programs,* $\mathtt{I}$ *be an* IML *process and* $k \in \mathbb{N}$ *is any upper bound on RNG steps, then, for all mixed* IML *and* **BIR** *traces* $R^{\iota\times b} \in \mathcal{R}^{\iota\times b}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0])$ *such that* $rng(R^{\iota\times b}) \leq k$, *there exist a mixed* IML *and* SBIR *trace* $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm_k^s}, \mathbf{r_k} \mapsto 0])$ *and an* H *such that* $R^{\iota\times b} \sim_{\mathtt{H},k} R^{\iota\times s}$.

Proof. By Theorem 6.6, for each $0 < i \leq m$, we have for all mixed IML and **BIR** traces $R_i^{\iota\times b} \in \mathcal{R}_i^{\iota\times b}(\mathtt{I}\{\mathbf{P_i}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0])$, there is a mixed IML and symbolic execution trace $R_i^{\iota\times s} \in \mathcal{R}_i^{\iota\times s}(\mathtt{I}\{\mathbf{P_i}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm_k^s}, \mathbf{r_k} \mapsto 0])$, and an interpretation $\mathtt{H_i}$ such that $R_i^{\iota\times b} \sim_{\mathtt{H_i},k} R_i^{\iota\times s}$. Then, we can conclude that for all mixed IML and **BIR** traces $R^{\iota\times b} \in \mathcal{R}^{\iota\times b}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0])$, there exist a mixed IML and symbolic execution trace $R^{\iota\times s} \in \mathcal{R}^{\iota\times s}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^s[\mathbf{RM} \mapsto \mathtt{rm_k^s}, \mathbf{r_k} \mapsto 0])$ and an interpretation H such that $R^{\iota\times b} \sim_{\mathtt{H},k} R^{\iota\times s}$ and $\mathtt{H_i} \subseteq \mathtt{H}$ for $0 < i \leq m$. □

Theorem 6.14 states that symbolically executed **BIR** programs $\mathbf{P_1}, ..., \mathbf{P_m}$ preserve all behaviors of the same programs in the concrete execution for an appropriately chosen interpretation and random memory. In the following, we measure the success probability of the attacker for **BIR** programs in $\mathtt{IML_B}$ execution semantics $\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}$ and extracted IML process $\mathtt{I}\{(\!|\mathbf{P_1}|\!)^\iota, ..., (\!|\mathbf{P_m}|\!)^\iota\}$. Then we show by Theorem 6.15 that $\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}$ is at least as secure as $\mathtt{I}\{(\!|\mathbf{P_1}|\!)^\iota, ..., (\!|\mathbf{P_m}|\!)^\iota\}$ with respect to any trace property ψ, security parameter n and upper bound k on the number of RNG steps.

Theorem 6.15 (Translation Preserves Attacks*). *Given an* IML *process* $\mathtt{I}$, **BIR** *programs* $\mathbf{P_1}, ..., \mathbf{P_m}$, *a security parameter* $n \in \mathbb{N}$, *a trace property* ψ *and an upper bound* $k \in \mathbb{N}$ *on the number of RNG steps in* $\mathfrak{T}^{\iota\times b}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, \eta_0^b[\mathbf{RM} \mapsto \mathtt{rm_k}, \mathbf{r_k} \mapsto 0])$, *we get that* $\mathbf{insec}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, n, k, \psi) \leq \mathbf{insec}(\mathtt{I}\{(\!|\mathbf{P_1}|\!)^\iota, ..., (\!|\mathbf{P_m}|\!)^\iota\}, n, \psi)$

Proof. The insecurity of $\mathtt{I}\{(\!|\mathbf{P_1}|\!)^\iota, ..., (\!|\mathbf{P_m}|\!)^\iota\}$ w.r.t. ψ is as follows:

$\mathbf{insec}(\mathtt{I}\{(\!|\mathbf{P_1}|\!)^\iota, ..., (\!|\mathbf{P_m}|\!)^\iota\}, n, \psi)$

$$= \sum_{\mathtt{t}\in\mathfrak{T}(\mathtt{I}\{(\!|\mathbf{P_1}|\!)^\iota,...,(\!|\mathbf{P_m}|\!)^\iota\},n)\cap\psi_\neg} pr(\mathtt{t})$$

Note that by Theorem 6.13, we have $R^{\iota\times s} \sim_{\mathtt{H},k,(\!|\cdot|\!)^\iota} \mathtt{R}^\iota$ which implies $\mathtt{H}(\mathtt{t}^{\iota\times s}) \in \psi_\neg \Leftrightarrow \mathtt{t} \in \psi_\neg$

$$= \sum_{0<i\leq m} \sum_{\substack{\mathtt{t_i}\in \\ \mathfrak{T}_\mathtt{i}((\mathtt{I}\{(\!|\mathbf{P_i}|\!)^\iota\},n)\cap\psi_\neg}} pr(\mathtt{t_i})$$

$$\overset{\text{Def. 6.2}}{=} \sum_{0<i\leq m} \mathbf{insec}(\mathtt{I}\{(\!|\mathbf{P_i}|\!)^{\iota}\}, n, \psi)$$

$$\overset{\text{Thm. 6.12}}{\geq} \sum_{0<i\leq m} \mathbf{insec}(\mathtt{I}\{\mathbf{P_i}\}, n, k, \psi)$$

$$\overset{\text{Def. 6.1}}{=} \sum_{0<i\leq m} 2^{-n\cdot k} \cdot \sum_{\substack{\mathtt{rm_k}\in\mathsf{BS}_n^k \\ \mathtt{t}_i^{\iota\times b}\in\psi_\neg\cap \\ \mathfrak{T}_i^{\iota\times b}(\mathtt{I}\{\mathbf{P_i}\},\eta_0^b[\mathbf{RM}\mapsto\mathtt{rm_k},\mathbf{r_k}\mapsto 0])}} pr(\mathtt{t}_i^{\iota\times b})$$

Note that by Theorem 6.14, we have $R^{\iota\times b} \sim_{\mathrm{H},k} R^{\iota\times s}$ which implies $\mathrm{H}(\mathtt{t}^{\iota\times s}) \in \psi_\neg \Leftrightarrow \mathtt{t}^{\iota\times b} \in \psi_\neg$

$$= 2^{-nk} \cdot \sum_{\substack{\mathtt{rm_k}\in\mathsf{BS}_n^k \\ \mathtt{t}^{\iota\times b}\in\mathfrak{T}^{\iota\times b}\cap\psi_\neg}} pr(\mathtt{t}^{\iota\times b})$$

$$= \mathbf{insec}(\mathtt{I}\{\mathbf{P_1}, ..., \mathbf{P_m}\}, n, k, \psi)$$

□

6.2 In Symbolic Setting

We instantiate our general framework (introduced in Part I) with different languages: (a) ARMv8 and RISC-V for verifying implementations of real-world protocols, (b) SAPIC$^+$ for modeling parties from the specification, and (c) DY rules for specifying our threat model. This section demonstrates how the theorems presented in Chapter 3 simplified the end-to-end proof in the symbolic setting, enabling us to mechanize it.

6.2.1 Specific Deduction Combiners

The parallel composition of SBIR and the DY attacker employs the combined deduction relation $\vdash^{\mathrm{bit}'}_{s_A}$, which represents a specialized variant of the combined deduction relation $\vdash^{\mathrm{bit}}_{12}$ (see Sec. 3.3.3) for SBIR, as presented below:

$$\Pi^s \uplus \Pi_A \vdash^{\mathrm{bit}'}_{s_A} \mathcal{K}(z) \Leftrightarrow \exists\, x, y, w.$$
$$\mathcal{K}(y) \in \Pi_A \wedge (y \doteq x \diamond_{\mathbf{b}} w) \in \Pi^s \wedge z \in \big(symbols(x) \cup symbols(w)\big) \quad (\mathrm{bit}')$$

During our symbolic execution, a logical predicate may be added into the SBIR predicate set (i.e., Π^{s}) and an SBIR event arises. For example, an equality predicate, represented as $\doteq$, is added to the SBIR predicate set as a result of processing the **assign** statement. Additionally, the DY attacker's predicate set (i.e., Π_{A}) is updated due to the combined deduction relation $\vdash^{\dot{\mapsto}}_{LA}$ and synchronization (Table 2.1 summarizes synchronization events). Recall that $\|^{\vdash^{\dot{\mapsto}}_{LA}}_{s}$ uses a deduction combiner specific to the DY attacker and library (defined in Sec. 3.3.2.1), while $\|^{\vdash^{\mathsf{bit}'}_{sA}}_{s}$ utilizes a specialized deduction relation between SBIR and DY. Moreover, $\|^{\vdash^{(\!|\mathsf{bit}'|\!)^{sp}}_{spA}}_{s}$ employs a deduction relation similar to $\vdash^{\mathsf{bit}'}_{sA}$, referred to as $\vdash^{(\!|\mathsf{bit}'|\!)^{sp}}_{spA}$, which particularly applies to SAPIC^{-} and DY predicate sets. The distinction from $\vdash^{\mathsf{bit}'}_{sA}$ lies in the fact that $\vdash^{(\!|\mathsf{bit}'|\!)^{sp}}_{spA}$ incorporates the translation of the binary operators $\diamond_{\mathsf{b}}$ as function symbols (see Fig. 5.7 for the translation of the binary operations).

Example 6.2. This example presents the sequence of **BIR** statements of example 3.2, along with the corresponding updates resulting from symbolic execution, model extraction and the deduction combiner $\vdash^{\mathsf{bit}'}_{sA}$. As shown in the last column of the illustrated box, the DY attacker gains further logical facts by using the deduction combiner $\vdash^{\mathsf{bit}'}_{sA}$ together with the DY and SBIR predicate sets. We use a number next to each piece of knowledge, to indicate in which order these facts are acquired in our example. We employed the combined deduction relations $\vdash^{\mathsf{bit}'}_{sA}$ and $\vdash^{(\!|\mathsf{bit}'|\!)^{sp}}_{spA}$ and extracted the presented SAPIC^{-} model using our toolchain to demonstrate the application of these combined deduction relations. This model reflects $\vdash^{\mathsf{bit}'}_{sA}$ and $\vdash^{(\!|\mathsf{bit}'|\!)^{sp}}_{spA}$ as destructors defined in the translation from SAPIC^{-} (and Co.) to SAPIC^{+}. These destructors derive the same terms that $\vdash^{\mathsf{bit}'}_{sA}$ derives in this example.
In this example, '*de*..' represents the constant value **0xde**.. and jumps (at lines 0, 3, 4, and 6) are the translation of *branch and link* instruction used for function calls in ARM, which requires updating the *link register* R30. We present this register update in [..] to mean that it is not relevant to what we intend to present in this example. Note that the **BIR** representation is simplified w.r.t. to the implementation in HolBA.

	BIR Statement	SBIR Predicate	SBIR Event	SAPIC^{-} Process	DY Predicate	Via $\vdash^{\mathsf{bit}'}_{sA}$	
0	[R30=1;]**jmp** (0x44) //rng	–	$SFr(k)$	new k;	$\mathsf{Fr}(k)$	–	
1	**assign** (R1, **var** (R0))	$R1 \doteq \mathbf{var}(k)$	$\mathsf{Asn}(R1, \mathbf{var}(k))$	let $R1 = k$ in	–	$\mathcal{K}(k)$	(2)
2	**assign** (R0, **m**)	$R0 \doteq \mathbf{m}$	$\mathsf{Asn}(R0, \mathbf{m})$	let $R0 = m$ in	–	–	
3	[R30=4;]**jmp** (0x20) //senc	$c \mapsto \mathsf{senc}(R0, R1)$	$FCall(\mathsf{senc}, R0, R1, c)$	let $c = \mathsf{senc}(R0, R1)$ in	$c \dot{\mapsto} \mathsf{senc}(R0, R1)$	–	
4	[R30=5;]**jmp** (0x04) //send	–	$P2A(c)$	out(c);	$\mathcal{K}(c)$	$\mathcal{K}(R0)$	(3)
5	**assign** (R2, R1⊕**0xde**..)	$R2 \doteq R1 \oplus \mathbf{0xde}..$	$\mathsf{Asn}(R2, R1 \oplus \mathbf{0xde}..)$	let $R2 = \mathsf{xor}(R1, \text{'}de..\text{'})$ in	–	–	
6	[R30=7;]**jmp** (0x04) //send	–	$P2A(R2)$	out($R2$);	$\mathcal{K}(R2)$	$\mathcal{K}(R1)$	(1)

6.2.2 Translation to SAPIC^{-}

To enable transferring verified properties from the SAPIC^{+} level back to **BIR** and then to the protocols' binary, it is essential to prove that the extracted SAPIC^{-} model preserves

the behaviors of the SBIR representation. To this end, we establish a proof that for every path in the symbolic execution tree T, there exists an equivalent SAPIC$^-$ trace derived from executing translated process $(\!|\mathrm{T}|\!)^{sp}$.

Since SAPIC$^+$ uses the event K to signify messages coming from the attacker instead of *A2P*, we use $\langle\!|\cdot|\!\rangle$ to translate between trace (sets) of SAPIC$^+$ and SAPIC$^-$/SBIR. Besides K, security properties in SAPIC$^+$ can only refer to events in the process, which $\langle\!|\cdot|\!\rangle$ keeps the same.

Theorem 6.16 (Trace Inclusion). *Let* T *be a* SBIR *execution tree. Then, all translated* SBIR *traces of* T, $\langle\!|\mathfrak{T}^s(\mathrm{T})|\!\rangle$, *are included in the traces of the translated* SAPIC$^-$ *process* $\mathfrak{T}^{sp}((\!|\mathrm{T}|\!)^{sp})$.

Proof. The proof is done by induction on the length of the translated traces $\langle\!|\mathfrak{T}^s(\mathrm{T})|\!\rangle$. In the base case, no actions are taken. For the inductive case, we apply the case distinction over synchronous and asynchronous events in the set of SBIR events. We mechanized Theorem 6.16's proof in HOL4 (see *Symbtree-to-Sapic*). □

Lindner et al. [115, Thm. 4.1] demonstrated that verified properties for SBIR transfer to **BIR**, ensuring that the verified properties hold for concrete execution semantics.

6.2.3 End-to-End Correctness Result (using Part I)

We then show how the instantiation of theorems in Part I come together to simplify the analysis of our target language, which we will equip with DY semantics. Our analysis below includes embedded links to the mechanized proof for each step. We start with the concrete, complete ARMv8 program in parallel with an unspecified attacker A.

$$\mathfrak{T}^c(C^{ARMv8} \parallel_c A)$$

As is often the case, we take a detour via an intermediary language, in our case, **BIR**. [116, Thm. 2] justifies this so-called *lifting* step, i.e., shows that this translation is semantics preserving. Thanks to Theorem 3.6, we can use this theorem in context with the attacker A.

$$= \mathfrak{T}^c(C^{\mathbf{BIR}} \parallel_c A)$$

We next require (Assumption 1) that the complete **BIR** program $C^{\mathbf{BIR}}$ is trace-equivalent to $P^{\mathbf{BIR}} \parallel_c L^{\mathbf{BIR}}$, i.e., that it can be split into a program-under-verification ($P^{\mathbf{BIR}}$) and a known library ($L^{\mathbf{BIR}}$). Sec.5.1 provides statically checkable criteria for **BIR** to verify this condition automatically. Again, Theorem 3.6 is used to apply this in context. Afterwards, we use the refinement theorem Theorem 3.4 and the relations indicated by the underbraces to move from the concrete to the symbolic. An interpretation function ι

evaluates SBIR symbolic expressions to **BIR** concrete values, as demonstrated in [115]. Because $\|_c$ is associative w.r.t. trace equivalence, we have:

$$= \mathfrak{T}^c(\underbrace{P^{\mathbf{BIR}}}_{\substack{[115,\text{ Thm. 4.1}]\\ \sqsubseteq}} \|_c \underbrace{L^{\mathbf{BIR}} \|_c A}_{\substack{\text{A2: Deduction Soundness}\\ \sqsubseteq}})$$

$$\sqsubseteq \mathfrak{T}^s(\overbrace{P^{\text{SBIR}}}^{} \|_s^{\vdash^{\text{bit}'}_{sA}} \overbrace{L^{DY} \|_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY}}^{})$$

The first relation is the soundness of symbolic execution. The second is an assumption on the attacker that we will talk about in a second. We use Theorem 3.2 (case 3) to apply our translation result from SBIR to SAPIC$^-$ (Theorem 6.16) that we showed in the previous subsection (note that $P^{\text{SAPIC}^-} = (\!|P^{\text{SBIR}}|\!)^{sp}$, $\vdash^{\text{bit}'}_{sA}$ is disabling, and $\vdash^{(\!|\text{bit}'|\!)^{sp}}_{spA}$ is enabling). We have:

$$\subseteq \mathfrak{T}^s(P^{\text{SAPIC}^-} \|_s^{\vdash^{(\!|\text{bit}'|\!)^{sp}}_{spA}} L^{DY} \|_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY})$$

We combine SAPIC$^-$ with the DY attacker and library to SAPIC$^+$. As a by-product, this step shows the correctness of both w.r.t. the DY semantics in SAPIC$^+$ (which are quite standard). Hence, the above system is

$$= \mathfrak{T}^s(P^{\text{SAPIC}^+})$$

We thus have an end-to-end correctness result. Thanks to the framework theorems in Part I, this proof can be adapted to many other languages, as the researcher needs to only show the correctness of the language-specific steps (correctness of lifting, splitting, and translation) when adapting. Moreover, they only need to be shown in isolation. Until now, the translation step was usually shown in the presence of the adversary [106, 7, P1] and also as the first step of our work presented in Sec. 6.1.

While Assumption 1 delineates the class of programs that is supported (and can be checked statically), Assumption 2 (short: A2) formalizes our threat model: whatever type of system the attacker controls, it can be abstracted as a DY attacker if we also abstract the library in the same way.

Computational soundness says that any computational trace (i.e., a trace produced by the protocol implementation and some probabilistic polynomial-time (PPT) Turing machine) is either improbable or an instance of a symbolic trace with a DY attacker. Cortier and Warinschi found that computational soundness can be obtained from two conditions: deduction soundness and the commutation property [64]. The appeal of this approach is that deduction soundness is somewhat composable [64, 50], while computational soundness is not (as far as we know), although deduction soundness

without the commutation property provides only guarantees against passive attackers.

Assumption 2 seems to be conceptually close to deduction soundness. Traditionally, computational soundness and related notions hardcode the complexity-theoretic execution model, so we have to argue the equivalence in spirit. Deduction soundness says that the computational attacker is unlikely to produce a bitstring that can be parsed to a DY term that is undeducible based on the terms received so far. Indeed, any such bitstring would result from a computational trace that could not be described as a refinement of some (symbolic) trace from $L^{DY} \parallel_s^{\overset{\cdot}{\mapsto}_{LA}} A^{DY}$. Vice versa, any concrete trace from $L^{\mathbf{BIR}} \parallel_c A$ that is not an instance of a symbolic trace must either be due to an incorrect library implementation or due to A, in which case it constitutes an 'undeducible' bitstring. A formal argument would require a probabilistic notion of refinement, but constitutes an interesting pursuit.

This indicates that the commutation property may be an artifact of the translation approach, and not in fact necessary to achieve the aims of computational soundness (thus opening up the possibility of composition results like for deduction soundness). Roughly speaking, the commutation property states that no (concrete) PPT Turing machine can distinguish between the concrete (computational) protocol and a translation function around the DY interpretation of the protocol. The step using [115, Thm. 4.1] fulfills the same purpose, but there is no translation inside the system; instead, the instantiation is a meta-mathematical relation between the traces of the concrete system and the symbolic system. The difference becomes tangible when considering the proof effort. In a proof like [115, Thm. 4.1], the researcher is given a concrete trace and can provide a mapping on the spot, as long as they can justify the symbolic trace the mapping applies to. For the commutation property, the researcher has to provide a PPT algorithm that not only translate every single concrete trace, but is also reversible. We, therefore, think that this assumption merits deeper exploration.

In the following, we extend this proof to two parties (client and server) and an unbounded number of copies thereof.

6.2.4 Multi-Programs Proof

In this section, we elucidate our proof structure for the composition of multiple protocol participants, cryptographic libraries, and an unspecified attacker A. Consider the ARMv8 programs corresponding to the WireGuard initiator (I^{ARM}) and responder (R^{ARM}), along with their employed cryptographic libraries (${L_i}^{ARM}$ and ${L_r}^{ARM}$ respectively).

$$\mathfrak{T}^c((I^{ARM} \parallel_c {L_i}^{ARM}) \parallel_c (R^{ARM} \parallel_c {L_r}^{ARM}) \parallel_c A) \tag{6.1}$$

$$= \mathfrak{T}^c((I^{\mathbf{BIR}} \parallel_c {L_i}^{\mathbf{BIR}}) \parallel_c (R^{\mathbf{BIR}} \parallel_c {L_r}^{\mathbf{BIR}}) \parallel_c A) \tag{6.2}$$

By employing [116]'s lifter, we obtain corresponding **BIR** programs and demonstrate their composition with A using Theorem 3.6. Building upon the soundness of the symbolic execution engine [115, Thm. 4.1] and relying on an assumption about the attacker, as discussed in the previous subsection, we move from the concrete to the symbolic using the refinement theorem Theorem 3.4.

$$\sqsubseteq \mathfrak{T}^s(I^{\text{SBIR}} \parallel_s R^{\text{SBIR}} \parallel_s^{\vdash^{\text{bit}'}_{sA}} \underbrace{L_i{}^{DY} \parallel_s L_r{}^{DY}}_{\overset{\text{Theorem 3.3}}{=}} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY}) \tag{6.3}$$

$$= \mathfrak{T}^s(I^{\text{SBIR}} \parallel_s R^{\text{SBIR}} \parallel_s^{\vdash^{\text{bit}'}_{sA}} \quad L^{DY} \quad \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY}) \tag{6.4}$$

As $\parallel_s$ is associative w.r.t. trace equivalence, we can employ Theorem 3.3 to demonstrate the composition of $L_i{}^{DY}$ and $L_r{}^{DY}$ libraries—whether with identical or distinct function signatures—is equivalent to a single DY library (L^{DY}) encompassing all these function signatures. Subsequently, we apply our translation result from SBIR to Sapic$^-$ (Theorem 6.16), by leveraging Theorem 3.2 presented in Sec. 3.4.

$$\sqsubseteq \mathfrak{T}^s(I^{\text{SBIR}} \parallel_s^{\vdash^{\text{bit}'}_{sA}} R^{\text{Sapic}^-} \parallel_s^{\vdash^{(\!|\text{bit}'|\!)^{sp}}_{spA}} L^{DY} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY}) \tag{6.5}$$

We perform symbolic execution and extract the Sapic$^-$ model for each component individually.

$$\sqsubseteq \mathfrak{T}^s(I^{\text{Sapic}^-} \parallel_s R^{\text{Sapic}^-} \parallel_s^{\vdash^{(\!|\text{bit}'|\!)^{sp}}_{spA}} L^{DY} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY}) \tag{6.6}$$

$$= \mathfrak{T}^s(IR^{\text{Sapic}^-} \parallel_s^{\vdash^{(\!|\text{bit}'|\!)^{sp}}_{spA}} L^{DY} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY}) \quad \text{with } IR = \text{I} \mid \text{R} \tag{6.7}$$

As the DY attacker and library are included within the semantics of Sapic$^+$, we conclude that:

$$= \mathfrak{T}^s(IR^{\text{Sapic}^+}) \tag{6.8}$$

We have proved this end-to-end correctness result in HOL4, which you can see here.

6.2.4.1 Extending to Arbitrarily Many Parties

This argument can be repeatedly applied to cover an arbitrary but bounded number of protocol implementations. Depending on the language, the individual components may support open-ended loops, hence this bound is on the number of components, e.g., parties, not sessions. Let RIR = !I |!R.

$$\mathfrak{T}^c(\underbrace{(I^{ARM} \parallel_c L_i{}^{ARM})}_{n \text{ times}} \parallel_c \underbrace{(R^{ARM} \parallel_c L_r{}^{ARM})}_{n \text{ times}} \parallel_c A)$$

We inductively apply transformations as in the earlier steps (6.1) and (6.6). (Note each step is applied n times, then the next.)

$$\sqsubseteq \mathfrak{T}^s(\underbrace{I^{\text{SAPIC}^-}}_{n \text{ times}} \parallel_s \underbrace{R^{\text{SAPIC}^-}}_{n \text{ times}} \parallel_s^{\vdash^{\langle\!|\text{bit}'|\!\rangle^{sp}}_{spA}} L^{DY} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY})$$

Following step (6.7), we can draw the first initiator component and the first responder component together (I | R) over approximate.

$$\sqsubseteq \mathfrak{T}^s(\underbrace{I^{\text{SAPIC}^-}}_{n-1 \text{ times}} \parallel_s \underbrace{R^{\text{SAPIC}^-}}_{n-1 \text{ times}} \parallel_s^{\vdash^{\langle\!|\text{bit}'|\!\rangle^{sp}}_{spA}} L^{DY} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY} \parallel_s^{\vdash^{\langle\!|\text{bit}'|\!\rangle^{sp}}_{spA}} RIR^{\text{SAPIC}^-})$$

We can repeat this another $n-1$ times, as I | R |!I |!R is equivalent to RIR in SAPIC$^-$ and SAPIC$^+$.

$$= \mathfrak{T}^s(RIR^{\text{SAPIC}^-} \parallel_s^{\vdash^{\langle\!|\text{bit}'|\!\rangle^{sp}}_{spA}} L^{DY} \parallel_s^{\vdash^{\dot{\mapsto}}_{LA}} A^{DY})$$
$$= \mathfrak{T}^s(RIR^{\text{SAPIC}^+})$$

Chapter 7

Case Studies

The case studies in this chapter demonstrate the versatility and scalability of our framework across a spectrum of cryptographic protocol implementations. Starting from simple, illustrative examples, we progress to increasingly complex and security-critical systems: TinySSH, a streamlined SSH implementation; WireGuard, a modern VPN protocol; Basic Access Control (BAC) for e-passports with side-channel modeling; and WhatsApp Desktop, a large closed-source messaging application. Each case study highlights different strengths of our approach—handling of control-flow complexity, integration of observational models, and the ability to extract and verify models from real-world binaries without source code.

We have implemented CryptoBap on the HOL4 theorem prover [165] using its metalanguage SML. CryptoBap relies on HolBA's semantics-preserving transpiler and symbolic execution [116, 115]. We significantly extended the HolBA vanilla symbolic execution to handle cryptographic primitives, communication with the attacker, indirect jumps, and loops, which are essential to verify the security of protocols.

To demonstrate the real-world applications of our methodology, we have analyzed multiple case studies ranging from toy examples to real-world protocols, Basic Access Control (BAC) protocol used in e-passports, TinySSH, an implementation of SSH, WireGuard, a modern VPN protocol, and WhatsApp, the world's most widely used messaging application. All of our case studies demonstrate that our methodology does not introduce any artifacts that inhibit verification.

The binary of our case studies is unaltered; however, the verifier must manually initialize and steer the verification process. Specifically, the user is required to specify: (a) to the lifter, the code fragments to be analyzed, (b) to the symbolic execution engine, which operates on the output of the lifter, i.e., **BIR** code, the function names grouped as trusted (libraries) or untrusted (network), (c) to the symbolic execution engine, the symbolic model of the cryptographic functions, and (d) the assumptions regarding the cryptographic primitives and the security properties we proved for our case studies in the Sapic$^+$ and IML input files. For WhatsApp, we also identified target functions in

Protocol		ARM Loc	Verified Code Size	Feasible Path	Infeasible Path	IML Loc	CryptoVerif (CV) Loc	ProVerif (PV) Loc	Time (Second) IML	CV	PV	Verified in	Primitives
RPC		1.8K	0.659K	8	178	23	236	102	10	0.035	0.012	CV & PV	UF-CMA MAC
RPC-enc		53K	0.294K	28	348	53	313	118	9	0.073	0.047	CV & PV	IND-CPA INT-CTXT AE
CSur		0.7K	0.382K	11	237	29	277	177	6	0.656	0.035	flaw	IND-CCA2 PKE
NSL		2.8K	0.595K	23	455	56	296	204	35	2.740	0.052	flaw, verified	IND-CCA2 PKE
Simple MAC		1.5K	0.294K	13	149	29	207	101	8	0.047	0.033	CV & PV	UF-CMA MAC
Simple XOR		2.4K	0.100K	2	50	7	141	–	4	0.40	–	CV	XOR
TinySSH		18K	0.476K	136	1079	87	286	190	55	0.077	0.079	CV & PV	CRHF-CDH† & UF-CMA SIGN
WG	Initiator	27K	1.323K	68	1482	181	–	222	52	–	59.646	PV	DH-X25519‡ & ROM-hash⋆
	Responder			153	1389	464			47				

Table 7.1: IML case studies.

Protocol		ARM Loc	Verified Code Size	Feasible Path	Infeasible Path	Sapic⁻ Loc	Tamarin Loc	ProVerif Loc	Time (seconds) SBIR	Sapic⁻	TM	PV	Verified in	Primitives
TinySSH		18K	0.476K	136	1223	204	107	117	120	493	7.32	0.114	TM & PV	DHKA, SE, DS, HF
WG	Initiator	27K	1.323K	68	1482	260	150	181	60	67	1.28	13.266	TM & PV	DHKA, AEAD, HF
	Responder			153	1389	380			60	50				

Table 7.2: Sapic⁺ case studies.

WhatsApp Protocol	ARM Loc	symb. exec. paths feasible	infeasible	Sapic⁻ Loc all	simplified	crypto.	TM Loc	PV Loc	time to extract all	simplified	crypto.	Overall verif. time TM	PV
(1) Initiate Session	6844	12	25	106	61 (42% ↓)	33	161	171	13	14 (7% ↑)	12	19.76	0.057
(2) Respond to (1)	5803	106	413	718	92 (87% ↓)	12			65	60 (7% ↓)	38		
(3) Send Message	3041	156	368	983	429 (56% ↓)	166			71	100 (40% ↑)	65		
(4) Receive Message	4181	7293	19322	31257	2033 (93% ↓)	360			26903	40128(49% ↑)	25782		

Table 7.3: WhatsApp protocol analysis.

Protocols	ARM Loc	symb. exec. paths feasible	infeasible	Sapic⁻ Loc all	simplified	DeepSec Loc	time to extract all	simplified	Verif. time in DeepSec
BAC with $\mathcal{M}_{ct}$	176	5	69	242	83 (65% ↓)	205	9	7 (22% ↓)	1
BAC with $\mathcal{M}_{spec}$	194	15	170	630	240 (61% ↓)	589	22	18 (18% ↓)	7
Initiate Session with $\mathcal{M}_{ct}$	6844	12	65	283	90 (68% ↓)	220	43	44 (2% ↑)	2
Initiate Session with $\mathcal{M}_{spec}$	6899	120	679	2872	903 (68% ↓)	2191	148	130 (12% ↓)	5

Table 7.4: Case studies with observation models.

Ghidra, which took us a few days and standard reverse-engineering expertise.

Tables 7.1, 7.2, 7.4, and 7.3 shows data we have collected during our evaluation. The Loc of ARM assembly represents the complete assembly code, including cryptographic functions, which were necessary to consider in our preprocessing step to compute the control flow of the program required in loop summarization. Moreover, we report the runtime for preprocessing and symbolic execution (SBIR), construction of the symbolic tree plus model extraction (IML and Sapic$^-$), and for verification using CryptoVerif (CV), ProVerif (PV), Tamarin (TM) and DeepSec.

We also adapted the Csec-modex's pipeline to process CryptoBap-generated IML. Table 7.1 shows the list of protocol implementations from which we extracted their IML models and analyzed them with the Csec-modex toolchain. Primitives presented in Table 7.1 are standard, except Collision-resistant hash based on computational Diffie-Hellman (†), Curve25519 [110] (‡) and hash function in the Random Oracle Model [31] (⋆). The abbreviations used in Table 7.2 are WG (WireGuard), DHKA (Diffie-Hellman Key Agreement), SE (Symmetric Encryption), DS (Digital Signatures), HF (Hash Functions), and AEAD (Authenticated Encryption with Additional Data).

Table 7.4 provides statistics for BAC and WhatsApp case studies using the $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$ models. For the WhatsApp case study, Table 7.3 presents the data collected during the analysis of the components of the Sesame and double ratchet protocols.

7.1 Simple Case Studies

Apart from the XOR case study discussed in Sec. 5.3.1.1, we also verified other case studies of Csec-modex [9, 8] to evaluate CryptoBap. The only exception to [9, 8] is *smart metering protocol* which is not open source. For other cases, we obtained the same result. Additionally, in contrast to [8], which could not handle CSur, we successfully verified this case study.

RPC implements the MAC-based remote procedure call protocol [32]. We verified the client request and the server response authenticity under the MAC unforgeability assumption against chosen-message attacks with symbolic and computational guarantees. RPC-enc is an implementation of the RPC protocol that uses authenticated encryption. We also verified the secrecy of the payloads (which is not protected by the MAC-based RPC) with an assumption that authenticated encryption is indistinguishable against the chosen-plaintext attack and provides ciphertext integrity. CSur is the Needham-Schroeder public-key authentication protocol [135]. We verified the secrecy and authentication properties for the CSur binary. Our analysis confirmed that CSur is vulnerable to attack in [121] and leaks protocol parties' nonces. Similar to [9], we also removed the assumption (i.e., all cryptographic material plus nonce are tagged) used in [8] for the Needham-Schroeder-Lowe (NSL) case study to obtain the computational

soundness result. We confirmed the flaw in the protocol discovered in [9]: if the nonce of the second party is not tagged and is sent separately, it can be (mis)used as the first protocol message.

Simple MAC implements the first half of the RPC protocol in which a single payload is concatenated with its MAC [32]. We verified the payload authenticity under the unforgeability of MAC against the chosen-message attack assumption. For simple XOR case study, we verified the secrecy of the payload with CryptoVerif. We did not attempt verification with ProVerif, as the analysis of theories with XOR requires extra effort [108], while CryptoVerif's attacker model is strictly stronger.

7.2 TinySSH

TinySSH is a minimalistic SSH server that implements a subset of SSHv2 features and ships with its own cryptographic library. To formulate authentication properties, which ought to hold for any communication partner to the TinySSH server that conforms to the SSH protocol specification, we modeled the client side of the SSH protocol in IML; agents at the other end of the communication line are manually developed. We manually specified the cryptographic assumptions about the primitives used by the TinySSH implementation in CryptoVerif and ProVerif templates. We verified mutual authentication with ProVerif and CryptoVerif.

We also automatically extracted the Sapic$^-$ model of TinySSH from its ARMv8 machine code and we used Sapic$^-$ to manually model the client of the SSH protocol. Using ProVerif and Tamarin, we verified mutual authentication and forward secrecy properties for TinySSH.

7.3 WireGuard

WireGuard implements virtual private networks akin to IPSec and OpenVPN. It is quite recent and was incorporated into the Linux kernel (stable) in March 2020. We have automatically extracted, *for the first time*, the WireGuard's IML and Sapic$^-$ models from its linux implementation binary—all other existing models are hand-written and derived from the specification [75, 117, 101]. Existing WireGuard's symbolic models often utilize pattern matching to verify authentication. This involves constructing the entire message from the initial stage and subsequently comparing it to the received message from the network, which requires further justification. In contrast, our extracted models from the implementation closely represents the actual behavior of WireGuard, as they deconstruct the network message using ChaCha20Poly1305 Decryption for the purpose of authenticating ChaCha20Poly1305 Aauthenticated Encryption with Associated Data (AEAD). As a result, our models obviate the need for additional caution

when verifying the authentication property. We model the handshake and first transport message, after which the key exchange is concluded, and prove the protocol participants mutually agree on the resulting keys in both ProVerif and Tamarin.

TinySSH and WireGuard are case studies with indirect jumps in their binary, and TinySSH is the only one that features multiple sessions. The case studies from Csecmodex [9, 8] do not contain loops as [9, 8] did not support them. We handle the loops inside the implementation of TinySSH using the summarization technique explained in Sec. 5.2.1, which is automated by CryptoBap—these loops handle, e.g., reading key files of variable size. TinySSH spawns a new server for each incoming TCP connection by using `inetd`, which we correctly cover in our template files. We hence handle multiple sessions, including potential replay attacks, correctly.

Moreover, we employed the combined deduction relations $\vdash^{\mathsf{bit}'}_{sA}$ and $\vdash^{(\!|\mathsf{bit}'|\!)^{sp}}_{spA}$ (defined in Sec. 6.2.1) in our TinySSH and WireGuard case studies to demonstrate the application of these combined deduction relations. As we extracted formal models of TinySSH and WireGuard from their respective implementations, we have identified no instances in which the DY attacker could acquire additional knowledge through the use of these combined deduction relations.

7.4 Basic Access Control with Observational Models

Many countries are adopting electronic biometric passports, a.k.a. e-passports. These passports encode the holder's digital information within a Radio Frequency Identification (RFID) chip for interaction with passport readers. However, e-passports face several threats related to security and privacy, including skimming, cloning, and eavesdropping [93, 14]. To address these concerns, the International Civil Aviation Organization (ICAO) standardized security mechanisms, among those, the Basic Access Control (BAC) protocol. BAC's goal is to ensure that only authorized parties can access personal information stored in passports' RFID. A party (the reader) is authorized if they know a machine-readable code printed on each e-passport, which the passport holder typically provides by physically handing the passport to the reader or scanning it themselves (if the reader is a device).

Despite the ICAO standard's goal to create an unlinkable BAC protocol, some implementations of the protocol permit reidentification of a passport holder. Arapinis et al. [11] found that these implementations output different error messages if a replayed message is invalid due to an incorrect MAC or an incorrect nonce. As the second case can only occur when the message is replayed to the same passport (because each interaction uses a fresh nonce), while the first can only occur if the message is replayed to a different passport (as the MAC key is fixed per passport), this allows for identification of the passport holder. Subsequently, the error messages were unified, but Chothia

and Smirnov [62] find that all e-passports are still vulnerable to a variant of the attack where instead of the error message, the attacker measures the computation time of the passport's response to distinguish the incorrect-MAC branch and the incorrect-nonce branch.

To showcase how our methodology enables automated verification against side-channel attacks on security-critical systems, we implemented the BAC protocol ourselves. While the machine code is not easily extractable from RFID chips for researchers (and the legal implications are unclear), standardization bodies could request access and run similar analyses. As depicted in Fig. 4.8, our implementation does not allow distinction by error message, but rather by side-channel leakage. Using our methodology, we find that an attacker can use this leakage to distinguish both types of failures and thus break unlinkability. The implementation is vulnerable under both $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$, see Table 7.4.

7.5 WhatsApp

WhatsApp allows users to exchange messages, share status posts, and make audio and video calls. Since 2016 [173] WhatsApp uses a modified version of the Signal protocol, developed by Open Whisper Systems, as the basis for end-to-end encryption. This encryption protocol prevents third parties and WhatsApp's servers from accessing the plaintext of user messages or calls. Messages are encrypted with ephemeral cryptographic keys that are regularly updated (using ratcheting or a new handshake), preventing attackers from decrypting previously transmitted messages, even if the current encryption keys are compromised.

We extracted, *for the first time*, a formal model of WhatsApp's binary to verify forward secrecy and post-compromise security properties. To this end, we reverse-engineered the WhatsApp Desktop's binary using Ghidra. Fortunately, the WhatsApp Desktop application did not strip the function symbols from the binary, unlike the iOS or Android versions. This allowed us to match the function names with the latest version of the `libsignal`, Signal's protocol implementation.

7.5.1 WhatsApp Components

The double ratchet protocol enables secure (confident and authentic) message exchange between the two parties. It builds on the Extended Triple Diffie-Hellman (X3DH) key agreement protocol [96], which allows these two parties to establish a shared secret key using mutual authentication based on their public keys.

Initially, the key is obtained via X3DH, and called the *root key*. Subsequent ephemeral keys are generated using ratcheting, so called because earlier ephemerals cannot be de-

rived from later ones. Ratcheting is *asymmetric* if the parties switch roles (i.e., from sender to receiver or vice versa) and *symmetric* if they maintain the same role. Symmetric ratcheting provides *forward secrecy*: even if long-term secrets (the secret keys to the public keys used in the handshake) or future ephemerals are revealed, past ephemerals and thus the messages sent with them remain secret. Asymmetric ratcheting establishes fresh ephemerals that are unknown to the attacker even if *past* ephemerals are known unless the attacker actively attacks the handshake. This provides *post-compromise* security: even after ephemeral keys are revealed, the authenticity and confidentiality of future ephemerals (and the messages encrypted with them) can be recovered. Ephemeral keys are used to derive the encryption and MAC keys that ensure each message's confidentiality and security. This provides cryptographic deniability because any party that knows a message must know the ephemeral key used to send it, and that key would allow the party trying to prove the authenticity of the message to fake any other message's MAC.

The Sesame protocol defines the session management and operates on the layer above the double ratchet protocol. It manages multiple double ratchet sessions between the different devices associated with each user accounts (e.g., if Alex has n_A devices and Blake n_B devices, on each of Alex's devices, Sesame manages n_B connections to Blake and n_A-1 connections to Alex's other devices). Sesame manages the creation, deletion, and use of sessions, maintaining a Local database that records each party's devices and their associated sessions. To establish a pairwise encrypted conversation between two users, Sesame manages an instance of the double ratchet protocol (including X3DH key agreement) between each of their devices. In addition, sent messages are forwarded to a party's other devices. Signal uses the same double ratchet protocol to this end, whereas WhatsApp uses a variant of the Signal protocol along with a proprietary multi-device architecture. In WhatsApp, each message is encrypted separately for each recipient device using a pairwise session, and WhatsApp's multi-device system ensures messages and other data stay automatically synced across a user's devices. In contrast, Signal operates with separate double ratchet sessions exclusively between sender and recipient devices. It treats each device as an independent endpoint and avoids seamless synchronization to protect privacy. Signal uses ephemeral keys and a decentralized architecture to minimize trust on the server. By contrast, WhatsApp focuses on usability and reliable multi-device synchronization. Another key difference concerns session lifetime and makes WhatsApp more vulnerable to clone attacks [178, 129, 68]. WhatsApp clients are not required to initiate new sessions once one has been established unless the session state is lost, which may occur due to the reinstallation of the application or device change. For Signal, in contrast, new sessions can be established because enough time has passed, enough messages have been sent, because the session lists become too large, and other reasons [68].

7.5.2 WhatsApp Model

Like the handwritten Tamarin model for Signal in [68], we abstract the X3DH key agreement protocol with a private channel that 'magically' communicates a master secret key (initial root key) between two parties. This allows us to focus on the Sesame and double ratcheting layers, while avoiding verification issues due to the Diffie-Hellman (DH) theory that is necessary to verify the X3DH handshake. ProVerif and DeepSec do not have dedicated support for DH theories and only allow for a coarse approximation using custom equations (that need to be linear or subterm convergent, see discussion [60, Sec. 2.3 and 2.4]). Tamarin has a richer DH theory, but does not permit addition in the exponent and can suffer from slowdowns in unification. As unification in a full DH theory is undecidable [76, 123] (and all those tools build on unification), full DH requires fundamental advancements. While a proof of the X3DH handshake in Tamarin's limited theory might be possible, we considered that work orthogonal to our goal of finding implementation-level flaws and wait for future improvements of our verification backends.

Excluding X3DH, we extract the model of four main components that are responsible for

1. *initiating a new session* in which the client requests the recipient's keys from the WhatsApp server,
2. *responding to a request for initiating a new session* by the recipient and calculating the corresponding session key,
3. *transmitting messages* in a session (symmetric ratchet), and
4. *receiving messages* within a session (asymmetric ratchet).

Table 7.3 presents the data obtained from our evaluation of the WhatsApp components using Tamarin and ProVerif. By considering all memory operations and function calls (abstracting the cryptographic calls), we obtain a Sapic$^-$ model of size nearly 33,000 Loc. The table includes, the size of the code under analysis, the explored symbolic paths, the extracted models—complete and simplified, consisting solely of cryptographic operations, and translated into Tamarin and ProVerif—along with the duration (in seconds) of each step.

Simplification, explained in Sec.5.3.4, is necessary, and reduces the extracted model to roughly 2,600 Loc. Despite this optimization and the high degree of maturity of Tamarin and ProVerif, both tools still struggle with models of this size. We, therefore, removed all observations (w.l.o.g., as none of them contained names, i.e., cryptographic values) and over-approximated all memory operations with variables (i.e., without loss of attacks). Even with these adjustments, we are unaware of any prior work attempting to verify WhatsApp at the same level of generality as achieved in this study.

7.5.3 Gaps Between Specification and Implementation

Our analysis showed that, in certain scenarios, the protocol implementation behaves differently from its specification in the WhatsApp security white-paper [173]. These scenarios concern the component responsible for decrypting incoming messages (component 4). The whitepaper specifies that the chain and root keys are updated upon receiving a response, yet it lacks a sufficient explanation of the ratcheting mechanism [173, p.15]. According to WhatsApp, the ratcheting process involves each party calculating its next chain and root keys using a shared ephemeral secret, derived from the sender's and receiver's ephemeral keys, and a root key whose origin is not clearly defined. Instead, the extracted model indicates three distinct behaviors of ratcheting as the next chain key is derived from: (a) the current chain key, (b) the new chain key, calculated from fresh root and ephemeral keys, and (c) the fresh chain key, calculated from the current root key and a new ephemeral key.

This highlights the strength of model extraction. Most protocol models have to rely on the specification to be correct, even if the source is available, a thorough comparison is tedious and requires tool support (such as model extraction for the source language). Often, behaviors are underspecified (as in this case), and the modeler fills the gap—here, source-code inspection is helpful, but for WhatsApp, this is not an option. Ultimately, formal analysis results rely on the adequacy of the model. Gaps through underspecification or mispecification can lead to overlooking attacks.

7.5.4 Authenticity with Tamarin and ProVerif

We used our extracted model to analyze WhatsApp's session management and double ratchet protocol with Tamarin and ProVerif. Utilizing these tools, we verify the security properties of our extracted model by defining queries and checking the reachability of all events in the model. Our analysis shows that Tamarin and ProVerif successfully verify forward secrecy for two parties initiating a session and exchanging two secret messages. However, our analysis revealed a major issue suspected due to prior work [68], which identified it in the related Signal protocol. *WhatsApp compromises post-compromise security despite employing the double ratchet protocol when faced with a clone attacker.* Cremers et al. [68] have proposed secure mechanisms that offer stronger guarantees. However, our analysis showed that WhatsApp does not implement any of the proposals, allowing a clone attacker to break post-compromise security.

Note, while the analysis of Signal's implementation benefits from direct access to the source code (the model in [68] was manually crafted but informed by the code), our models were derived entirely from the binary. Moreover, while ProVerif automatically identifies the post-compromise security violation for WhatsApp, we employed

heuristics to guide Tamarin's proof search and manually selected 303 proof steps from the available options.

7.5.5 Privacy Properties with DeepSec

In recent years, hardware-induced attacks [118, 103] have emerged as a game changer in computer security. They are no longer just theoretical attack vectors but practical threats that can leak secrets from systems previously considered or even verified to be secure. Despite this, unfortunately, almost none of the widely used cryptographic protocols are analyzed against this class of attacks. Using this thesis's methodology, we analyze the WhatsApp application and show how vulnerable it is to side-channel attacks.

We start by annotating the **BIR** translation of the session initialization component (component 1) using both the constant time $\mathcal{M}_{ct}$ and the speculation $\mathcal{M}_{spec}$ models. As we found an attack on this component, we did not investigate the composition with the other components, as we would find the same attack again. For each of the two resulting annotated **BIR** programs, we perform symbolic execution, extract its Sapic$^-$ model and translate into DeepSec. The evaluation results for session initialization (component 1) with DeepSec are listed in Table 7.4.

DeepSec verifies trace equivalence (and other process equivalences), which is typically used to encode privacy properties like unlinkability, strong secrecy, voting privacy and more [61]. One of the relevant privacy properties of WhatsApp is *unlinkability*, which indicates that an attacker cannot distinguish between two sessions—one involving protocol parties Alice and Bob, and another with different parties, Alice and Charlie. The unlinkability property is maintained for both extracted models, specifically in relation to the constant-time model and the speculation model. When considering session establishment, we define a more refined version of unlinkability that should also ensure that an attacker cannot tell whether a user is initiating a new session to start a new conversation or establishing a new session for an existing one. However, our analysis revealed a vulnerability: *the attacker can indeed distinguish these cases by monitoring the state of the microarchitectural components, namely instruction cache.*

It is worth noting that we assume the WhatsApp server is honest in our models and model the communication between WhatsApp clients and the server—which, in reality, is secured using the Noise protocol framework [173, p. 35]—as private channels.

7.5.6 Discovery of Privacy Attack

Our core discovery is a privacy violation in the WhatsApp Desktop application, where an attacker can determine whether a victim (i.e., a WhatsApp account holder) is initiating a conversation with a third party for the first time. When a victim attempts to

establish a pairwise encrypted session with the recipient, they retrieve the recipient's *pre-key bundle* from the WhatsApp server. The pre-key bundle includes the public identity key, the public signed pre-key, and a single public one-time pre-key (OTPK) of the recipient [173]. According to the WhatsApp security whitepaper [173], the recipient's public OTPK is used only once and is removed from the WhatsApp server storage after it has been requested to initiate the first-time conversation. Therefore, whether or not the recipient's public OTPK is present when the victim establishes a new pairwise encrypted session with the recipient distinguishes a first-time conversation from a new session for an existing conversation.

This attack applies to one-to-one conversations as well as to group chats, which makes it more severe. Using our attack, an attacker can create a group with the victim and add any chosen third party, then check if the victim and the selected third party have been in contact. By repeating this process and inviting more third parties to the group chat, the attacker can reconstruct the victim's social graph—that is, the set of individuals with whom the victim communicates. When an attacker reconstructs the victim's social graph, sensitive details about the victim's personal life, including their relationships, interests, and activities, are revealed.

We analyzed the **BIR** program for the WhatsApp component's assembly code, which initiates a secure session by retrieving and processing the recipient's pre-key bundle. By annotating this **BIR** program with observational models and extracting the respective models, we were able to analyze these models using DeepSec (as detailed in table 7.4). Our analysis revealed a condition tied to the recipient's public OTPK. As a result, an attacker with sufficient access to the victim's system can determine if the recipient's public OTPK exists by observing WhatsApp's control flow.

Impact. The vulnerabilities we have found, can have real-life impact. Imagine that V is a journalist who is connected with numerous sources, including whistle-blower T. The attacker R is a regime that enforces its citizens to install a certain application which allows it to mount instruction cache attacks. The regime infiltrates some of V's WhatsApp groups unrelated to their journalistic work. R convinces a moderator in this group to include T in a chat discussion. Using our attack, R confirms that V was in contact with T. This is plausible, e.g., Russia [34], Kasachstan [79], Saudi-Arabia [2] and Singapore [159] provide certain services like visa or passport requests, tax declaration or social services exclusively via government-provided Smartphone Apps. In other countries, like Austria, Estonia or Germany digitalized services with a strong App focus increase the pressure to install such Apps, often in connection to digital ID [33]. Some states in the U.S. also increase the pressure to install Apps for social services[143]. Many more countries had indirect enforcement of the use of Apps during the Covid-Pandemic, e.g., to access public buildings or for citizens that tested positive [166].

7.5.6.1 Attack Vector: Instruction Cache

In our model, the above attack requires the side channel to reveal the program's control flow to learn the presence or absence of the recipient's public one-time pre-key. Both observation models represent a realistic attack vectors [54]; we will now outline how this attack can be realized as an *instruction cache attack*. Such attacks exploit the timing differences in the processor instruction cache behavior to infer which program paths are executed without requiring direct access to the program counter or memory contents.

WhatsApp Desktop's assembly code includes a conditional branch that checks for the existence of the one-time pre-key in the pre-key bundle during session establishment. When the one-time pre-key is present (indicating a first-time conversation), a specific set of instructions is executed to process it, whereas a different program path is taken if the key is absent (indicating an existing conversation). These two distinct execution paths result in different patterns of instruction cache usage. In our proof-of-concept attack implementation, we used the Prime+Probe technique [146] to monitor the instruction cache by co-locating a process on the same CPU core as the WhatsApp Desktop application. Interested reader can find out the details of our PoC in the following.

In a Prime+Probe attack, the attacker first 'primes' the instruction cache by executing their own code to fill specific cache sets. Next, the attacker triggers the victim to establish a new encrypted session, e.g., by initiating a group chat with the victim and inviting the third person. Then, the attacker 'probes' the cache by measuring the time it takes to re-execute their code. If the victim's execution of the pre-key processing code evicts the attacker's instructions from the cache (due to cache set conflicts), the probe will take longer, revealing that the 'first-time conversation' path was taken. Conversely, if the cache remains largely untouched, it indicates the 'existing conversation' path was executed. By mapping the WhatsApp application's instruction addresses to cache sets, the attacker can reliably distinguish between these two cases. The instruction cache attack eliminates the need to observe the program counter in the clear or have privileged access to the victim's system memory.

To evaluate the feasibility of our attack, we implemented a proof-of-concept targeting the WhatsApp Desktop application. This implementation is based on the modified Flush+Flush framework designed for macOS, available here. Experiments were conducted on a 2019 MacBook Pro equipped with an Intel Core i7-9750H processor (6 cores, 2.6 GHz) and 16 GB of RAM, running macOS Sonoma Version 14.1.2.

The attack leverages a Prime+Probe instruction cache side channel to infer whether the victim initiates a new conversation, specifically, whether a secure session is being established. The adversary controls a co-resident user-space process scheduled on the same physical core as the target application. We assume the attacker knows the

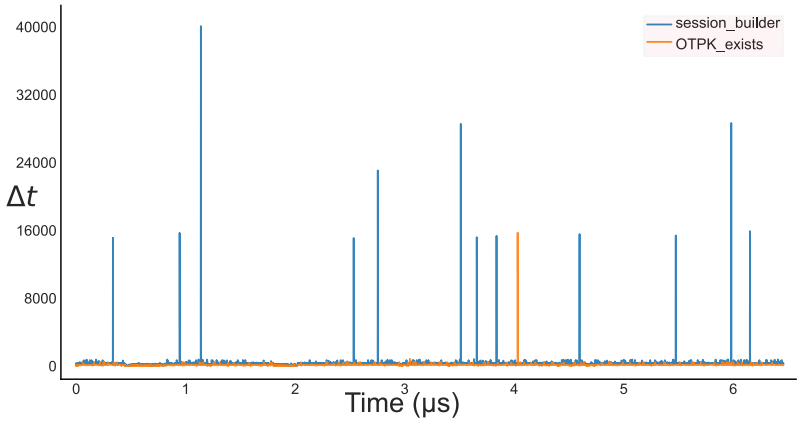


Figure 7.1: The observed access latency (Δt) during the designated time period.

virtual addresses of two functions within the WhatsApp binary: one responsible for initiating a secure session, and another triggered when a one-time pre-key is present (*session_builder* and *OTPK_exists* in Fig. 7.1, respectively).

In each attack cycle, the attacker primes the instruction cache with knowledge of these addresses to access another code that maps to these cache sets and lines. After a short delay to allow for potential victim execution, the attacker re-accesses the same addresses and measures the access latency (Δt). Elevated latency indicates eviction, implying that the victim executed instructions mapping to the same cache line. Conversely, low latency suggests the cache lines remained untouched.

The macOS-compatible iteration of the Flush+Flush framework integrates timing-based cache probing by utilizing the *mach_absolute_time(.)* function to measure time intervals. This function offers time measurements in platform-dependent units and, on macOS, produces a high-resolution timestamp akin to a cycle count. It is employed for precise timing evaluations of access latency. Our implementation employs a straightforward access timing technique to infer potential cache evictions, thus enhancing the accuracy of cache-related timing calculations.

Fig. 7.1 presents a time series plot of the access latency Δt. The x-axis represents time (in microseconds), while the y-axis denotes measured latency. *session_builder* represents the address of the function that initiates a secure session, while *OTPK_exists* indicates the address of an instruction executed when a one-time pre-key is used in the creation of the master secret key for the secure session. Distinct latency spikes correspond to the victim executing one or both target code paths, revealing instances of first-time conversation establishment (highlighted in red in Fig. 7.1).

Responsible disclosure. The vulnerabilities found in WhatsApp Desktop were responsibly disclosed to Meta by the authors in March 2025. Meta has confirmed that the application is vulnerable to these identified attacks.

Chapter 8

Related Work

In the last decade, cryptographers started to employ and even develop theorem provers to develop verifiable proofs [92]. This started with the CertiCrypt framework for Coq [28], which subsequently developed into EasyCrypt [27]. These tools support reasoning about probabilistic programs and classes of (e.g., poly-time restricted) adversaries via probabilistic and probabilistic relational Hoare logic. There are also embeddings of probabilistic reasoning like FCF [147], Verypto [15] and CryptHOL [119], which are easier to combine with other techniques (this was demonstrated [35, 147] for C code), but they require tedious manual analysis and a deep understanding of the underlying relational probabilistic logic.

These methods are typically used to verify cryptographic constructions. By contrast, complex protocols are analyzed using symbolic models, where cryptographic primitives like encryption, signatures, etc. are abstracted using a term algebra and a set of reduction rules. This makes the analysis of protocols that use such primitives amenable to automation, e.g., using first-order SAT solving [83], Horn clause resolution [45] or constraint solving [152]. To great success: within a decade, protocol verification tools went from analyzing small academic protocols [47, 107] to fully-fledged TLS models [67, 38].

This degree of automation comes at the cost of abstraction, both in terms of the computation environment and the cryptographic primitives. The former is owed to the focus on protocol *specifications* rather than implementations. This makes sense, because often, the task at hand is to evaluate designs or standards rather than specific implementations, which can be incomplete or nonexistent. There are efforts to translate implementations into the protocol specifications, but they are limited to high-level languages such as *C* [8]. The same holds for verification tools that operate at the source-code level [77, 109].

Verified cryptographic Protocols' Implementation. The application of formal methods to analyze cryptographic protocols and verify properties like secrecy and authen-

Table 8.1: Selected model extraction approaches; Symb = Symbolic, Comp = Computational, Auth = Authentication, JS = JavaScript.

Papers	Language	Abstract Model	Model Type	Property	Soundness
Aiazatulin et al.[9, 8]	C	Applied-pi	Symb. + Comp.	Secrecy, Auth.	✓
Chaki et al.[59]	C	ASPIER	Symb.	Secrecy, Auth.	✓
Goubault-Larrecq et al.[85]	C	Horn clauses	Symb.	Secrecy, Info. flow	✓
Backes et al.[18]	F#	RFC	Symb. + Comp.	Safety properties	✓
Bhargavan et al.[43, 40]	F#	Pi/CryptoVerif	Symb. + Comp.	Secrecy, Auth.	✗
Jürjens[98]	Java	First-order Logic	Symb.	Secrecy, Auth.	✗
HE et al.[94]	Python-JS	Applied-pi	Symb.	Secrecy, Auth.	✗
This work	Binary	Applied-pi	Symb. + Comp.	Secrecy, Auth.	✓

tication dates back to Lowe's work [121]. Traditionally, domain experts translate protocol specifications into formal models to analyze high-level behavior using theorem proving [144, 30], model checking [121, 29], and symbolic analysis [48, 60, 126]. Tools like ProVerif [48] and Tamarin [126] are proven effective in verifying security properties. However, these specification-based models abstract away implementation details, potentially overlooking vulnerabilities introduced during execution, such as hardware-induced leakages [118, 103] or compiler bugs [156, 158].

To address discrepancies between specifications and implementations, techniques like fuzz testing [51, 10], differential testing [125], symbolic execution [7], code generation [57, 3], deductive verification [57, 111, 13], and type checking [42, 44] have been employed, which directly target the implementation of protocols. Model extraction techniques further aim to verify implementations in high-level languages like *F#* [41, 18, 38], Java [139, 98], C [85, 59, 7], and Rust [111]. Yet, the complexity of these languages limits their practicality. Table 8.1 compares selected works that verify the security of protocols via model extraction in some form.

There are also works that recover formats of protocol messages at the binary level [55, 113, 175, 176]. However, their intent is different from ours. Existing approaches are mostly applied to malware binaries and use heuristics to gain insights into their operation. Thus, they are not concerned with soundness, and the inferred message formats can be wrong.

CryptoBap addresses these gaps by extracting formal models directly from protocol binaries, avoiding reliance on specifications or high-level code. We integrate microarchitectural leakage models into protocol formal models. This enables us to utilize the extracted models to assess (a) whether protocol properties are preserved in the presence of hardware leakages and (b) the existence of microarchitectural leakages through the protocol verifier DeepSec. Moreover, we extract the memory operation models into Sapic$^+$ and apply simplification rules, enabling the extracted models to be analyzed by protocol verifiers.

Side-Channel Analysis of Cryptographic Libraries. In recent years, side-channel detection, mitigation, and formal analysis have become active areas of research, with numerous papers published on the topic. Here, we do not intend to survey all works in this domain (an incomplete list can be found [120, 58]); rather, we focus on a few selected works that are closer to our approach.

Leaks, such as timing information, have been exploited by side-channel attacks to extract secret information from cryptographic implementations. These vulnerabilities have been analyzed using formal methods, both static and dynamic. Various works, including MaskVerif [25] and methodologies aimed at resilience against these leakages, strive to maintain strong security properties for cryptographic implementations, even in the presence of side-channel attacks. Furthermore, symbolic execution techniques [99] have been employed to uncover potential side-channel vulnerabilities in cryptographic implementations.

Cryptographic libraries have been the prime target of side-channel attacks, e.g., timing attacks. The constant-time (CT) paradigm mitigates these by ensuring control flow and memory accesses are independent of secret data under sequential execution [36]. High-assurance frameworks like Jasmin [141] and FaCT [154] use formal methods, such as information-flow type systems, to enforce CT and produce verified, efficient implementations.

However, Spectre attacks [102], exploiting speculative execution, challenge the CT paradigm by leaking secrets even in CT-compliant code. Research has extended CT to speculative constant-time (SCT) to protect against Spectre variants. Shivakumar et al. [155] introduced a type system in Jasmin for SCT against Spectre-v1 with minimal overhead using selective speculative load hardening. They have also addressed declassification issues under speculation, proposing relative non-interference (RNI) and efficient countermeasures. Arranz Olmos et al. [141] extended this to all Spectre variants, including Spectre-RSB, with low overheads. Other approaches like Blade [169], Swivel [134], and Serberus [131] offer protections but may rely on hardware or incur higher overheads.

However, to the best of our knowledge, previous studies have not analyzed protocol implementations in the context of hardware leakages. This oversight neglects the leakages that can arise from the interaction between cryptographic functions and applications, namely, protocol implementations, which can also disclose secret information. In this work, we propose a methodology designed to identify such leakages and address this gap.

Comparison to Csec-modex Toolchain. Closely related to CryptoBap is Aizatulin's work [8, 9]. Analogous to our work, they also proved the soundness of their approach, i.e., showed that all attacks present in the *C* code are preserved in the extracted `IML` models. The main difference between the two approaches is the fact that

we target protocols' binary. Moreover, compared to their approach, which can handle only a single execution path, CryptoBap handles *all execution paths* of protocols, including those that contain *conditionals*, *bounded loops*, and *indirect jumps*. More specifically, Aizatulin's approach restricts the input programs to programs that have no "else" branches and no loops. Conceptually, one may see this as irrelevant because one might speculate that all paths outside the "main" part are not useful, i.e., they do not produce network output, or at least none that reveals cryptographic information (e.g., error codes). But this is still restrictive. First, many protocols are specified to produce network output in error cases, for example, *decoy messages* in anonymity protocols or error messages that are encrypted. Moreover, seemingly regular protocols have "else" branches as part of their "main" message flow, once we look at them with the level of detail necessary to analyze implementations. For example, cipher suite negotiation in TLS must be formulated with multiple branches depending on the input message.

Also, their translation from *C* to *C virtual machine* (CVM)—the intermediate language used for extracting the IML model of protocols—is not verified. This renders relating the verified properties to the *C* implementation infeasible. Similarly, they abstracted cryptographic libraries, attacker calls, and random number generation in their verification. However, they do not formulate the requirements on the program under the analysis explicitly. CVM already assumed to include primitives for library calls, attacker calls, and random number generation—which are not primitives of the *C* language. In this regard, CVM is fairly close to IML. Moreover, our translation into IML is entirely different (a) because the input language has a more complex state and interaction with the other IML entities and (b) because in contrast to Aizatulin's approach, we handle conditionals, hence our symbolic execution is not only a sequence of actions. Finally, while they had to change the source code of protocols and add dummy functions to flag the occurrence of events, CryptoBap automates releasing events during symbolic execution and minimizes user involvement.

Our method is inspired by and builds on a line of work by Aizatulin et al. [9, 8]; Fig. 8.1 highlights differences between the two approaches. We adopt their process calculus *intermediate model language* (IML) and translation from IML to ProVerif and CryptoVerif. However, we extract the IML model from the machine code of protocols, providing more reliable security guarantees that are independent of the compiler used to generate the object code of protocols. We demonstrated the effectiveness of our approach by covering the case studies of Aizatulin et al. [9], including the CSur case study which they could not verify due to limitations of their framework in handling network messages as *C* structures.

Both Csec-modex and CryptoBap model extractions to IML approaches build upon computational soundness, which imposes stringent requirements on the use of cryptography and protocols. Computational soundness is incredibly difficult to prove mechanically [119], which was the main motivation for our general framework (see Part I),

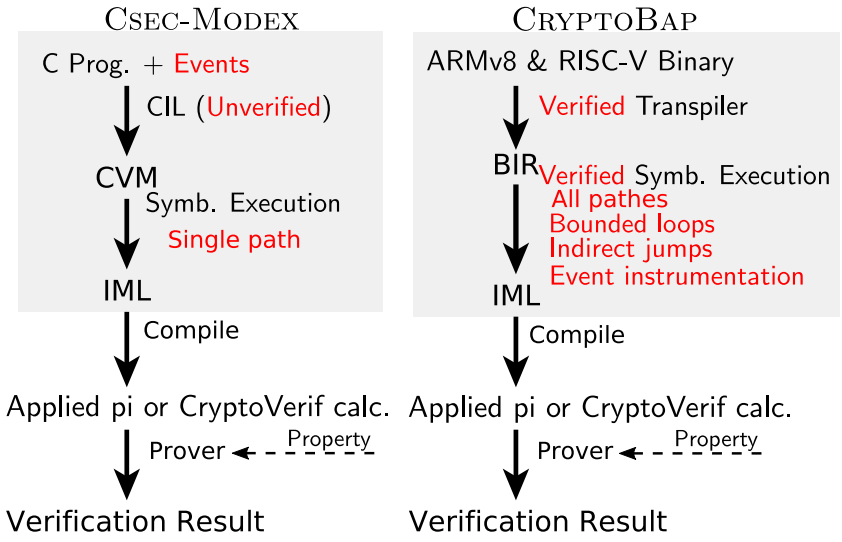


Figure 8.1: CRYPTOBAP VS. CSEC-MODEX

as it (a) enables us to avoid the detour via computational soundness and (b) enables compositional proofs. Where both CSEC-MODEX and ours explained in Sec. 6.1 rely on pen-and-paper proofs in the cryptographic model, we have a mechanized end-to-end symbolic proof as showed in Sec. 6.2.

Separation Logic. The line of works [153, 105, 142, 161, 13, 12] used separation logic to analyze the implementation of security protocols. Sprenger et al. [161] introduced a methodology where a protocol model is first formalized in Isabelle/HOL [145] and then translated into I/O specifications, which are verified using separation-logic based verifiers. Arquint et al. [13] extended this to the TAMARIN verifier to enable verification against TAMARIN's models. Follow-up work [12] stepped away from verifying against the specification, and directly verified the protocol properties, which are *stable under concurrency*, by building on a programming language that incorporates protocol operations and modeling the attacker in that language.

Among those [161, 13, 12] used verifiers like Nagini [80] for Python, Gobra [174] for Go, and VeriFast [97] for Java and C. Nonetheless, the soundness of these approaches depends on the correctness of utilized verifiers, including their dependencies (e.g., Nagini relies on Viper [133]). In theory, they could be proven sound, but the languages are not ideal for formalized results. By contrast, **BIR**'s decompilation approach already provides a formalized soundness results from lifting to symbolic execution while covering machine code produced by compiling these languages.

Throughout this line of work [161, 13, 12], a translation function maps between byte-level messages and DY terms (or an injective function in the other direction). Therefore, our arguments in Sec. 3.1 apply. However, a cursory glance suggests that

our general framework (i.e., the subject of Part I) might help with this, as their proof structure likewise consists of a refinement step, followed by decomposition and translation to a verification language and their communication model builds on a (subclass of) LTS and CSP-style parallel composition (with built-in translation).

Model Validation. Several formalisms were proposed for modeling distributed systems [52, 128, 164, 91], including a hierarchical modeling language and the hybrid process calculus [52] that focuses on bisimulation notions and congruence results w.r.t. parallel composition. Strubbe et al. [164] introduced a technique to deal with the nondeterminism in distributed systems, which was later extended by Meseguer et al. [128] to handle the asynchrony of communications.

The methodologies in [128, 164, 91] required checking cross-system variable consistency during communication due to shared variables. This direct impact of one component's actions on others poses a challenge. In contrast, our synchronization method, introduced in Sec. 3.2, relies on events containing symbols. By avoiding the reuse of symbols, cross-system consistency is not a concern for us. This improves our approach's efficacy and makes it ideal for modeling distributed systems.

DY Code Analysis. Similar to some of our case studies, DY^* [37] permits code analysis w.r.t. a DY attacker, but for a high-level language (F^*) that allows conducting proofs using dependent types. Their DY attacker is formulated within F^*, whereas our framework results apply to a DY attacker that may compose with other languages. Proofs in their framework are internal to F^*, while we depend on the correctness of the protocol verifiers. [37, Sec. 1] discusses the trade-off in automation versus modularity. DY^* is complemented by Comparse [170] which provides type combiners and lemmas to deal with packing and unpacking. These lemmas are proven at the bit-level, solving (in many cases) the problems of *limited bit-level reasoning* and *strong parsing assumptions* mentioned in Sec. 3.1 —although we emphasize that DY^* and Comparse are not a multi-language composition, instead integrating the DY attacker as a library. Instead of constructing the format types (and their proofs of validity), we extract the formats using our symbolic execution as **BIR**-level terms. We translate those to DY terms, possibly losing bit-level message confusing attacks should the deduction combiner be incomplete. The criteria in [170, Sec. 2] could be useful to judge the soundness of this deduction combiner w.r.t. the message formats that are abstracted in this way, i.e., w.r.t. a given (set of) implementations.

Comparison to CompCert. CompCert was also used to verify the multi-language protocols at the assembly-code level [149, 162, 86, 87, 172, 160, 104, 140]. Among others, [149, 162, 160, 104] achieved multi-language composition by enforcing a common interaction protocol across all languages, while [86, 87, 172, 140] enforce spe-

Table 8.2: Selected approaches; Comp = Computational, No Parsing Assum. = No strict parsing assumptions (see Sec. 3.1)

Papers	Model Origin	Attacker Model	No Parsing Assum.	Formalized
Sprenger et al. [161]	Required	DY	✗	Isabelle/HOL
Arquint et al. [13, 12]	Required	DY	✗	—
Hahn et al. [91]	Required	Comp.	✓	—
Sammler et al. [150]	Required	Unbounded	✗	Coq
Bhargavan et al. [37]	Code-based	DY	✓	F^*
Wallez et al. [170]	Code-based	DY	✓	F^*
Bhargavan et al. [43, 40]	Extracted	DY / Comp.	✗	—
Aizatulin et al. [7]	Extracted	DY / Comp.	✗	—
This work (To IML)	Extracted	DY / Comp.	✗	—
This work (To SAPIC$^+$)	Extracted	DY	✓	HOL4

cific memory-sharing patterns, along with other restrictions, on the interaction between different components. In contrast, we neither depend on a common language nor impose any restrictions on the interaction of components. Our proposed method in Sec. 3.2 uses symbols for communication and predicates over these symbols for reasoning, allowing the verification toolchain to understand this interaction.

Research on multi-language semantics has explored translation between languages using a wrapper [124, 5, 138, 150]. DimSum [150] is the most relevant to our work presented in Part I. DimSum's wrapper-based composition ($\lceil\cdot\rceil_{1\rightleftharpoons 2}$) serves as a translation tool between two components written in different languages as well as between a component and the environment. Like Igloo [161], they reason about an arbitrary number of languages communicating via events and build on CSP-style parallel composition and translate between languages. Instead, we use a shared set of symbols to denote equations and deduce relations between bitstrings in different languages. DimSum requires m^2 wrappers to facilitate communication of m languages, suffering from a complexity blow-up associated with compositional soundness. Our generic deduction combiners (Sec. 3.3) can remove this burden. For computational attackers, DimSum's composition does not support probabilistic semantics and it lacks a notion of runtime bounds for attackers. As far as the DY model goes, the issues in Sec. 3.1 apply (e.g., there is no single suitable DY term that $\lceil\lceil \mathsf{senc}(m,k)\rceil_{\mathcal{T}\rightarrow \mathsf{BS}} + \mathbf{0x1}\rceil_{\mathcal{T}\leftarrow \mathsf{BS}}$ should give).

Table 8.2 compares selected works and our proposed approach in Part I.

Chapter 9

Conclusion

The increasing reliance on complex software systems for critical applications, particularly cryptographic protocols, necessitates rigorous security analysis. As compiler optimizations can subtly introduce vulnerabilities and the sheer scale of modern protocol code poses significant challenges, the traditional focus on high-level specifications often falls short. This thesis set out to address the critical need for a robust and scalable framework to analyze and verify cryptographic protocol implementations directly at the binary level, thereby bridging the gap between theoretical security properties and their real-world assurance on deployed hardware.

Our core contribution, CryptoBap, represents a significant advance in this domain. By enabling the automated extraction of formal models from low-level binaries for ARMv8 and RISC-V architectures, CryptoBap provides the foundational capability to verify essential security properties such as authentication, secrecy (weak and forward), and post-compromise security. We formally demonstrated the semantic preservation of these extracted models, establishing a sound basis for transferring verified properties back to the computational setting, thus ensuring that our analyses faithfully reflect the behavior of the original binaries. Furthermore, our novel multi-language symbolic parallel composition framework enables modular and scalable analysis of complex, heterogeneous systems, supporting end-to-end correctness results by preventing lossy translations inherent in traditional approaches.

Beyond functional correctness, a crucial innovation of this work lies in integrating resilience against microarchitectural attacks. By extending CryptoBap with hardware-induced leakage models and incorporating tools like Ghidra and provers such as DeepSec, we developed a methodology to analyze real-world protocols for subtle, implementation-level side-effects. This extended capability allowed us to uncover significant vulnerabilities in a widely used protocol like WhatsApp, including a privacy-compromising social graph inference attack via side-channel leakage, a post-compromise security violation under a clone attacker model, and critical functional discrepancies between its specified design and actual implementation. These findings underscore the inher-

ent limitations of specification-based analyses and highlight the critical importance of our binary-level approach for uncovering vulnerabilities in closed-source software that would otherwise remain undetected.

In summary, this thesis has provided a comprehensive and highly adaptable framework for binary-level cryptographic protocol analysis, pushing the boundaries of what is achievable in verifying the security of real-world implementations. By unifying advances in symbolic modeling, formal verification, and side-channel resilience, our work offers a solid and scalable foundation not only for further academic research but also for practical applications in securing critical software, ultimately enhancing the trustworthiness of digital communications and systems. The methodology developed herein provides a powerful paradigm for understanding and mitigating complex threats arising from the intricate interplay between code, compilers, and hardware, thereby contributing to the development of more secure and resilient software ecosystems.

9.1 Outlook

Finally, we discuss open challenges, some limitations of our methodology and its practical applications, and promising directions for advancing knowledge in this field.

Soundness of Model with Observations. Regarding soundness of models with observations, transpilation from assembly to **BIR** and symbolic execution remain sound, although we lack formal proofs validating our translation (see section 5.3.3). Since our primary objective with the proposed methodology is attack detection, this is acceptable, as attacks can be manually validated if false positives remain manageable. However, a positive security result from DeepSec cannot be fully trusted. For instance, we obtain a verification result for the strong secrecy of the master key (initial root key) derived during WhatsApp's session initialization with DeepSec, but refrain from reporting it as we cannot be sure it would hold.

Symbolic execution is usually only shown sound, but it is possible to show its completeness w.r.t. assignments consistent with a path condition. If the subsequent translation step can be shown to maintain that pattern, we should be able to show that a symbolic equivalence between the translated processes implies a concrete trace equivalence as long as (symbolic) traces with consistent assignments are only mapped to (symbolic) traces with consistent assignments. In our concrete application, this could be achieved by revealing the path constraints in the trace, as the program is compared to itself and thus the conditionals are observed through the PC observations.

Component Selection via Ghidra. Currently, WhatsApp components are manually identified in Ghidra through: a) preserved function symbols from its binary that correspond to the Signal protocol, and b) components interaction with the crypto library and network I/O, both of which can be systematically recognized [127]. In principle, information-flow analysis could be used to identify relevant components automatically by utilizing existing works such as [127] for crypto library and network I/O detection, and over-approximating the other component by treating all incoming inputs as potentially arbitrary.

Mitigation Against Side-channel Attacks. Side-channel attacks can be mitigated either through code-level constant-time rewrites, which involve removing secret-dependent branches and equalizing memory accesses, or by removing the cache channel, for example, via partitioning or flushing during context switches. Implementations that deploy appropriate mitigations, such as cache isolation or partitioning or flushes, would not be vulnerable under our model. Our toolchain is capable of analyzing the soundness of code-level mitigations: we re-extract the binary and execute DeepSec equivalence under $\mathcal{M}_{ct}$ and subsequently $\mathcal{M}_{spec}$. However, our current toolchain is unable to verify fence-based mitigations or hardware and operating system defenses, as this would necessitate extending the existing semantics and observation models.

Limitations of Protocol Verification Tools. The models we extracted reach the limit of what state-of-the-art verifiers can handle automatically. However, note that even some hand-written (and hand-optimized) models hit this limit, such as the Tamarin models for TLS 1.3 [67], WPA2 [69], 5G [66] and SPDM [65], which take between two hours and several days to verify, even with manually written lemmas and partially hard-coded proofs. TLS, in particular, pushes the boundaries, using 100GB of RAM and a day to validate (not generate) a proof over 1 million lines long. Over the past decade, these tools have made significant progress in handling more complex protocols, but scalability regarding the representation of the protocol (i.e., the number of rules) has not been the main focus. This is because (a) most models are created by researchers familiar with the tools, (b) for scientific publication, having a clever model-specific abstraction or manual proof intervention provides an advantage. Instead, our approach would benefit from improved scalability of these tools along with methods for decomposition, which currently, none of these tools support.

Furthermore, the problem of deciding secrecy is generally undecidable [130] even in dedicated protocol specification languages whose semantics are designed for verification. Hence, the verification tool at the backend must necessarily sacrifice automation or completeness. In practice, ProVerif sacrifices completeness but verifies many typical protocols. It is as easy to construct programs that are not recognized as secure by ProVerif as it is to write programs that purposefully break our abstractions

in symbolic execution (e.g., by using a randomly generated key to guide the control flow).

Dependence on Cryptographic Libraries. Additionally, our analysis assumes that the cryptographic library correctly adheres to the Dolev-Yao model. If the implementation deviates from this model, whether due to bugs, unforeseen cryptographic weaknesses, or side-channel vulnerabilities, our results may not hold. This assumption puts trust in the correctness of cryptographic primitives, which is not always true in real-world scenarios.

Applicability Constraints. Finally, there are practical limitations regarding applicability. Our approach requires that the target application uses a known and trusted cryptographic library and that its machine code is available for inspection. For certain applications, such as e-passport implementations, external researchers do not have access to the machine code, only standardization bodies do. While WhatsApp employs some basic obfuscation techniques, we were nevertheless able to perform a meaningful analysis. However, an application using targeted obfuscation techniques specifically designed to defeat our toolchain might succeed in evading analysis, though such deliberate obfuscation would likely raise suspicion.

Bibliography

Author's Papers for this Thesis

[P1] Nasrabadi, F., Künnemann, R., and Nemati, H. Cryptobap: a binary analysis platform for cryptographic protocols. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. Association for Computing Machinery, Copenhagen, Denmark, Aug. 2023, 1362–1376. URL: https://doi.org/10.1145/3576915.3623090.

[P2] Nasrabadi, F., Künnemann, R., and Nemati, H. Symbolic parallel composition for multi-language protocol verification. In: *2025 IEEE 38th Computer Security Foundations Symposium (CSF'25)*. IEEE Computer Society, Los Alamitos, CA, USA, June 2025, 458–473. URL: https://doi.ieeecomputersociety.org/10.1109/CSF64896.2025.00030.

[P3] Nasrabadi, F., Künnemann, R., and Nemati, H. Automated side-channel analysis of cryptographic protocol implementations. *arXiv preprint* (2025). URL: https://arxiv.org/abs/2511.11385.

Other references

[1] Abadi, M. and Rogaway, P. Reconciling two views of cryptography: the computational soundness of formal encryption. In: *IFIP International Conference on Theoretical Computer Science*. Springer. 2000, 3–22.

[2] *Absher*. URL: https://en.wikipedia.org/wiki/Absher_%5C%28application%5C%29.

[3] Acay, C., Gancher, J., Recto, R., and Myers, A. C. Secure Synthesis of Distributed Cryptographic Applications. In: *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, July 2024, 433–448. URL: https://www.computer.org/csdl/proceedings-article/csf/2024/620300a315/1W0eVQN43Kw.

[4] Acıiçmez, O. and Koç, Ç. K. Trace-driven Cache Attacks on AES (Short Paper). In: *Proceedings of the 8th International Conference on Information and Communications Security*. ICICS. Springer-Verlag, Raleigh, NC, 2006, 112–121.

[5] Ahmed, A. and Blume, M. An equivalence-preserving cps translation via multi-language semantics. In: *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. 2011, 431–444.

[6] Aizatulin, M. *Csec-Modex*. 2011. URL: https://github.com/tari3x/csec-modex.

[7] Aizatulin, M. Verifying Cryptographic Security Implementations in C Using Automated Model Extraction. PhD thesis. The Open University, 2015. URL: http://arxiv.org/abs/2001.00806.

[8] Aizatulin, M., Gordon, A. D., and Jürjens, J. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security - CCS '11*. ACM Press, Chicago, Illinois, USA, 2011, 331. URL: http://dl.acm.org/citation.cfm?doid=2046707.2046745.

[9] Aizatulin, M., Gordon, A. D., and Jürjens, J. Computational verification of C protocol implementations by symbolic execution. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS '12*. ACM Press, Raleigh, North Carolina, USA, 2012, 712. URL: http://dl.acm.org/citation.cfm?doid=2382196.2382271.

[10] Ammann, M., Hirschi, L., and Kremer, S. Dy fuzzing: formal dolev-yao models meet cryptographic protocol fuzz testing. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, 1481–1499.

[11] Arapinis, M., Chothia, T., Ritter, E., and Ryan, M. Analysing unlinkability and anonymity using the applied pi calculus. In: *2010 23rd IEEE computer security foundations symposium*. IEEE. 2010, 107–121.

[12] Arquint, L., Schwerhoff, M., Mehta, V., and Müller, P. A generic methodology for the modular verification of security protocol implementations. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, 1377–1391.

[13] Arquint, L., Wolf, F. A., Lallemand, J., Sasse, R., Sprenger, C., Wiesner, S. N., Basin, D., and Müller, P. Sound verification of security protocols: from design to interoperable implementations. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, 1065–1081.

[14] Avoine, G., Beaujeant, A., Hernandez-Castro, J., Demay, L., and Teuwen, P. A survey of security and privacy issues in epassport protocols. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–37.

[15] Backes, M., Berg, M., and Unruh, D. A formal language for cryptographic pseudocode. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Cervesato, I., Veith, H., and Voronkov, A. Springer, Berlin, Heidelberg, 2008, 353–376.

[16] Backes, M., Dreier, J., Kremer, S., and Künnemann, R. A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, 76–91.

[17] Backes, M., Künnemann, R., and Mohammadi, E. Computational soundness for dalvik bytecode. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, 717–730.

[18] Backes, M., Maffei, M., and Unruh, D. Computationally sound verification of source code. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. 2010, 387–398. URL: https://doi.org/10.1145/1866307.1866351.

[19] Baelde, D., Delaune, S., Jacomme, C., Koutsos, A., and Moreau, S. An Interactive Prover for Protocol Verification in the Computational Model. In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, 537–554.

[20] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2018).

[21] Bana, G. and Comon-Lundh, H. Towards unconditional soundness: computationally complete symbolic attacker. In: *International Conference on Principles of Security and Trust*. Springer. 2012, 189–208.

[22] Bana, G. and Comon-Lundh, H. A Computationally Complete Symbolic Attacker for Equivalence Properties. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Association for Computing Machinery, New York, NY, USA, Nov. 2014, 609–620. URL: https://doi.org/10.1145/2660267.2660276.

[23] Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., and Parno, B. Sok: computer-aided cryptography. In: *2021 IEEE symposium on security and privacy (SP)*. IEEE. 2021, 777–795.

[24] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and Cohn-Gordon, K. *RFC 9420: The Messaging Layer Security (MLS) Protocol*. USA, 2023.

[25] Barthe, G., Belaid, S., Cassiers, G., Fouque, P.-A., Gregoire, B., and Standaert, F.-X. Maskverif: automated verification of higher-order masking in presence of physical defaults. In: *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer. 2019, 300–318.

[26] Barthe, G., Betarte, G., Campo, J., Luna, C., and Pichardie, D. System-level non-interference for constant-time cryptography. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, 1267–1279.

[27] Barthe, G., Gregoire, B., Heraud, S., and Beguelin, S. Z. Computer-aided security proofs for the working cryptographer. In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Rogaway, P. Springer, Berlin, Heidelberg, 2011, 71–90.

[28] Barthe, G., Gregoire, B., and Zanella Beguelin, S. Formal certification of code-based cryptographic proofs. *ACM SIGPLAN Notices* 44, 1 (2009-01-21), 90–101.

[29] Basin, D., Cremers, C., and Meadows, C. Model Checking Security Protocols. In: *Handbook of Model Checking*. Ed. by Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R. Springer International Publishing, Cham, 2018, 727–762. URL: https://doi.org/10.1007/978-3-319-10575-8_22.

[30] Basin, D., Lochbihler, A., Maurer, U., and Sefidgar, S. R. Abstract modeling of system communication in constructive cryptography using crypthol. In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE. 2021, 1–16.

[31] Bellare, M. and Rogaway, P. Random oracles are practical: a paradigm for designing efficient protocols. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. 1993, 62–73.

[32] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D., and Maffeis, S. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 1–45.

[33] Bennetts, M. *E-government in Europe*. URL: https://en.wikipedia.org/wiki/E-government_in_Europe.

[34] Bennetts, M. *Putin launches spy app to keep Russians in 'digital gulag'*. URL: https://www.thetimes.com/world/russia-ukraine-war/article/putin-moscow-whatsapp-ban-plan-max-app-launch-b789tt6ts.

[35] Beringer, L., Petcher, A., Ye, K. Q., and Appel, A. W. Verified correctness and security of openssl hmac. In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. USENIX Association, Washington, D.C., 2015, 207–221.

[36] Bernstein, D. J. Curve25519: new diffie-hellman speed records. In: *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*. Ed. by Yung, M., Dodis, Y., Kiayias, A., and Malkin, T. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, 207–228. URL: https://doi.org/10.1007/11745853%5C_14.

[37] Bhargavan, K., Bichhawat, A., Do, Q. H., Hosseyni, P., Küsters, R., Schmitz, G., and Würtele, T. DY*: a modular symbolic verification framework for executable cryptographic protocol code. *2021 IEEE european symposium on security and privacy (EuroS&P)* (2021), 523–542.

[38] Bhargavan, K., Blanchet, B., and Kobeissi, N. Verified models and reference implementations for the tls 1.3 standard candidate. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, 483–502.

[39] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., and Strub, P. Triple handshakes and cookie cutters: breaking and fixing authentication over TLS. In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. 2014, 98–113.

[40] Bhargavan, K., Fournet, C., Corin, R., and Zalinescu, E. Cryptographically verified implementations for TLS. In: *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*. 2008, 459–468. URL: https://doi.org/10.1145/1455770.1455828.

[41] Bhargavan, K., Fournet, C., and Gordon, A. D. Verified reference implementations of ws-security protocols. In: *International Workshop on Web Services and Formal Methods*. Springer. 2006, 88–106.

[42] Bhargavan, K., Fournet, C., and Gordon, A. D. Modular verification of security protocol code by typing. *ACM Sigplan Notices* 45, 1 (2010), 445–456.

[43] Bhargavan, K., Fournet, C., Gordon, A. D., and Tse, S. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 1 (2008), 1–61.

[44] Bhargavan, K., Fournet, C., and Kohlweiss, M. miTLS: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy* 14, 6 (2016), 18–25. URL: https://doi.org/10.1109/MSP.2016.123.

[45] Blanchet, B. et al. An efficient cryptographic protocol verifier based on prolog rules. In: *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. Vol. 1. 2001, 82–96.

[46] Blanchet, B. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Secur. Comput.* 5, 4 (2008), 193–207. URL: https://doi.org/10.1109/TDSC.2007.1005.

[47] Blanchet, B. Using horn clauses for analyzing security protocols. *Formal Models and Techniques for Analyzing Security Protocols* (2011), 86–111. URL: https://ebooks.iospress.nl/doi/10.3233/978-1-60750-714-7-86.

[48] Blanchet, B. The Security Protocol Verifier ProVerif and its Horn Clause Resolution Algorithm. *Electronic Proceedings in Theoretical Computer Science* 373 (Nov. 2022), 14–22. URL: http://arxiv.org/abs/2211.12227v1.

[49] Bodo, M., Duong, T., and Kotowicz, K. This poodle bites: exploiting the ssl 3.0 fallback. In: *Security Advisory 21*. 2014, 34–58.

[50] Böhl, F., Cortier, V., and Warinschi, B. *Deduction Soundness: Prove One, Get Five for Free*. Nov. 2013, 1272.

[51] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, 1032–1043.

[52] Brinksma, E., Krilavičius, T., and Usenko, Y. S. Process algebraic approach to hybrid systems. *IFAC Proceedings Volumes* 38, 1 (2005), 325–330.

[53] Brookes, S. D., Hoare, C. A., and Roscoe, A. W. A theory of communicating sequential processes. *Journal of the ACM (JACM)* 31, 3 (1984), 560–599.

[54] Buiras, P., Nemati, H., Lindner, A., and Guanciale, R. Validation of side-channel models via observation refinement. In: *MICRO*. 2021. URL: https://doi.org/10.1145/3466752.3480130.

[55] Caballero, J. and Song, D. Automatic protocol reverse-engineering: message format extraction and field semantics inference. *Comput. Networks* 57, 2 (2013), 451–474.

[56] Cadar, C., Dunbar, D., et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In:

[57] Cadé, D. and Blanchet, B. From computationally-proved protocol specifications to implementations. In: *2012 Seventh International Conference on Availability, Reliability and Security*. IEEE. 2012, 65–74.

[58] Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., and Stefan, D. Sok: practical foundations for software spectre defenses. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, 666–680. URL: https://doi.org/10.1109/SP46214.2022.9833707.

[59] Chaki, S. and Datta, A. ASPIER: an automated framework for verifying security protocol implementations. In: *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*. 2009, 172–185. URL: https://doi.org/10.1109/CSF.2009.20.

[60] Cheval, V., Jacomme, C., Kremer, S., and Künnemann, R. Sapic+: protocol verifiers of the world, unite! In: *USENIX Security Symposium (USENIX Security), 2022*. 2022.

[61] Cheval, V., Kremer, S., and Rakotonirina, I. DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, May 2018, 529–546. URL: https://ieeexplore.ieee.org/document/8418623/.

[62] Chothia, T. and Smirnov, V. A traceability attack against e-passports. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2010, 20–34.

[63] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The mathsat5 smt solver. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, 93–107.

[64] Cortier, V. and Warinschi, B. A composable computational soundness notion. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Association for Computing Machinery, New York, NY, USA, Oct. 2011, 63–74. URL: https://doi.org/10.1145/2046707.2046717.

[65] Cremers, C., Dax, A., and Naska, A. Breaking and provably restoring authentication: A formal analysis of SPDM 1.2 including cross-protocol attacks. *IACR Cryptol. ePrint Arch.* (2024), 2047. URL: https://eprint.iacr.org/2024/2047.

[66] Cremers, C. and Dehnel-Wild, M. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In: Jan. 2019.

[67] Cremers, C., Horvat, M., Hoyland, J., Scott, S., and Merwe, T. van der. A comprehensive symbolic analysis of TLS 1.3. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Association for Computing Machinery, New York, NY, USA, 2017-10-30, 1773–1788.

[68] Cremers, C., Jacomme, C., and Naska, A. Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, 1235–1252. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-session-handling.

[69] Cremers, C., Kiesl, B., and Medinger, N. A formal analysis of IEEE 802.11's WPA2: Countering the kracks caused by cracking the counters. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Capkun, S. and Roesner, F. USENIX Association, 2020, 1–17. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/cremers.

[70] Dam, M., Guanciale, R., Khakpour, N., Nemati, H., and Schwarz, O. Formal verification of information flow security for a simple arm-based separation kernel. In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 2013, 223–234. URL: https://doi.org/10.1145/2508859.2516702.

[71] David, A. *Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems*. Packt Publishing Ltd, 2021.

[72] De Moura, L. and Bjørner, N. Z3: an efficient smt solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, 337–340.

[73] De Nicola, R. and Loreti, M. Multi labelled transition systems: a semantic framework for nominal calculi. *Electron. Notes Theor. Comput. Sci.* 169 (Mar. 2007), 133–146. URL: https://doi.org/10.1016/j.entcs.2007.05.019.

[74] Dolev, D. and Yao, A. C. On the security of public key protocols. In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. SFCS '81. IEEE Computer Society, USA, 1981, 350–357. URL: https://doi.org/10.1109/SFCS.1981.32.

[75] Donenfeld, J. A. and Milner, K. Formal verification of the wireguard protocol. *Technical Report, Tech. Rep.* (2017).

[76] Dougherty, D. J. and Guttman, J. D. An Algebra for Symbolic Diffie-Hellman Protocol Analysis. In: *Trustworthy Global Computing*. Ed. by Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Palamidessi, C., and Ryan, M. D. Vol. 8191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, 164–181. URL: http://link.springer.com/10.1007/978-3-642-41157-1_11.

[77] Dupressoir, F., Gordon, A. D., Jürjens, J., and Naumann, D. A. Guiding a general-purpose C verifier to prove cryptographic protocols. *J. Comput. Secur.* 22, 5 (2014), 823–866. URL: https://doi.org/10.3233/JCS-140508.

[78] Dutertre, B. and De Moura, L. The yices smt solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf* 2, 2 (2006), 1–2.

[79] *E-government*. URL: https://en.wikipedia.org/wiki/E-government.

[80] Eilers, M. and Müller, P. Nagini: a static verifier for python. In: *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*. Springer. 2018, 596–603.

[81] Erbsen, A., Philipoom, J., Gross, J., Sloan, R., and Chlipala, A. Simple high-level code for cryptographic arithmetic: with proofs, without compromises. *ACM SIGOPS Operating Systems Review* 54, 1 (2020), 23–30.

[82] Force, P. T. Pki for machine readable travel documents offering icc read-only access. *Technical report, International Civil Aviation Organization* (2004).

[83] Fraikin, B., Frappier, M., and St-Denis, R. Supervisory control theory with alloy. *Sci. Comput. Program.* 94 (2014), 217–237. URL: https://doi.org/10.1016/j.scico.2014.04.016.

[84] Goldwasser, S. and Micali, S. Probabilistic encryption. *Journal of Computer and System Sciences* 28, 2 (1984), 270–299. URL: https://www.sciencedirect.com/science/article/pii/0022000084900709.

[85] Goubault-Larrecq, J. and Parrennes, F. Cryptographic protocol analysis on real C code. In: *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*. 2005, 363–379. URL: https://doi.org/10.1007/978-3-540-30579-8%5C_24.

[86] Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X., Weng, S.-C., Zhang, H., and Guo, Y. Deep specifications and certified abstraction layers. *ACM SIGPLAN Notices* 50, 1 (2015), 595–608.

[87] Gu, R., Shao, Z., Kim, J., Wu, X., Koenig, J., Sjöberg, V., Chen, H., Costanzo, D., and Ramananandro, T. Certified concurrent abstraction layers. *ACM SIGPLAN Notices* 53, 4 (2018), 646–661.

[88] Guanciale, R., Balliu, M., and Dam, M. Inspectre: breaking and fixing microarchitectural vulnerabilities by formal analysis. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, 1853–1869.

[89] Guarnieri, M., Köpf, B., Reineke, J., and Vila, P. Hardware-software contracts for secure speculation. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, 1868–1883.

[90] Gupta, P. and Shmatikov, V. Towards computationally sound symbolic analysis of key exchange protocols. In: *Proceedings of the 2005 ACM workshop on Formal methods in security engineering*. 2005, 23–32.

[91] Hahn, E. M., Hartmanns, A., Hermanns, H., and Katoen, J.-P. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design* 43, 2 (2013), 191–232.

[92] Halevi, S. *A plausible approach to computer-aided cryptographic proofs*. 181. 2005. URL: https://eprint.iacr.org/2005/181.

[93] Hancke, G. P. Practical eavesdropping and skimming attacks on high-frequency rfid tokens. *Journal of Computer Security* 19, 2 (2011), 259–288.

[94] He, X., Liu, Q., Chen, S., Huang, C., Wang, D., and Meng, B. Analyzing security protocol web implementations based on model extraction with applied PI calculus. *IEEE Access* 8 (2020), 26623–26636.

[95] Hoe, A. V., Sethi, R., and Ullman, J. D. *Compilers-principles, techniques, and tools*. Vol. 5. Pearson Addison Wesley Longman Boston, 1986.

[96] Inc., S. Tech. rep. Latest Version Updated on 2025. URL: https://signal.org/docs/.

[97] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. Verifast: a powerful, sound, predictable, fast verifier for c and java. *NASA Formal Methods* 6617 (2011), 41–55.

[98] Jürjens, J. Automated security verification for crypto protocol implementations: verifying the jessie project. *Electronic Notes in Theoretical Computer Science* 250, 1 (2009), 123–136.

[99] King, J. C. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (1976), 385–394.

[100] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. Sel4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. URL: https://doi.org/10.1145/1743546.1743574.

[101] Kobeissi, N., Nicolas, G., and Bhargavan, K. Noise explorer: fully automated modeling and verification for arbitrary noise protocols. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, 356–370.

[102] Kocher, P., Horn, J., Fogh, A., and Genkin, D. Daniel gruss, werner haas, mike hamburg, moritz lipp, stefan mangard, thomas prescher, michael schwarz, and yuval yarom. spectre attacks: exploiting speculative execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[103] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. Spectre attacks: exploiting speculative execution. *Communications of the ACM* 63, 7 (2020), 93–101.

[104] Koenig, J. and Shao, Z. Compcerto: compiling certified open c components. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, 1095–1109.

[105] Koh, N., Li, Y., Li, Y., Xia, L.-y., Beringer, L., Honoré, W., Mansky, W., Pierce, B. C., and Zdancewic, S. From c to interaction trees: specifying, verifying, and testing a networked server. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019, 234–248.

[106] Kremer, S. and Künnemann, R. Automated analysis of security protocols with global state. *Journal of Computer Security* 24, 5 (2016), 583–616.

[107] Künnemann, R. and Nemati, H. Mac-in-the-box: verifying a minimalistic hardware design for MAC computation. In: *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*. 2020, 525–545.

[108] Küsters, R. and Truderung, T. Reducing protocol analysis with xor to the xor-free case in the horn theory based approach. In: *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, 129–138.

[109] Küsters, R., Truderung, T., and Graf, J. A framework for the cryptographic verification of java-like programs. In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. 2012, 198–212. URL: https://doi.org/10.1109/CSF.2012.9.

[110] Langley, A., Hamburg, M., and Turner, S. *Elliptic curves for security*. Tech. rep. 2016.

[111] Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J. R., Padon, O., and Parno, B. Verus: A Practical Foundation for Systems Verification. In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Association for Computing Machinery, New York, NY, USA, Nov. 2024, 438–454. URL: https://dl.acm.org/doi/10.1145/3694715.3695952.

[112] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[113] Lin, Z., Jiang, X., Xu, D., and Zhang, X. Automatic protocol format reverse engineering through context-aware monitored execution. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. 2008.

[114] Lincoln, P., Mitchell, J., Mitchell, M., and Scedrov, A. A probabilistic poly-time framework for protocol analysis. In: *Proceedings of the 5th ACM Conference on Computer and Communications Security*. 1998, 112–121.

[115] Lindner, A., Guanciale, R., and Dam, M. *Proof-Producing Symbolic Execution for Binary Code Verification*. 2023. URL: https://arxiv.org/abs/2304.08848.

[116] Lindner, A., Guanciale, R., and Metere, R. Trabin: trustworthy analyses of binaries. *Sci. Comput. Program.* 174 (2019), 72–89. URL: https://doi.org/10.1016/j.scico.2019.01.001.

[117] Lipp, B., Blanchet, B., and Bhargavan, K. A mechanised cryptographic proof of the wireguard virtual private network protocol. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, 231–246.

[118] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., et al. Meltdown: reading kernel memory from user space. *Communications of the ACM* 63, 6 (2020), 46–56.

[119] Lochbihler, A. Probabilistic functions and cryptographic oracles in higher order logic. In: *Programming Languages and Systems*. Ed. by Thiemann, P. Springer, Berlin, Heidelberg, 2016, 503–531.

[120] Lou, X., Zhang, T., Jiang, J., and Zhang, Y. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.* 54, 6 (2022), 122:1–122:37. URL: https://doi.org/10.1145/3456629.

[121] Lowe, G. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters* 56, 3 (Nov. 1995), 131–133. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019095001442.

[122] Lowe, G. A hierarchy of authentication specifications. In: *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*. CSFW '97. IEEE Computer Society, USA, 1997, 31.

[123] Matiyasevič, Y. Enumerable sets are diophantine. *Mathematical logic in the 20th century* (2003), 269–273.

[124] Matthews, J. and Findler, R. B. Operational semantics for multi-language programs. *ACM SIGPLAN Notices* 42, 1 (2007), 3–10.

[125] McKeeman, W. M. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[126] Meier, S., Schmidt, B., Cremers, C., and Basin, D. The tamarin prover for the symbolic analysis of security protocols. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, 696–701.

[127] Meijer, C., Moonsamy, V., and Wetzels, J. Where's crypto?: automated identification and classification of proprietary cryptographic primitives in binary code. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, 555–572.

[128] Meseguer, J. and Sharykin, R. Specification and analysis of distributed object-based stochastic hybrid systems. In: *Hybrid Systems: Computation and Control: 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006. Proceedings 9*. Springer. 2006, 460–475.

[129] Milner, K., Cremers, C., Yu, J., and Ryan, M. Automatically detecting the misuse of secrets: foundations, design principles, and applications. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE. 2017, 203–216.

[130] Mitchell, J., Scedrov, A., Durgin, N., and Lincoln, P. Undecidability of bounded security protocols. In: *Workshop on formal methods and security protocols*. Citeseer. 1999.

[131] Mosier, N., Nemati, H., Mitchell, J. C., and Trippel, C. Serberus: protecting cryptographic code from spectres at compile-time. In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, 4200–4219. URL: https://doi.org/10.1109/SP54263.2024.00048.

[132] *Most Popular Messaging Apps In 2025*. URL: https://www.similarweb.com/blog/research/apps/worldwide-messaging-apps/.

[133] Müller, P., Schwerhoff, M., and Summers, A. J. Viper: a verification infrastructure for permission-based reasoning. In: *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*. Springer. 2016, 41–62.

[134] Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D. M., and Stefan, D. Swivel: hardening webassembly against spectre. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Bailey, M. D. and Greenstadt, R. USENIX Association, 2021, 1433–1450. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[135] Needham, R. M. and Schroeder, M. D. Using encryption for authentication in large networks of computers. *Communications of the ACM* 21, 12 (Dec. 1978), 993–999. URL: https://dl.acm.org/doi/10.1145/359657.359659.

[136] Nemati, H., Buiras, P., Lindner, A., Guanciale, R., and Jacobs, S. Validation of abstract side-channel models for computer architectures. In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer. 2020, 225–248.

[137] Neve, M. and Seifert, J.-P. Advances on access-driven cache attacks on AES. In: *Proceedings of the 13th International Conference on Selected Areas in Cryptography*. SAC'06. Springer-Verlag, Montreal, Canada, 2007, 147–162.

[138] New, M. S. and Ahmed, A. Graduality from embedding-projection pairs. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[139] O'Shea, N. Using elyjah to analyse java implementations of cryptographic protocols. In: *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS-2008)*. 2008.

[140] Oliveira Vale, A., Melliès, P.-A., Shao, Z., Koenig, J., and Stefanesco, L. Layered and object-based game semantics. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–32.

[141] Olmos, S. A., Barthe, G., Chuengsatiansup, C., Grégoire, B., Laporte, V., Oliveira, T., Schwabe, P., Yarom, Y., and Zhang, Z. Protecting cryptographic code against spectre-rsb. *IACR Cryptol. ePrint Arch.* (2024), 1070. URL: https://eprint.iacr.org/2024/1070.

[142] Oortwijn, W. and Huisman, M. Practical abstractions for automated verification of message passing concurrency. In: *Integrated Formal Methods: 15th International Conference, IFM 2019, Bergen, Norway, December 2–6, 2019, Proceedings 15*. Springer. 2019, 399–417.

[143] Owens, K., Alem, A., Roesner, F., and Kohno, T. Electronic monitoring smartphone apps: an analysis of risks from technical, Human-Centered, and legal perspectives. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, Aug. 2022, 4077–4094. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/owens.

[144] Paulson, L. C. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security* 6, 1-2 (Jan. 1998), 85–128. URL: https://journals.sagepub.com/action/showAbstract.

[145] Paulson, T. N. L. C. and Wenzel, M. *A Proof Assistant for Higher-Order Logic*. 2013. URL: https://isabelle.in.tum.de/.

[146] Percival, C. Cache Missing for Fun and Profit. In: *BSDCan*. 2005.

[147] Petcher, A. and Morrisett, G. A mechanized proof of security for searchable symmetric encryption. In: *2015 IEEE 28th Computer Security Foundations Symposium*. 2015 IEEE 28th Computer Security Foundations Symposium. ISSN: 2377-5459. 2015-07, 481–494.

[148] Plotkin, G. An operational semantics for csp. In: *Workshop on Logic of Programs*. 1980.

[149] Ramananandro, T., Shao, Z., Weng, S.-C., Koenig, J., and Fu, Y. A compositional semantics for verified separate compilation and linking. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. 2015, 3–14.

[150] Sammler, M., Spies, S., Song, Y., D'Osualdo, E., Krebbers, R., Garg, D., and Dreyer, D. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 775–805. URL: https://dl.acm.org/doi/10.1145/3571220.

[151] Scerri, G. Proofs of Security Protocols Revisited. PhD thesis. Ecole Normale Supérieure de Cachan, Jan. 2015. URL: https://theses.hal.science/tel-01133067.

[152] Schmidt, B., Meier, S., Cremers, C., and Basin, D. Automated analysis of diffie-hellman protocols and advanced security properties. In: *2012 IEEE 25th Computer Security Foundations Symposium*. 2012 IEEE 25th Computer Security Foundations Symposium. ISSN: 2377-5459. June 2012, 78–94.

[153] Sergey, I., Wilcox, J. R., and Tatlock, Z. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[154] Shivakumar, B. A., Barnes, J., Barthe, G., Cauligi, S., Chuengsatiansup, C., Genkin, D., O'Connell, S., Schwabe, P., Sim, R. Q., and Yarom, Y. Spectre declassified: reading from the right place at the wrong time. In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, 1753–1770. URL: https://doi.org/10.1109/SP46215.2023.10179355.

[155] Shivakumar, B. A., Barthe, G., Grégoire, B., Laporte, V., Oliveira, T., Priya, S., Schwabe, P., and Tabary-Maujean, L. Typing high-speed cryptography against spectre v1. In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, 1094–1111. URL: https://doi.org/10.1109/SP46215.2023.10179418.

[156] Sidhpurwala, H. Security flaws caused by compiler optimizations. *Red Hat Blog* (2019).

[157] Silva, R. and Butler, M. Shared event composition/decomposition in event-b. In: *Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29-December 1, 2010. Revised Papers 9*. Springer. 2012, 122–141.

[158] Simon, L., Chisnall, D., and Anderson, R. What you get is what you c: controlling side effects in mainstream c compilers. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, 1–15.

[159] *Singpass Singapore's National Digital Identity (Factsheet)*. URL: https://www.mddi.gov.sg/newsroom/singpass-factsheet-02032022.

[160] Song, Y., Cho, M., Kim, D., Kim, Y., Kang, J., and Hur, C.-K. Compcertm: compcert with c-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.

[161] Sprenger, C., Klenze, T., Eilers, M., Wolf, F. A., Müller, P., Clochard, M., and Basin, D. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–31. URL: https://dl.acm.org/doi/10.1145/3428220.

[162] Stewart, G., Beringer, L., Cuellar, S., and Appel, A. W. Compositional compcert. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015, 275–287.

[163] Strejcek, J. and Trtík, M. Abstracting path conditions. In: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 2012, 155–165.

[164] Strubbe, S. N. and Schaft, A. van der. Compositional modelling of stochastic hybrid systems. *Cassandras and Lygeros [CL06]* (2006), 47–77.

[165] team, H. development. *HOL Interactive Theorem Prover*. 2022. URL: https://hol-theorem-prover.org/.

[166] *TraceTogether*. URL: https://en.wikipedia.org/wiki/TraceTogether.

[167] Tromer, E., Osvik, D. A., and Shamir, A. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.* 23, 2 (Jan. 2010), 37–71.

[168] Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. Cryptanalysis of DES implemented on computers with cache. In: *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*. CHES'03, LNCS. Springer, 2003, 62–76.

[169] Vassena, M., Disselkoen, C., Gleissenthall, K. von, Cauligi, S., Kici, R. G., Jhala, R., Tullsen, D. M., and Stefan, D. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. URL: https://doi.org/10.1145/3434330.

[170] Wallez, T., Protzenko, J., and Bhargavan, K. Comparse: Provably Secure Formats for Cryptographic Protocols. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Association for Computing Machinery, New York, NY, USA, Nov. 2023, 564–578. URL: https://doi.org/10.1145/3576915.3623201.

[171] Wang, X., Zeldovich, N., Kaashoek, M. F., and Solar-Lezama, A. Towards optimization-safe systems: analyzing the impact of undefined behavior. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Association for Computing Machinery, Farminton, Pennsylvania, 2013, 260–275. URL: https://doi.org/10.1145/2517349.2522728.

[172] Wang, Y., Wilke, P., and Shao, Z. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[173] WhatsApp Inc. / Meta Platforms. *WhatsApp Encryption Overview*. Tech. rep. Version 8 Updated August 19, 2024.

[174] Wolf, F. A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J. C., and Müller, P. Gobra: modular specification and verification of go programs. In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer. 2021, 367–379.

[175] Wondracek, G., Comparetti, P. M., Krügel, C., and Kirda, E. Automatic network protocol analysis. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. 2008.

[176] Xiao, M., Zhang, S., and Luo, Y. Automatic network protocol message format analysis. *J. Intell. Fuzzy Syst.* 31, 4 (2016), 2271–2279.

[177] Xu, J., Lu, K., Du, Z., Ding, Z., Li, L., Wu, Q., Payer, M., and Mao, B. Silent bugs matter: a study of compiler-introduced security bugs. In: *Proceedings of the 32th USENIX Conference on Security Symposium*. SEC'23. USENIX Association, Aug. 2023. URL: https://www.usenix.org/system/files/sec23fall-prepub-123-xu-jianhao.pdf.

[178] Yu, J., Ryan, M., and Cremers, C. Decim: detecting endpoint compromise in messaging. *IEEE Transactions on Information Forensics and Security* 13, 1 (2017), 106–118.

[179] Zinzindohoué, J.-K., Bhargavan, K., Protzenko, J., and Beurdouche, B. Hacl*: a verified modern cryptographic library. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, 1789–1806.