PICO: A Presburger In-bounds Check Optimization for Compiler-based Memory Safety Instrumentations

TINA JUNG, FABIAN RITTER, and SEBASTIAN HACK, Saarland University, SIC, Germany

Memory safety violations such as buffer overflows are a threat to security to this day. A common solution to ensure memory safety for C is code instrumentation. However, this often causes high execution-time overhead and is therefore rarely used in production.

Static analyses can reduce this overhead by proving some memory accesses in bounds at compile time. In practice, however, static analyses may fail to verify in-bounds accesses due to over-approximation. Therefore, it is important to additionally optimize the checks that reside in the program.

In this article, we present PICO, an approach to eliminate *and* replace in-bounds checks. PICO exactly captures the spatial memory safety of accesses using Presburger formulas to either verify them statically or substitute existing checks with more efficient ones. Thereby, PICO can generate checks of which each covers multiple accesses and place them at infrequently executed locations.

We evaluate our LLVM-based PICO prototype with the well-known SoftBound instrumentation on SPEC benchmarks commonly used in related work. PICO reduces the execution-time overhead introduced by SoftBound by 36% on average (and the code-size overhead by 24%). Our evaluation shows that the impact of substituting checks dominates that of removing provably redundant checks.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Software safety**; **Software performance**; • **Security and privacy** → **Systems security**

Additional Key Words and Phrases: Optimization, spatial memory safety, Presburger, C language, LLVM

ACM Reference format:
Tina Jung, Fabian Ritter, and Sebastian Hack. 2021. PICO: A Presburger In-bounds Check Optimization for Compiler-based Memory Safety Instrumentations. *ACM Trans. Arch. Code Optim.* 18, 4, Article 45 (July 2021), 27 pages.
https://doi.org/10.1145/3460434

1 INTRODUCTION

Buffer overflow vulnerabilities have been a threat to security for decades. This threat still persists: In 2019, the Common Weakness Enumeration Top 25 [MITRE Corporation 2019] ranks buffer overflows the most dangerous vulnerability.

Memory safety violations are mostly a concern of programming languages such as C, where accessing a pointer outside of its bounds is undefined behavior. The programmer cannot always

45

We acknowledge support by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) and Saarland University within the funding programme Open Access Publishing.
Authors' addresses: T. Jung, F. Ritter, and S. Hack, Saarland University, SIC, Compiler Design Lab, Germany; emails: {t.jung, fabian.ritter, hack}@cs.uni-saarland.de.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/author(s).
1544-3566/2021/07-ART45
https://doi.org/10.1145/3460434

detect undefined behavior when running the program, as the program might still behave as expected. Thus, memory safety errors are both dangerous and hard to find manually.

A common solution to detect memory safety violations in C automatically is code instrumentation. Memory safety instrumentations monitor allocations and memory accesses to report an error if an access is out of bounds. For this purpose, they insert calls to a runtime library at compile time. The runtime library keeps track of allocations and provides functions for in-bounds checks. Unfortunately, this instrumentation incurs substantial execution-time overhead. Because of this overhead, instrumentation is rarely used in production environments [Szekeres et al. 2013]. Since one of the earliest instrumentations introduced by Jones and Kelly [1997], many other solutions have been proposed (e.g., References [Akritidis et al. 2009; Dhurjati and Adve 2006; Duck and Yap 2016; Grossman et al. 2005; Kroes et al. 2018; Necula et al. 2002; Tarditi et al. 2018]), all with different safety guarantees and different execution-time overhead. The reported slowdowns for those that provide full spatial safety range from 1.7–1.8× [Nagarakatte et al. 2015] to 11–12× [Ruwase and Lam 2004].

Two main approaches are used to reduce execution-time overhead of in-bounds checks. First, static analyses [Logozzo and Fähndrich 2008; Nazaré et al. 2014] remove checks if they can statically show that these checks never fail. The second class of approaches targets loops, since checks in loops tend to be executed frequently at runtime. Akritidis et al. [2009] and Ye et al. [2014] compute sufficient conditions that an access within a loop is in bounds. If these sufficient conditions are fulfilled at runtime, the exact check within the loop is skipped. However, the exact check has to be executed in addition if these conditions are not met, because the conditions are only sufficient and *not necessary.* Effectively, they introduce further check code to create a fast path in the loop.

PICO, the optimization we present in this article, is a novel technique to optimize bounds checks and their placement that contains proving checks unnecessary as a special case. To this end, PICO computes Presburger formulas that describe sufficient *and necessary* conditions for accesses to be in bounds. If the computed formula is valid, then the memory access is safe and there is no need for a runtime check. Otherwise, the formula is used to generate code for a runtime check. PICO's formulas are specific to program locations: Parameters are used to abstract from non-affine expressions, such that a check can only be placed at locations where the values for these expressions are available.

PICO addresses two aspects of existing memory safety instrumentations that are hindering an efficient execution of a safe program. First, they place one check for every access. However, programs tend to access overlapping or neighboring memory, such that a range check that guards multiple accesses is often more efficient. Second, instrumentations place checks directly at the accesses. This is especially costly if the check is within a loop. The underlying problem of both aspects is that, in most cases, a check cannot be moved without affecting the program behavior. If a check is moved, then it might be executed although the access would not be, which can lead to false violation reports. Even worse, if the access is executed and the check is not, then the program is unsafe.

PICO tackles both problems: First, it can compute checks that do not only guard one but possibly multiple accesses by combining the Presburger formulas for the accesses. For overlapping or neighboring accesses, the resulting checks need fewer instructions than individual checks. Second, PICO is able to place checks at different program locations by incorporating control-flow conditions into the check such that an error is reported *if and only if* the access is unsafe and *executed.*

Our optimization relies on a memory safety instrumentation to provide bounds for runtime checks, but it is conceptually independent of the instrumentation used. We demonstrate the effectiveness of PICO by evaluating it with SoftBound [Nagarakatte et al. 2009], currently one of the

```
 1  int *f(int n, int x) {
 2    if (n < 1)
 3      return NULL;
 4
 5    int *Ar =
        malloc(n * sizeof(int));
 6
 7
 8    for (int i=0; i<n; i++) {
 9

10      Ar[i] = 0;
11    }
12

13    Ar[x] = 1;
14    return Ar;
15  }
```

```
int *f(int n, int x) {
  if (n < 1)
    return NULL;

  int *Ar =
    malloc(n * sizeof(int));
  char *ArBs = loadBase(Ar);
  char *ArBnd = loadBound(Ar);
  for (int i=0; i<n; i++) {
    checkIB(ArBs, ArBnd, Ar+i,
            sizeof(int));
    Ar[i] = 0;
  }
  checkIB(ArBs, ArBnd, Ar+x,
          sizeof(int));
  Ar[x] = 1;
  return Ar;
}
```

```
int *f(int n, int x) {
  if (n < 1)
    return NULL;
  assert(x >= 0 && x < n);
  int *Ar =
    malloc(n * sizeof(int));


  for (int i=0; i<n; i++) {


    Ar[i] = 0;
  }


  Ar[x] = 1;
  return Ar;
}
```

Fig. 1. Running example (left), its instrumented version (middle), and its optimized version (right).

best-performing memory safety instrumentations. PICO is implemented in the LLVM compiler framework [Lattner and Adve 2004].

In summary, the contributions of this article are:

- *A Constraint Computation* technique to statically compute exact in-bounds constraints for memory accesses. The accesses are proven in bounds if and only if the constraints are valid. If they are not, then the computed constraints can be used as a runtime check for the accesses.
- *A Check Placement Strategy* to automatically place checks at program locations where they incur low runtime overhead. Additionally, a PICO check can cover multiple memory accesses.
- *The Safety Guarantees* after optimization are the same as those given by the underlying memory safety instrumentation. No false violation reports are introduced, as the computed in-bounds check conditions are necessary.
- We present a *proof-of-concept implementation in the LLVM compiler framework* that we evaluate with the memory safety instrumentation SoftBound. We use the SPEC CPU 2000/2006 Benchmarks commonly used in related work to demonstrate that the optimization scales to real-world programs. Our evaluation shows that PICO reduces
  - the execution-time slowdown of SoftBound from 1.99× to 1.64× on average and
  - the size overhead of the instrumented binary by 24%.

  Furthermore, we show that substituting checks by more efficient ones improves performance far more than just removing provably redundant checks.

2 OVERVIEW

Before we go into the details of PICO, we want to convey an intuition of the optimization goal. For this purpose, we show the result of a typical memory safety instrumentation and explain the results of our optimization by means of a running example.

The C program in Figure 1 (left) allocates an integer array of size n, initializes all of its elements to 0 and additionally places a 1 at location x.

A compiler-based memory safety instrumentation[1] places in-bounds checks before every memory access, e.g., in line 9 and 12 in our example. The resulting code looks similar to the one shown in Figure 1 (middle). *loadBase* and *loadBound* (lines 6, 7) return a pointer to the beginning and one

[1]Applies to, e.g., SoftBound [Nagarakatte et al. 2009; Jones and Kelly 1997] as well as Low-Fat Pointers [Duck and Yap 2016].

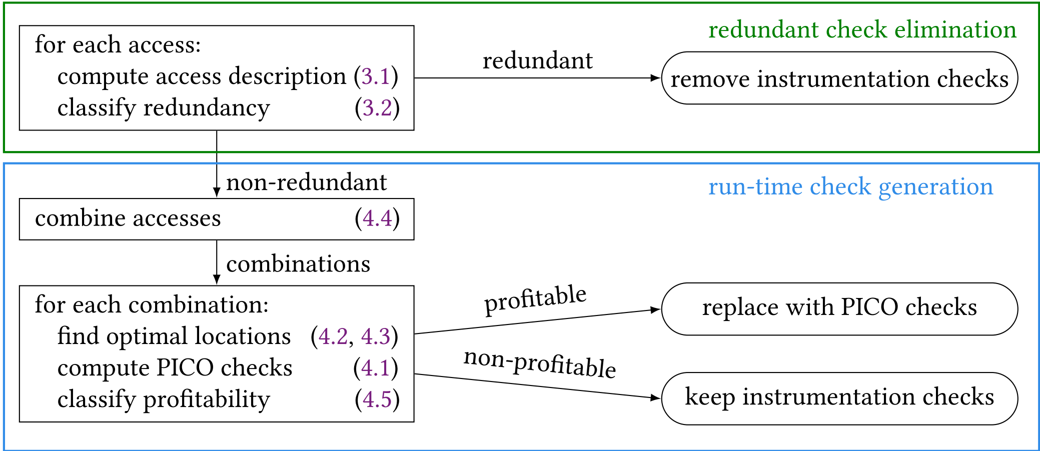

Fig. 2. Overview of PICO.

past the end of the memory allocation that is valid to be accessed through *Ar*. *checkIB* uses base, bound, the pointer itself, and the access width to ensure that *Ar* still points to its assigned region.

The number of calls to *checkIB* in this example depends on the function parameter n. In case n is less than one, no calls are made. Otherwise, the number of calls is $n + 1$. The allocation has size $n \cdot sizeof(int)$ and the access ranges from 0 to $n - 1$, which means that no memory beyond the allocated space is accessed and the access in the loop is always in bounds.

The check inside the loop is what we call *redundant*, i.e., it will never fail. This is different for the check in line 12. It depends on the relation between x and n whether the access is in bounds or out of bounds. The constraint that needs to hold for the $Ar[x]$ access to be within bounds is that $x \geq 0$ and $x < n$. Based on this idea, PICO optimizes the instrumented function as shown in Figure 1 (right). If $n \geq 1$, then the optimized function now executes two comparisons instead of $n+1$ runtime in-bounds checks. PICO optimizes the program at compile-time and is fully automated.

Figure 2 gives an overview of PICO: The *redundant check elimination* box describes PICO's initial steps. A static analysis is used to collect information on memory regions that are accessed (Section 3.1). Based on this information, some checks can be determined to never fail, while for others this may depend on the inputs (Section 3.2). Checks that are shown to never fail, i.e., redundant checks, can be removed from the instrumented program.

PICO aims to additionally improve on the checks that cannot be eliminated from the program. The *runtime check generation* box shows PICO's steps to transform a program such that it executes faster than the original instrumented program, while preserving the memory safety guarantees.

First, we decide which accesses to check together (Section 4.4). For each of these combinations of accesses, we pick a check location (Sections 4.2, 4.3). A check should only issue an error if the access is invalid and executed. The latter has to be considered when placing the check at a location that is not right before the access; we discuss our solution in Section 4.1. Finally, we decide whether using a PICO check is more profitable than keeping the one from the instrumentation (Section 4.5).

3 RUNTIME CHECK GENERATION

PICO computes memory safety checks at compile time. It requires knowledge about the memory accesses as well as the bound information of arrays and structures. The first step is therefore an analysis to collect this information. Sections 3.1 and 3.2 introduce the background for our work.

3.1 Memory Access Analysis

PICO employs a flow-sensitive value analysis that associates every variable with a Presburger[2] formula. An example for such an association is:

$$\{v \mid (v = x + 5 \land x \leq 0) \lor (v = x - 5 \land x > 0)\}.$$

The value described by this set is $x+5$ if $x \leq 0$ holds and $x-5$ otherwise. We refer to free variables, such as x in the example, as parameters.

Our analysis is based on LLVM's Scalar Evolution [Pop et al. 2005] analysis, which is a flow-insensitive analysis that computes closed-form recurrences for loop-variant expressions. First, we describe the basic translation from Scalar Evolution information to parametrized sets that is available in LLVM through the polyhedral optimizer Polly [Grosser et al. 2012]. Afterwards, we explain our extension to flow-sensitive information [Jung 2015].

3.1.1 Flow-insensitive Information. *SCEVs* are the result of SCalar EVolution's analysis and express value information for an expression. For example, the SCEV for $x + 5$ is:

$$\mathit{AddExpr}(\mathit{Unknown}(x),\ \mathit{Constant}(5)).$$

The basic building blocks of SCEVs are *Constant*s and *Unknown*s. Constants are used whenever a compile-time constant is encountered, whereas Unknowns express that there is no value information available for the expression (e.g., for values loaded from memory or function parameters). Everything more complex than these basic elements, e.g., an *AddExpr*, is recursively defined on top of them. This SCEV is translated to the parametrized set comprehension $\{v \mid v = x + 5\}$. The *Unknown* becomes a parameter x that we use in our description of the value. The set describes the value that the expression has, given a value for x.

The more interesting information that Scalar Evolution can (in certain cases) provide is value information of loop-variant expressions. This information is given as a so-called *additive recurrence* [Bachmann et al. 1994; Engelen 2001], which describes the evolution of a value within a loop. The general form of an additive recurrence looks as follows:

$$\mathit{AddRec}(\mathit{start}, +, \mathit{stride}, \mathit{loop}).$$

The recurrence describes the initial value as *start* and that it is incremented by *stride* in every iteration of the loop. This recurrence can be translated to the set comprehension

$$\{v \mid v = \mathit{start} + \mathit{stride} \cdot l\}.$$

The parameter l is introduced to allow reasoning about the value given a loop iteration l. If *stride* is a compile-time constant, then the constraint is in the theory of Presburger formulas.[3]

Consider the running example in Figure 1 again. Scalar Evolution provides the recurrence $\mathit{AddRec}(0, +, 1, \mathrm{loop}_i)$ for the loop variable i. The name loop_i uniquely identifies the loop that the recurrence refers to; we introduce l_i instead of l for the same reason. The recurrence corresponds to the set comprehension $\{v \mid v = l_i\}$ as $\mathit{start} = 0$ and $\mathit{stride} = 1$. Now that we have value information for i, we can describe the memory accessed by $Ar[i]$. We introduce a parameter to abstract from the pointer value of Ar and get the following description for the accessed memory addresses *Mem*[4]:

$$\{\mathit{Mem} \mid \mathit{Mem} = \mathit{Ar} + l_i\}.$$

[2] Presburger arithmetic, also known as **linear integer arithmetic (LIA)**, is a decidable fragment of Peano arithmetic that excludes the multiplication of two variables. Derived from the use of Presburger formulas, we gave the article the title **P**resburger **I**n-bounds **C**heck **O**ptimization.

[3] Finkel and Leroux [2002] provide a more thorough account on when state sets are representable as Presburger formulas.

[4] The pointer arithmetic here operates on int-sized data for easy readability (the implementation uses byte granularity).

```
1   int a[10];
2   if (i == 5)
3     a[i] = 0;
```

Fig. 3. Control-flow conveys value information.

This precisely describes the memory accessed in each individual iteration of the loop, but we want to argue about all accessed memory locations later on, and for this the current information is not sufficient. The missing piece of information is how often the loop is executed. Scalar Evolution provides a so called *backedge-taken count* (n in our example), that describes how often the jump from the end of the loop to its start is taken. The backedge-taken count is the number of executions of the loop body for `for`- and `while`-loops.[5] We enrich our description with the number of loop iterations by adding the constraint $0 \leq l_i < n$ accordingly:

$$\{Mem \mid Mem = Ar + l_i \wedge 0 \leq l_i < n\}.$$

We now get a description of all memory accessed within the loop by using *projection* (e.g., [Verdoolaege 2010]). Projection is a standard technique to eliminate a variable from a constraint system without changing the satisfiability of the constraint system. We will denote the projection of a variable v as π_v in the following. The projection π_{l_i} results in:

$$\{Mem \mid Ar \leq Mem < Ar + n\}.$$

This complies with the intuition that the memory starting from address Ar up to $Ar + n$ is accessed within the loop.

The memory access in line 10 of our running example (Figure 1) is always safe. This finding is based on knowledge about the allocation size of the accessed array, which is derived from the `malloc` in line 5. To enable this reasoning in our optimization, we not only collect information on accesses but also on allocations.

If `malloc` successfully allocates memory, then the returned pointer points to the beginning of the region that is valid to access.[6] The argument handed to the call describes the size of the allocated memory. We describe the accessible memory for `Ar` derived from this knowledge with:

$$\{Mem \mid Ar \leq Mem < Ar + n\}.$$

3.1.2 Flow-sensitive Information. The value information provided by Scalar Evolution is flow-insensitive. Control flow can provide information on variable values that can only be captured by analyzing conditionals. Consider the small example in Figure 3.

The true branch is executed if $i == 5$ evaluates to true. By utilizing Scalar Evolution to provide information for both operands of the conditional, i and 5, we can compute a constraint that represents the equality $i = 5$. We propagate this condition into the true block by adding the constraint $i = 5$ to every SCEV value information computed for this block. The false branch propagates the information $i \neq 5$. Whenever a block has multiple control-flow predecessors, we join their control-flow conditions using a disjunction. In our example, we get the condition $i = 5 \vee i \neq 5$ after the if statement, which can be simplified to *true*. The value *true* indicates that the block is always executed.

[5] On the IR level, which lacks the high-level loop structure, the backedge-taken count is used for accesses in blocks that are post-dominated by the loop backedge.

[6] We assume that if the allocation of memory through `malloc` fails, a temporal memory safety instrumentation triggers an error upon access. We therefore model the accessible memory region as if `malloc` was always successful.

Flow sensitivity allows to reason that the value of i can only be 5 at the location of the access. If we consider that the allocation is of size 10, then this makes the difference between proving the access in bounds at compile time and having to execute a runtime check.

3.2 Check Generation for Statically Known Allocation Sites

The next two sections describe the automatic derivation of in-bounds constraints, given the access and allocation information from Section 3.1. In general, the allocation from which a pointer is derived cannot be identified at compile time. In some cases, however, a statically unique allocation for a pointer can be determined. We refer to those as *pointers with a statically known allocation site*.

PICO derives precise constraints that are fulfilled if and only if the access is in bounds. To achieve this for a pointer with a statically known allocation site, we now explain a method by Doerfert et al. [2017] that we extend in Section 3.3 to handle arbitrary pointer origins. We first describe the formulas for the derivation of in-bounds conditions and afterwards show how they can be used to generate the constraints that were placed as a runtime check in the running example.

To compute in-bounds conditions for an access, we use the set comprehension of the access itself, *Access*, and the one for the bounds of the allocation, *Bounds*. The memory region that is not to be accessed is the complement of the bounds ($\overline{Bounds}$). The intersection of the access with the out-of-bounds region yields a description of the out-of-bounds memory location accessed (*OOBMem*), along with constraints under which an out-of-bounds access occurs:

$$OOBMem := Access \cap \overline{Bounds}.$$

As we are only interested in the constraints under which an out-of-bounds access happens (*OOB*), not the memory locations, we project out *Mem*:

$$OOB := \pi_{Mem}(OOBMem).$$

Finally, the condition that describes that the access is in bounds (*IB*), is the complement of the out-of-bounds condition:

$$IB := \overline{OOB}.$$

The access is within bounds iff the *IB* constraints are fulfilled. There are two notable special cases of constraints: *True* means that the access is always within bounds, while *False* indicates that the access is always out of bounds.

In the following, we use the presented technique to derive the results seen in Figure 1 (right). The access information for `Ar[i]` (*iA*), `Ar[x]` (*xA*), and the accessible memory information for `Ar` (*ArAlloc*) look as follows:

$$iA := \{Mem \mid Ar \leq Mem < Ar + n\},$$
$$xA := \{Mem \mid Mem = Ar + x\},$$
$$ArAlloc := \{Mem \mid Ar \leq Mem < Ar + n\}.$$

The constraints for the access `Ar[x]` can be derived as follows:

$$\overline{ArAlloc} = \{Mem \mid Mem < Ar \vee Mem \geq Ar + n\},$$
$$OOBMem := \overline{ArAlloc} \cap xA,$$
$$= \{Mem \mid Mem = Ar + x \wedge (x < 0 \vee n \leq x)\},$$
$$OOB := \pi_{Mem}(OOBMem) = x < 0 \vee n \leq x,$$
$$IB := \overline{OOB} = x \geq 0 \wedge n > x.$$

The constraint *IB* derived here is the one seen in the optimized running example (Figure 1 (right)).

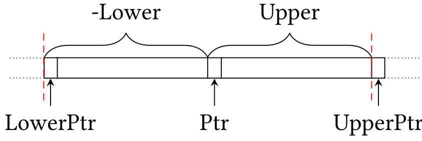

Fig. 4. Computation of symbolic bounds for a pointer. The dashed lines mark the allocation boundaries.

```
1 void f(int n, int *Ar) {
2   for (int i=0; i<n; i++)
3     Ar[i] = 0;
4 }
```

Fig. 5. Conditionally executed access.

The loop access in the running example has the same description as the allocation it accesses:

$$ArAlloc = iA,$$
$$OOBMem := \overline{iA} \cap iA = \{\},$$
$$OOB := \pi_{Mem}(OOBMem) = false,$$
$$IB := \overline{OOB} = true.$$

The access Ar[i] is statically shown to be in bounds and no runtime check needs to be placed.

3.3 Extension to Arbitrary Pointer Origins

For many programs, the allocation site of a pointer cannot be determined precisely at compile time. Consider a function that takes a pointer as an argument and has multiple call sites. The origin of the pointer might depend on the call site. For such pointers, we do not statically have the information on their base and bound that is necessary for the in-bounds check computation. In this case, PICO uses the bounds provided by an accompanying memory-safety instrumentation. The instrumentation provides a lower bound address *LowerPtr* and an upper bound address *UpperPtr* of the allocation for a given pointer *Ptr*.

To apply the check generation technique from Section 3.2, we are interested in the offsets of the pointer to its lower and upper bound, rather than the pointer values themselves. Figure 4 shows the offsets *Lower* and *Upper*, which can be computed from the pointer *Ptr* and the associated allocation bounds (*LowerPtr* and *UpperPtr*).

A general form for bounds that utilizes this idea can be described as:

$$symBounds := \{Mem \mid Ptr + Lower \leq Mem < Ptr + Upper\}.$$

Since these bounds refer to our newly introduced symbols *Lower* and *Upper* and are not directly derived from program expressions, we call them *symbolic bounds* (*symBounds*).

The *symBounds* are similar to what was derived for `malloc` in our running example (Figure 1). Note, however, that in case of `malloc` the offset to the lower bound is always zero, as the pointer points to the beginning of the allocated region. The C language allows pointers to point to the middle of an allocation, therefore, we model the more general case with an offset to the upper and the lower bound. In case the pointer is in bounds, $Lower \leq 0$ and $Upper > 0$ holds.

With the help of symbolic bounds, we can use the technique described in Section 3.2 without modifications to compute in-bounds checks for pointer accesses with unknown pointer origin.

4 RUNTIME CHECK PLACEMENT

This section presents PICO's technique to compute combined runtime checks for favorable locations. The *runtime check generation* box in Figure 2 summarizes the steps to achieve this, and this section gives a more detailed insight.

4.1 Location-dependent Check Generation

In the optimized version of our running example in Figure 1 (right), a precise in-bounds check is placed before the allocation of the accessed array. This is possible without considering control flow, as all memory accesses are executed unconditionally after that point. This does not hold in general. If a check is placed at a location different from the access location, then the check might be executed although the access is not. Executing the check but not the access can lead to false alarms.

There are two ways in which a check can depend on its location: First, the control-flow conditions that have to be taken into account at different locations to avoid false alarms can differ. Second, checks for accesses that depend on some loop iteration variable (we call these accesses loop variant) will look differently within and outside of the loop. An access is *loop variant* in some loop L whenever the computed address changes at some iteration of the loop L. As this is not statically decidable in general, we use a conservative approximation: We classify an access as loop variant if some value involved in the address computation might change in some iteration. We discuss both aspects in more detail next.

Control-flow Conditions. Figure 5 shows a function containing a loop that, given a number n and a pointer to an array Ar, sets the first n elements of Ar to 0. The description of the access is the same as in the running example (Figure 1), as the loop is identical:

$$iA := \{Mem \mid Ar \leq Mem < Ar + n\}.$$

However, as no allocation site for Ar is available, we cannot show that this check is redundant. We use symbolic bounds and apply the technique from Section 3.2 to derive the in-bounds condition:

$$n \leq 0 \vee (Lower \leq 0 \wedge Upper \geq n).$$

The first part of the conjunction, $n \leq 0$, stems from the fact that the access is only executed if $n > 0$ holds. The other part, $Lower \leq 0 \wedge Upper \geq n$, ensures that if the loop is executed, then all accesses are in bounds.

The flow-sensitive analysis described in Section 3.1.2 propagates all control-flow conditions along the control-flow edges to the accesses. These conditions are helpful to get more precise information on variables, so we can prove accesses such as the one in our running example safe. However, assume we want to place a check in the block where the access resides. At that location, there is no need to check that an error is issued only if the access is executed, as we know it is executed.

Take the access in Figure 5 for example. If we compute a check for the location within the loop, then the check would still contain the constraint $n \leq 0$. However, from the control flow that leads to the loop body, we know that $n \leq 0$ does not hold and can simplify the check.

Let the function $CFCondsAt(B)$ return all control-flow conditions that hold at a location B. As seen in the example, a check for an access that is located at block A contains the negated conditions $\overline{CFCondsAt(A)}$. This ensures that the check only reports an error if the access is executed. However, if we want to place a check at some location L, then we know that the conditions $\overline{CFCondsAt(L)}$ hold and can use this to simplify the check. Formally, we compute a set of constraints X, such that $X \cap CFCondsAt(L) = \overline{CFCondsAt(A)} \cap CFCondsAt(L)$ holds and use it instead of $\overline{CFCondsAt(A)}$ in the check. This simplification is commonly referred to as the *gist* operation [Pugh 1994].

Loop-variant Accesses. Checks for loop variant accesses (such as the one in Figure 5) will refer to the iteration variable whenever a check is placed in the loop. When computing a check for a location before the loop, the iteration variable of this loop is projected out (cf. Section 3.1.1).

4.2 Possible Check Locations

Now that we know how to derive in-bounds constraints as well as how to adapt them for different program locations, we will go into detail on which locations are *valid* and can therefore be used for the check placement.

4.2.1 Valid Locations Algorithm. Algorithm 1 computes the locations that are valid for a check. Given a list of memory-accessing instructions (*Accesses*), it computes valid check locations for each one individually (*LocsForAccesses*) (lines 2, 15).

ALGORITHM 1: Determine valid check locations for every access.

```
Input: Accesses: List of accesses
Output: LocsForAccess: Access → Valid Locations
1  LocsForAccess = {}
2  foreach Access ∈ Accesses do
3      ValidLocations = [Access.Location]
4      Loc = Access.Location
5      while Loc.hasDominator() do
6          Loc = Loc.getImmediateDominator()
7          ExLps = exitedLoops(Access.Location, Loc)
8          if |ExLps| > 0 and isLoopVariant(Access, ExLps) and noLoopBoundsAvailable(ExLps)
9              break
10         if blockingVariableDefined(Access.Location, Loc)
11             break
12         if callOnPath(Access.Location, Loc)
13             break
14         ValidLocations.add(Loc)
15     LocsForAccess[Access] = ValidLocations
16 return LocsForAccess
```

It is always possible to place a check right before the access, therefore, we insert this location into its list of valid locations (line 3).

A check should always be executed before the access that it guards is executed. Therefore, we consider only locations that are on a path from the entry of the function to the access as valid check locations. To ensure this, the algorithm starts with the location of the access (line 4) and considers only locations that dominate the access location (lines 5, 6). The conditionals in lines 7–13 ensure that only locations for which we can compute a precise check are considered for check placement.

The function *exitedLoops* called in line 7 computes all loops that surround the location of the access, *Access.Location*, but not the currently considered location *Loc*. Whenever we face an access that is variant in the loops that no longer surround the location *Loc* (*|ExLps| > 0 and isLoopVariant(Access, ExLps)* in line 8), the check has to account for all memory accessed within the loop, without referring to the loop induction variables. We require a Presburger formula that restricts the values of the loop induction variables[7] to precisely describe all accessed memory within the loops. If no such formula exists (*noLoopBoundsAvailable(ExLps)* in line 8, i.e., for non-affine loop variable increments), then we will not consider a location outside of the loops for the check placement.

The check of *blockingVariableDefined* (line 10) ensures that if a variable relevant for the access runs out of scope, then the check cannot be placed before its definition. A variable is relevant for

[7]As described in Section 3.1.1, Scalar Evolution provides this information in form of the backedge-taken count.

```
1  void init(int *A, int n) {
2    for (int i=n-1; i>=0; i-=2) {
3      if (i > 0)
4        A[i - 1] = 1;
5      A[i] = 0;
6      int sq = i*i;
7      if (sq < n)
8        A[sq] = 2;
9    }
10 }
```

Fig. 6. Initialization function.

the calculation if it is involved in the address calculation (as pointer value or as part of the index) or if it is used in a control-flow condition between *Access.Location* and *Loc*. Such blocking variables are:

- values loaded from memory,
- return values from function calls, and
- computations that are non-affine.

The Presburger formulas contain parameters for blocking variables, as their values can otherwise not be expressed precisely. As the parameters refer to the variables, only locations where the variables are defined can be considered for check placement.

A last limitation is that we will not move checks across function calls (line 12), since this can lead to false positives. If the call does not return, e.g., because `exit` is called, then the access after the call is not executed and it does not need to be in bounds.[8]

4.2.2 Valid Locations Example. Consider the example in Figure 6, which we use to discuss the results of Algorithm 1. Later on Algorithm 2 and the related work discussion (Figure 15) revisit it.

The function `init` takes a pointer `A` and an integer `n`. The function initializes the array pointed to by `A` with alternating zeros and ones. The alternating initialization is done by iterating backwards over the array with a stride of two and accessing `A` at `i` and `i-1`. Additionally, whenever `i*i` is smaller than `n`, `A[i*i]` is initialized to two.

Locations that are valid for a check are computed per access, therefore, we start with the valid locations for the access `A[i-1]` in line 4. First, a check at the access location is always possible, so the first valid location is line 4, we denote this as LocsForAccess[`A[i-1]`] = [4] in the following.[9] The next location that dominates the access location is line 3, which is within the same loop, therefore *|ExLps| > 0* is false. No used variables are defined on the paths from location 4 to 3, so there are also no blocking variables. The same holds for calls. Line 3 is therefore also valid to place a check. The next dominating location is line 2. This time, a loop is exited, and the access is loop variant as it uses `i`. However, a loop trip count ($\lfloor \frac{1+max(0,n)}{2} \rfloor$) is available, so we can precisely capture all accessed memory locations. Again, we do not have any blocking variables or calls. Hence, the final result is:

LocsForAccess[`A[i-1]`] = [4, 3, 2].

[8] A possible solution to this limitation is to employ a separate analysis that marks functions as will-return and use this information to allow for more check locations.

[9] We use the access expression `A[i-1]` as index into *LocsForAccess,* as the accesses in the example are different; in the implementation, we refer to the LLVM IR instruction so accesses that look the same in C are not conflicting as keys.

The valid locations for the access `A[i]` in line 5 are derived analogously and look as follows:

LocsForAccess[`A[i]`] = [5, 3, 2].

A check for the access `A[sq]` in line 8, however, can only be placed at locations [8, 7]. If we consider the dominating location of line 7, then we find that the variable `sq`, which is used in the control flow that leads to the access and in the access itself, is a non-affine calculation that we cannot capture in a Presburger formula.[10] We rely on the available expression for `sq` in the check and therefore consider no further locations valid for check placement.

Algorithm 1 computes all valid locations for a check of an access, as this enables more possible combinations in Section 4.4. We refer to the location highest up in the dominator tree that is valid for an access as the *top-most valid location* in the rest of the article.

4.3 Check Placement Strategy

To ensure the memory safety of an access, it must be guarded by a check on every path from the function entry to the access. Easy solutions to this problem are placing the check at the access or at the top-most valid location. Placing checks at the access location is done by memory safety instrumentations and incurs significant runtime overhead whenever this access is inside of a loop. A check for the top-most valid location might be significantly more complex than it is at the access, and it might be executed more frequently than the access.

Intuitively, we would like checks to be in as few loops as possible, without remodeling too much control flow (see Section 4.1). Our heuristic therefore estimates the cost for a check C at a location L as follows:

$$cost(L, C) := execFrequency(L) \cdot complexity(L, C).$$

Here, *execFrequency(L)* is the statically estimated number of executions of a program location L.[11] Note that this number does not necessarily reflect the actual runtime frequency, but gives an idea of the execution frequency of one location relative to another location within the same function. The function $complexity(Location, Check)$ approximates the computational overhead introduced by the given check placed at the given location. For this purpose, we compute the check for the given location and count the number of comparisons, arithmetic, and logical operations in the check.

To find a good check location using this cost metric, we adapt an idea from code-placement optimizations [Ebner et al. 2009; Xue and Knoop 2006] that place code by finding *minimum cuts* in a **control-flow graph (CFG)**. A minimum cut is a set of edges in a graph that, when removed, disconnects a given source and sink such that the cost of the edges in the cut is minimal.

In our setting, the cut is computed on the blocks of the CFG rather than the edges. To compute the block min-cut, the costs on the original CFG edges are set to infinity. Then each basic block is split into two vertices that are connected by an artificial inner-block edge with the cost of the check at that basic block. This ensures that only inner-block edges are considered for the resulting cut. The edges computed by the min-cut are the least expensive locations that guarantee that the check is executed on every path from the top-most valid location (source) to the access (sink).

As an example, consider the program in Figure 7. The function contains a memory access inside of the loop (line 10). A valid location for a runtime check is right before the loop in block D. The

[10] Note that our implementation works on LLVM IR level and not C, such that a multiplication will be a stand-alone instruction that can be referred to later on and it is not a requirement on the source code to contain such a temporary variable for non-affine expressions.
[11] We use the execution frequency analysis implemented in LLVM [LLVM 2020].

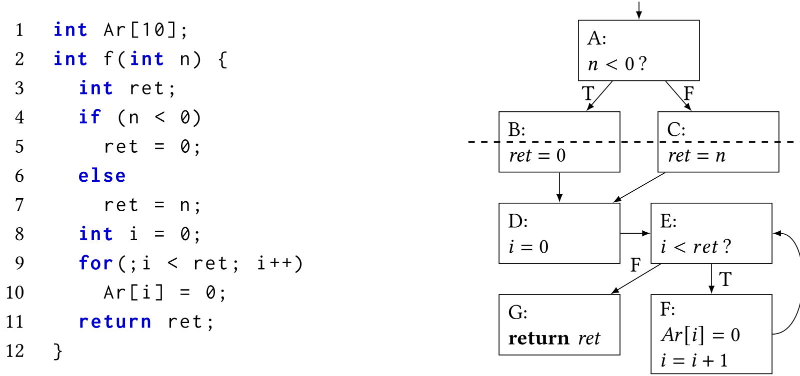

Fig. 7. A program (left) and its control-flow graph (right), the dotted line indicates the minimum cut.

check for this location is $n \leq 10$, as the array has size 10 and `ret` is either 0 or `n`. However, the algorithm chooses blocks B and C to place runtime checks. The execution frequency of block D is equal to the sum of the frequencies of B and C. Yet there is one advantage within the conditional. In block B, we know that $n < 0$ holds, which implies the condition $n \leq 10$, such that no check needs to be placed in block B. The cost for this check decreases to zero. The only check in the optimized version is therefore in block C.

4.4 Summarizing Accesses

It is common that an array used within a function is accessed at different program locations. In this section, we describe how we can compute a summarized check for multiple accesses.

Consider the first two accesses of Figure 6 again. The access information for the accesses `A[i-1]` and `A[i]` is given by:

$$\begin{aligned}
&\{Mem \mid Mem = A + (n-1) - 2 \cdot l_i - 1 && \wedge\ 2 \cdot l_i < n - 1 && \wedge\ l_i \geq 0\},\\
&\{Mem \mid Mem = A + (n-1) - 2 \cdot l_i && \wedge\ 2 \cdot l_i < n && \wedge\ l_i \geq 0\}.
\end{aligned}$$

To obtain a check for both accesses, we compute the union of the sets and then apply the same technique to generate a runtime check as described earlier for single accesses. The resulting constraints to check before the loop are:

$$n \leq 0 \vee (Lower \leq 0 \wedge Upper \geq n).$$

The disjunct $n \leq 0$ ensures that we do not report an error if the accesses are not executed. The other disjunct ensures that all accesses in the loop are within bounds. The lowest index accessed is 0, so the offset to the lower bound needs to be less than or equal to zero. The upper bound needs to be at least n.

Due to the limitations on check locations (cf. Section 4.2), we cannot simply check all accesses at once. Hence, we need a systematic way to determine accesses that should be checked together.

For an optimal solution of the summarization and placement with respect to the cost heuristic (cf. Section 4.3), the minimum cut for all possible partitionings of accesses into groups needs to be computed. A partitioning is a split of the original set into subsets, such that no set is empty, the sets are disjoint, and the join of all subsets yields the original set. How many of such partitionings exist was first formally described by Bell [1938] and is called the Bell number. For a given set of size n, the Bell number B_n is exponential in n. It would be infeasible to compute the minimum cut for all of these partitionings, which is why we employ a greedy algorithm to choose a partitioning.

Programs often access overlapping or neighboring addresses. When such accesses are checked together, the combined check is often as complex as one of the individual access checks.[12] To benefit from this property, we try to maximize the number of the accesses to the same array[13] that are checked together.

ALGORITHM 2: Determine accesses to check together.

```
Input: LocsForAccess: Access → Valid Locations
Output: Partitioning of the input
 1 Buckets = []
 2 foreach (Access, Locations) ∈ LocsForAccess do
 3     placed = False
 4     foreach Bucket ∈ Buckets do
 5         if Locations ∩ Bucket.CommonLocs ≠ ∅
 6             Bucket.CommonLocs = Locations ∩ Bucket.CommonLocs
 7             Bucket.Accesses = Bucket.Accesses ∪ Access
 8             placed = True
 9             break
10     if not placed
11         Buckets.addBucket(Access, Locations)
12 return Buckets
```

4.4.1 Summarization Algorithm. Algorithm 2 shows our greedy algorithm to determine a partitioning. The goal is to assign accesses to buckets such that the number of buckets is minimal. The input to the algorithm is a map that maps the information on an access to locations where checks for this access can be placed (computed by Algorithm 1). For each element of this map, the algorithm tries to find a fitting bucket. A bucket has a set of common locations, which represent this bucket (*Bucket.CommonLocs*). It additionally tracks all accesses sorted into this bucket (*Bucket.Accesses*). An access can be assigned to a bucket if the access has at least one common location with all the accesses that are already in the bucket. If the access has no overlapping locations with any bucket, then a new bucket is opened and the element's locations are the new representative for this bucket.

4.4.2 Summarization Example. Recall the results of Algorithm 1 applied to the `init` function in Figure 6:

$$\begin{aligned} \text{LocsForAccess}[A[i-1]] &= [4, 3, 2], \\ \text{LocsForAccess}[A[i]] &= [5, 3, 2], \\ \text{LocsForAccess}[A[sq]] &= [8, 7]. \end{aligned}$$

We can see that `A[i-1]` and `A[i]` have common locations, therefore Algorithm 2 will add them to the same bucket. The last access, `A[sq]`, has no overlapping location with the other two, thus it will have a separate bucket.

The placement of the check for each bucket of accesses is done using the technique that we described in Section 4.3 for a single access. As a sink to the minimum cut problem, we choose

[12] Consider Ar[0] and Ar[1] are accessed. Individual checks could be $Lower \leq 0 \wedge Upper > 0$ and $Lower \leq 1 \wedge Upper > 1$. Checking both at once can simply be done by checking $Lower \leq 0 \wedge Upper > 1$.

[13] More precisely, we try to summarize accesses to the same so-called *base pointer*, which might be a function argument, a pointer loaded from memory, or a similar source of a pointer.

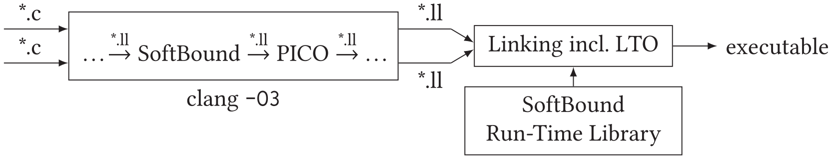

Fig. 8. Technical setup (LTO = link-time optimization).

the closest common ancestor of the accesses in the dominance tree. This ensures that the check is executed before any of the accesses happen.

Applied to the example, line 3 is the closest common ancestor of `A[i-1]` and `A[i]`. The location chosen by the minimum cut for this bucket is line 2, where the check contains the additional constraint $n \leq 0$, but this is outweighed by the estimated reduced execution frequency of line 2 compared to line 3. The chosen check location for $A[sq]$ is line 8, as both the complexity of the check and the expected execution frequency are lower there.

4.5 Profitability of Check Replacement

For a lot of memory safety instrumentations, the in-bounds checks themselves are not expensive when it comes to the number of instructions that they need. They consist of instructions that compare pointer values against a lower and upper bound pointer value [Nagarakatte et al. 2009] or of efficient arithmetic operations [Akritidis et al. 2009; Duck and Yap 2016]. The idea behind PICO's checks is not that they reduce the number of instructions for a check of a single access (although that might happen, cf. Figure 7 where we only need a single comparison), but rather that several accesses are checked at once at less frequently executed locations.

Our measure to decide whether the replacement is profitable is based on the cost heuristic presented in Section 4.3. A PICO check is profitable if the cost for the combined PICO check is lower than the sum of the individual checks of the instrumentation. More concisely (where *loc* returns the location of the check and *covered* the individual checks covered by a *combi*ned check):

$$\sum_{cutLoc \in mincut(combi)} cost(cutLoc,\ combi) < \sum_{c \in covered(combi)} cost(loc(c),\ c).$$

The combined checks are used whenever the formula evaluates to true.

5 EVALUATION

We use the memory-safety instrumentation SoftBound to generate a safe program, which we compare to the PICO-optimized safe program and the unsafe program with respect to the following criteria: Execution time (Section 5.2), binary size (Section 5.3), and compile time (Section 5.4).

To evaluate the additional value of the runtime check generation over only removing provably redundant checks, we show the results for the latter in Section 5.2.1. **Link-time optimization (LTO)** influences the results in several ways, which we discuss in Section 5.5. Last, we discuss limitations of PICO in Section 5.6.

5.1 Machine Information and Experimental Setup

Figure 8 shows the technical setup for the evaluation. The input C files are compiled separately and processed by SoftBound and PICO, which run as part of the `-O3` pipeline. PICO uses the Scalar Evolution pass available in LLVM as well as the open source implementation of Grosser et al. [2015] to generate code for the constraints we compute using the **integer set library ISL** [Verdoolaege

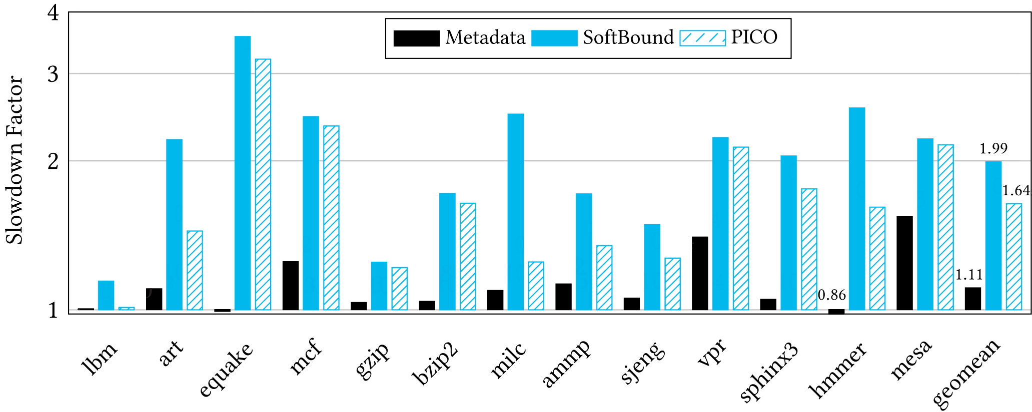

Fig. 9. Execution time overhead of SoftBound and PICO normalized to clang -O3 (lower is better).

2010]. The linker adds the SoftBound runtime library and applies **Link-Time Optimizations (LTO)** before generating the program executable.

All benchmarks were run on a 3.6 GHz AMD Ryzen 5 2600X with 32 GB RAM, running a 64-bit Arch Linux (Kernel version: 5.8.7). The presented numbers are the median of 21 runs executed sequentially. If not mentioned otherwise, then the presented speedups are statistically significant with a confidence level of 99% according to the median speedup test by Touati et al. [2013].

We used the most recent SoftBound+CETS version [Nagarakatte 2016] that is based on LLVM 3.9 for the evaluation. This is currently the best-performing memory safety instrumentation when it comes to safety and speed [Song et al. 2019, Table II]. It is configured to ensure spatial memory safety (which basically disables the CETS extension). Older versions of SoftBound featured a dominance-based spatial check elimination, which is no longer working. We fixed and reenabled it, but the performance of SoftBound was slightly worse with the optimization enabled (we observed a similar effect for some benchmarks when evaluating the removal of redundant checks with PICO; see Section 5.2.1). For the numbers we present, we therefore left this optimization disabled.

The presented benchmarks are selected from the SPEC CPU 2000 V1.3.1 [SPEC 2000] and 2006 V1.1 [Henning 2006] benchmark sets. The selection criterion was whether they worked with SoftBound and our optimization: In contrast to the original SoftBound [Nagarakatte et al. 2009] and WPBound [Ye et al. 2014][14] benchmark selection, `libquantum`, `twolf`, and `h264ref` did not properly execute under the publicly available SoftBound implementation. SoftBound reported a benign memory safety violation for `h264ref`; the others failed even though there was no violation.[15] The only benchmark we excluded because it reached the two-hour compile-time limit with PICO is `crafty`.

The benchmarks are presented in ascending order of **source lines of code (SLOC)**, ranging from 904 for `lbm` to 42,492 for `mesa`.

5.2 Runtime Performance

Figure 9 shows the slowdown of SoftBound and PICO normalized to plain LLVM, all of them were run with optimization level -O3. A factor of one means that the execution times are equal

[14] WPBound is discussed in the Related Work, Section 6.2.

[15] Due to those differences in the selection of benchmarks, the newer compiler and SoftBound version, as well as different hardware, our results deviate from those in the original paper [Nagarakatte et al. 2009].

to LLVM -O3, everything above is a slowdown. The slowdown factor is shown on a logarithmic scale.

The first bar, *Metadata*, is a configuration that SoftBound offers to measure the overhead of their metadata propagation. Metadata propagation places load- and store-metadata calls for bound information in the source code such that bounds are available for checking. Although metadata load calls are necessary when checks are inserted, their results are often unused when there are no checks. As LTO is enabled, the metadata calls to the SoftBound runtime library are inlined. At that point, the compiler is able to show that some of the metadata loads are unused and it optimizes them away. The *Metadata* bar therefore underestimates the actual cost of loading and storing metadata (we will discuss an example in Section 5.5).

The overhead of the *Metadata* configuration is low in most cases, with the exception of `vpr`, `mcf`, and `mesa`. In `mesa` it accounts for almost half of the overhead. Unexpectedly, we can see a reduction in the execution time of `hmmer` in the metadata case. In our setup (cf. Figure 8), SoftBound is run as part of the -O3 pipeline. This means that the metadata insertion has effects on passes that run afterwards, in a positive or negative way.[16]

PICO reduces the overhead of SoftBound in all cases. More than 50% of the overhead is removed in `lbm`, `milc`, `ammp`, `hmmer`, and `art`. In `lbm` and `milc` the overhead is almost fully removed. Overall, the geometric mean of the slowdowns decreases from 1.99× for SoftBound to 1.64× for PICO; this is an overhead reduction of 36%.

To verify the stability of the reported speedups, we additionally ran the benchmarks on a 3.3 GHz Intel Core i9-7900X with 64 GB of RAM running a 64-bit Arch Linux (Kernel version: 5.10.6). While the absolute execution times varied, the slowdowns showed a very similar picture. The average slowdowns were 1.99× for SoftBound and 1.70× for PICO.

5.2.1 Impact of the Removal of Redundant Checks. One particularly interesting question is how PICO fares in comparison to approaches that only focus on removing checks by static analysis. To enable such a comparison PICO features a configuration to only remove checks: *Remove-Only* (cf. Figure 2, the upper box), which disables check optimization and placement. We identified two metrics to evaluate the removal of checks that are commonly used in related work (cf. Section 6.2):

(1) The number of checks removed from the program.
(2) The runtime impact of the removal.

Figure 10 shows our result for both metrics. Each benchmark corresponds to two circles: A hollow one for the *Remove-Only* setting and a solid one for the full optimization presented in this article. The line that connects the two circles is a visual aid to make it easy to identify the circles of the same benchmark. The horizontal position denotes how many checks can be proven redundant statically (in percent of the total number of checks, cut at 70%). The further to the right a benchmark is, the more checks are removed. The vertical position describes by how much the execution-time overhead of SoftBound compared to -O3 is reduced, ranging from −10% to 100%. The higher a dot, the more the execution is sped up. The benchmarks `hmmer` and `mcf` do not show a statistically significant speedup.

If we focus on the *Remove-Only* configuration, then we can see that although in 6 out of 13 benchmarks more than 40% of checks can be proven safe at compile time, the execution-time impact is below 15% for all but `sjeng` (22%). Nine of the benchmarks reduce the overhead by less than 5% and three benchmarks even lose performance.

[16] We will usually expect this to be a negative effect, as the check calls are very effective optimization barriers.

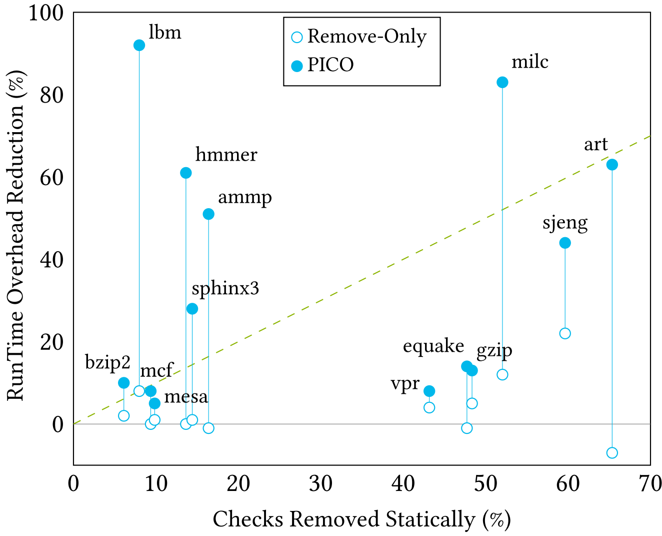

Fig. 10. Percentage of redundant checks removed (horizontal axis) and the impact of the removal on runtime (vertical axis).

Table 1. *#l/s:* number of loads and stores in the IR program, *IB:* accesses shown in bounds at compile time, *repl.:* SoftBound checks replaced with PICO checks

bench.	#l/s	IB	repl.
lbm	401	8%	87%
art	583	65%	12%
equake	1,012	48%	30%
mcf	630	9%	45%
gzip	1,607	48%	22%
bzip2	3,585	6%	21%
milc	3,707	52%	35%
ammp	5,130	16%	54%
sjeng	5,136	60%	26%
vpr	4,668	43%	14%
sphinx3	5,711	14%	35%
hmmer	11,752	14%	32%
mesa	22,185	10%	43%

Points on the diagonal would mean that each removal of a check leads to an equally high execution time gain. The results for the *Remove-Only* configuration show that this does not hold for our benchmarks; the execution time gain is usually much lower.

The gaps between the PICO and *Remove-Only* results show that large parts of the execution time improvements of PICO can be attributed to the placement of more efficient runtime checks rather than the removal of redundant checks.

5.2.2 Static Numbers on Removal and Replacement. Table 1 shows how many loads and stores are in the IR program (*#l/s*), how many of those are removed because they are shown in bounds at compile time (*IB*), and how many PICO replaces (*repl.*) for each benchmark (*bench.*).

The number of loads and stores ranges widely from 401 for `lbm` to 22,185 for `mesa`. The in-bounds column shows the values of the horizontal axis of Figure 10 more precisely. The leftover percentage 100%—*repl.*—*IB* is mostly deemed not profitable (cf. Section 4.5), or, in rare cases, a limitation applies (described in Section 5.6).

The numbers of removed and replaced checks vary even for well-performing benchmarks such as `lbm`, `art`, and `hmmer`: `lbm` has only very few redundant checks, namely, 8%, but PICO replaces most of the residual checks, 87%. In `art` it is the other way around: Most checks are removed without benefit (65%), while the additional replacement of 12% of the checks boosts the performance. In `hmmer` more than 60% of the overhead is eliminated, while less than 50% of the checks are removed or replaced. For `mesa` and `mcf`, however, the performance benefits are small although many checks are replaced. This again shows that the static numbers are hardly an indicator for the execution-time performance of the benchmarks.

5.3 Binary Size

Code instrumentation typically adds overhead to the size of the program binary. PICO removes instrumentation code by eliminating and summarizing runtime checks. Figure 11 shows that this decreases the overhead of the SoftBound instrumentation: The size of the PICO-optimized binary is smaller for each benchmark, reducing the average overhead from 3.58× to 2.97×. The average

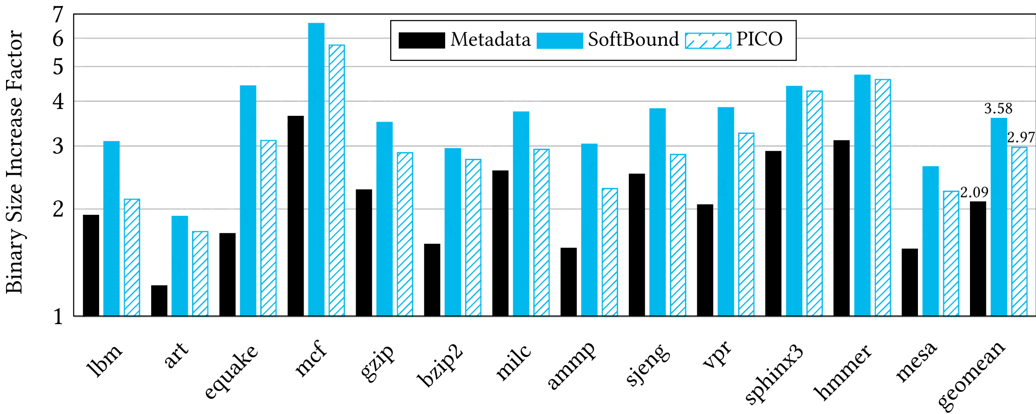

Fig. 11. Binary (.text) size of SoftBound and PICO normalized to clang -O3 (lower is better).

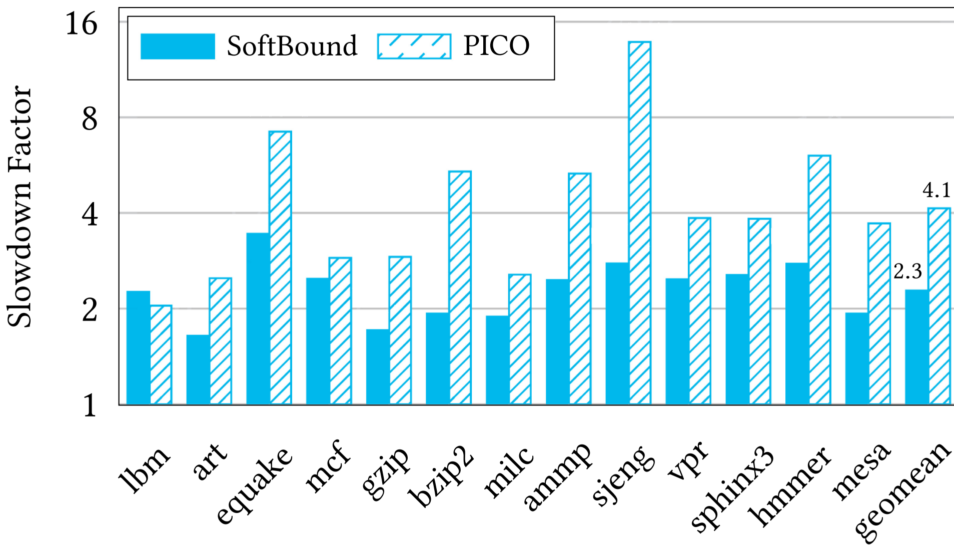

Fig. 12. Compile-time slowdown of SoftBound and PICO normalized to clang -O3 (lower is better).

overhead when only eliminating redundant checks (cf. Section 5.2.1) is 3.33×, which is considerably higher than for PICO. We conclude from this that the newly inserted checks are not only more execution-time efficient than the ones placed by SoftBound, but also require less space.

5.4 Compile-time Costs

Figure 12 shows the compile-time slowdown of SoftBound and PICO normalized to LLVM. The numbers are composed of compile and link time. Since PICO is executed in addition to SoftBound, PICO's compile time is expected to be always higher than that of SoftBound. The link time, however, might be influenced in a positive way: PICO removes SoftBound instrumentation and therefore avoids that the calls need to be inlined and further optimized. In `lbm`, where linking and compiling takes similarly long and PICO works particularly well, the overall time is improved. The

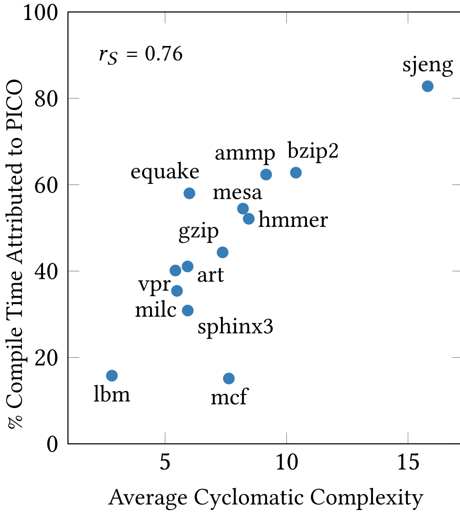

Fig. 13. Relating compile time to code complexity.

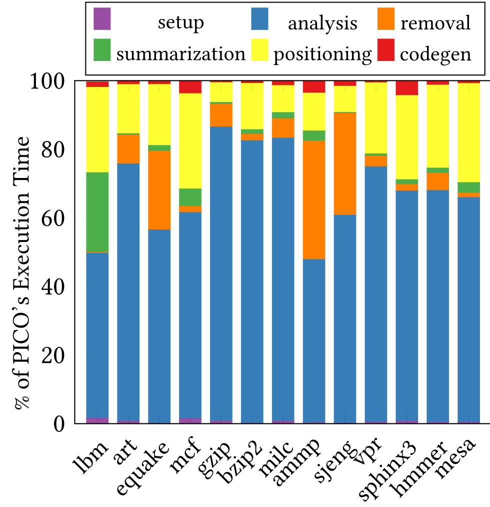

Fig. 14. Composition of PICO's execution time.

compile-time slowdowns are considerable, ranging up to 13.8× with an average of 4.1×; however, they are not intractable.

Our results show that the technique employed by PICO is applicable to sizable programs despite the doubly-exponential worst-case complexity of Presburger Arithmetic [Fischer and Rabin 1998]. In the rest of this section, we will further analyze which program properties impact the execution time of PICO (Section 5.4.1) and what it spends the time on (Section 5.4.2).

5.4.1 Correlation of Compile-time Increase to Program Properties. This section discusses benchmark characteristics that affect the compile time and how they correlate to PICO's contribution to the overall compile time. We investigated the number of instructions, basic blocks, and edges as well as the cyclomatic complexity [McCabe 1976] of the IR functions in each benchmark.

There is a strong correlation for the average cyclomatic complexity, the average number of basic blocks and edges, as well as the maximal number of those three measures. The correlation for the average and maximal number of instructions was weaker. We show the results for the average cyclomatic complexity in Figure 13. The benchmarks we evaluated are the data points; their horizontal position shows their average cyclomatic complexity. The vertical position indicates the compile time used by PICO as a fraction of the total compile time.

We can see the tendency that the higher the average cyclomatic complexity, the higher the percentage of compile time that can be attributed to PICO. We used the Spearman rank correlation coefficient (r_S) to determine how strong the monotonic relation is. The outcome of this correlation coefficient is in the interval $[-1, 1]$, where -1 indicates a perfect negative correlation, 0 no correlation at all, and 1 a perfect positive correlation. The result is a strong positive correlation of 0.76.

PICO employs a flow-sensitive analysis, which typically scales with the number of conditionals. Additionally, the cost of the min-cut computation increases the more locations are available. Hence, a correlation between the cyclomatic complexity and the increase in compile time is reasonable.

5.4.2 Execution-time Composition. Figure 14 shows how the individual components of PICO contribute to the overall execution time. The *setup* phase queries required analyses such as dominance information and is always well below 2% of the total time needed.

The second step, running the *analysis* to collect access information (cf. Section 3.1), always contributes the largest fraction of the execution time of PICO. It ranges from 48% for `ammp` to 86% for `gzip`. The analysis eagerly computes information on program variables and expressions, whether they are necessary for PICO later on or not.

The next phase, *removal*, computes which checks can be removed without a replacement. Checks can only be classified as redundant if the allocation site is statically known (cf. Section 3.2). The ratio of known allocation sites varies from 8% for `lbm` to 91% for `sjeng`. As a result, the cost for this phase also varies (see Figure 14), ranging from 1% (`lbm`) to 35% (`ammp`). Filtering out redundant checks early takes load off of the later phases, as they have to deal with fewer checks.

The *summarization* phase induces low overhead for all benchmarks but `lbm`. Due to the fact that only 8% of the instrumentation checks are shown to be redundant in this benchmark, 92% remain for the later phases. This increases the relative cost for the summarization. Our method to summarize accesses is quadratic in the number of accesses in the worst case, but does not apply costly algorithms on Presburger formulas. Hence, a low cost is expected.

The *positioning* phase calculates the min-cut and determines the final check constraints. It is expected that this takes up some time, as it calculates the constraints for different locations to estimate their cost. Overall, this phase takes up to 29% of the total time needed.

The *code generation* only needs a moderate amount of the total time, always less than 5%.

The compile-time composition shows that most time is spent in the program-analysis part. PICO would therefore profit most from a more demand-driven computation of the necessary information.

In addition to the contributions of the individual PICO phases, we measured how much time is spent in ISL, the library we use to manipulate Presburger formulas. ISL is used across multiple phases of PICO, and therefore to be considered orthogonally to the results shown in Figure 14. On average, 86% of the time required by PICO is spent in ISL when optimizing the benchmarks.

5.5 Disabling Link-time Optimization

We claimed earlier that the results of SoftBound show a different picture without LTO. The slowdown of SoftBound increases to 6.21× on average, for the same setup described in Figure 8, the only change being that LTO is disabled. PICO can reduce this to 2.98× (and Remove-Only to 5.76×). The binary size overhead decreases from 3.57× for SoftBound to 2.90× for PICO on average.

Due to LTO, measuring the cost of metadata propagation alone is not as straightforward as leaving out the checks and placing metadata load and store calls (cf. Section 5.2). We inspected the benchmark `equake`, where `-O3` and `-O3` + LTO perform equally well. The difference between *Metadata* with and without LTO for this benchmark is significant: With LTO, it performs as good as `-O3` (visible in Figure 9). Without LTO, it slows down by a factor of 4.6×. When inspecting this benchmark with perf,[17] one can see that 90% of the time is spent in a single call to load metadata. This call is within the hot loop of the benchmark and looks up bounds for a pointer that is loaded from memory. The overhead of this call is not visible in the LTO runs, as there it becomes evident to the compiler that the load result is unused and it is therefore removed.

Our static analysis, as most related approaches, does not track memory content and therefore cannot avoid the load of metadata at that location. Benchmarks with similar characteristics as `equake` are hard to optimize and will likely suffer significant slowdowns, even after optimization.

[17] `perf` is a popular Linux profiling tool, for more information see https://perf.wiki.kernel.org/index.php/Main_Page.

5.6 Limitations

Presburger formulas work on mathematical integers and do not account for the wrapping behavior of machine integers with limited bit width automatically. We model unsigned integers using modulo operations such that their defined behavior is preserved. Signed integer overflow, however, has no defined semantics in C. Our optimization does not include countermeasures to detect signed integer overflow. PICO could be extended using an existing technique by Doerfert et al. [2017] or complemented with an external tool as presented by Long et al. [2014] to detect these overflows.

SoftBound does not guarantee memory safety for variable argument functions. We inherit this limitation from SoftBound, as our checks rely on the instrumentation to provide bounds. The benchmarks `sphinx3` and `hmmer` define variable argument functions, which are hence unchecked. In both cases, the variable argument functions contain less than 0.5% of the total number of accesses.

The value analysis PICO uses does not work for functions with irreducible control flow. The benchmark `bzip2` contains such functions, which remain unoptimized (44% of the accesses).

PICO is designed to work on C programs where accessing an array outside of its bounds is undefined behavior. This includes that there is no specified point in the program at which an error should be reported for an out-of-bounds access. We utilize this to find cheap locations for error reporting and to simply abort the program at runtime in case an out-of-bound access is detected. Due to this assumption, our approach is not straightforwardly applicable to languages such as Java, where the semantics precisely define the point at which a specified exception has to be thrown.

6 RELATED WORK

We first discuss related work in the area of memory safety instrumentations for the C language and motivate why we focused our research on optimizing compiler-based instrumentations. The second part discusses other optimizations for memory safety instrumentations.

6.1 Memory Safety Instrumentations

Memory safety for the C programming language has been a topic of extensive research for years. We can classify existing memory safety approaches into three categories: those that require source language or code adaptations, binary instrumentations, and compiler-based techniques.

Some efforts were made to modify the programming language, such that memory safety violations can be ruled out efficiently [Grossman et al. 2005; Rust 2020]. These techniques require rewriting existing C code to match their restricted language. In case of Cyclone [Grossman et al. 2005], an average of 6% of the code had to be rewritten to port it from C to Cyclone. Restrictions introduced by Cyclone such as always initialized pointers, constraints on pointer arithmetic, and gotos were part of this rewriting. They introduce runtime bound information only for pointers used in pointer arithmetic to reduce bound checking overhead. For computationally intense benchmarks, the authors report slowdowns of a factor of 6 (on average, roughly 2). The practicability of approaches modifying the language seems to be small; in huge code bases even changing 6% of the lines of code is a significant investment.

Binary instrumentations such as Dr. Memory [Bruening and Zhao 2011] and Valgrind [Nethercote and Seward 2007] do not even require recompilation of the program. To find memory errors or memory leaks, these tools have proven to be helpful, but they come with slowdowns of 10.2× in case of Dr. Memory and 20.4× for Valgrind on average [Bruening and Zhao 2011]. While substantially easier to use, the slowdowns are very large and they only give limited safety guarantees. They do not capture violations such as out-of-bounds pointers that happen to point to a different

object after going out of bounds, as they cannot correlate pointers to the object they are derived from.

The last class of approaches is code instrumentations that report an error if a memory safety violation occurs during the execution of the program [Akritidis et al. 2009; Dhurjati and Adve 2006; Duck and Yap 2016; Jones and Kelly 1997; Nagarakatte et al. 2015; Necula et al. 2002; Ruwase and Lam 2004; Tarditi et al. 2018]. These approaches insert runtime checks at memory accesses or pointer arithmetic and come with a runtime system to keep track of allocations that are necessary for the safety checks. The instrumentation is done automatically during compilation of the program, without the need for the programmer to adapt the code. Many of the instrumentations also work in the presence of uninstrumented libraries [Dhurjati and Adve 2006; Duck and Yap 2016; Jones and Kelly 1997; Ruwase and Lam 2004], though some approaches need to wrap calls to these libraries [Akritidis et al. 2009; Nagarakatte et al. 2015; Necula et al. 2002] to avoid false alarms.

Manual source code modifications and binary instrumentation tools do not seem to be promising approaches to fulfill the needs of giving ample safety guarantees and being easy to use at the same time. The compiler-based approaches give strong guarantees and are fully automated; however, the execution time overhead is non-negligible. PICO is therefore designed to reduce the overhead of these otherwise promising approaches.

6.2 Optimizations of Memory Safety Instrumentations

First, we discuss related work in static analysis, which improves memory safety instrumentations by removing redundant checks. The second part covers approaches that compute runtime checks, but that do not provide strategies for positioning these checks. The last part takes a closer look at approaches that compute additional runtime checks and use these checks to avoid the execution of memory safety checks placed by an instrumentation.

Redundant Check Removal. Static analyses are often used to prove accesses in bounds, such that they can be elided from the program without losing safety guarantees.

Nazaré et al. [2014] describe an analysis based on symbolic ranges. This analysis domain differs from the classical interval domain in that it allows for general expressions over variables in the lower and upper part of the range. Their approach collects information on allocations using a region analysis. For soundness, the information on accessed memory ranges is approximated such that it is larger than the actual access, whereas the allocated regions are approximated to be smaller. Using the information on allocated memory regions and accessed memory regions interprocedurally, they compute whether a check is in bounds. They show 43% of the accesses to be safe (without the taint analysis, which lowers the safety guarantees) on average in the CINT SPEC2006 benchmarks.

Bodík et al. [2000] and Logozzo and Fähndrich [2008] present static analyses that remove bounds checks in memory-safe languages. ABCD [Bodík et al. 2000] works in the dynamic compilation context of Java, hence they use a lightweight analysis domain. Due to their dynamic setting, they can identify frequently executed checks and try to optimize these checks on demand. The domain underlying their analysis is tracking inequality relations between program variables that are relevant for memory accesses. By combining these relations with information on the array length field, they build a graph of inequalities. With this graph, redundant checks can be identified. This can be used to efficiently eliminate in-bounds checks, though the expressiveness of their analysis domain is rather restricted. Logozzo and Fähndrich [2008] use a more elaborate domain, pentagons, that equips the domain of inequalities of Bodík et al. [2000] with value ranges. Their results show that they are very effective in reducing the number of checks in four .NET assemblies, where they can

```c
void init(int *A, int n) {
  /* BaseA + BoundA def */
  int wp1 = wpChk(A-1, A+n-1, BaseA, BoundA);
  int wp2 = wpChk(A, A+n, BaseA, BoundA);
  for (int i = n-1; i >= 0; i -= 2) {
    if (i > 0) {
      if (wp1)
        sbChk(A+i-1, BaseA, BoundA, sizeof(int));
      A[i - 1] = 1;
    }
    if (wp2)
      sbChk(A+i, BaseA, BoundA, sizeof(int));
    A[i] = 0;
  }
}
```

```c
void init(int *A, int n) {
  /* BaseA + BoundA def */
  ptrdiff_t LB =
        (intptr_t) BaseA - (intptr_t) A;
  ptrdiff_t UB =
        (intptr_t) BoundA - (intptr_t) A;
  assert(n <= 0 ||
    (LB <= 0 && UB >= 4*n));

  for (int i = n-1; i >= 0; i -= 2) {
    if (i > 0)
      A[i - 1] = 1;
    A[i] = 0;
  }
}
```

Fig. 15. First two accesses from Figure 6 optimized with WPBound (left) and PICO (right).

eliminate 89% of all checks. Nazaré et al. [2014] showed that this number decreases significantly to 27.2% on the CINT SPEC2006 benchmarks.

Due to the differences in tools (e.g., the Jalapeño Java Compiler [Burke et al. 1999] in the case of ABCD and Clousot [Fähndrich and Logozzo 2010] in the case of Pentagons) and benchmarks (e.g., .NET assemblies, Java codes, and SPECINT benchmarks unsupported by SoftBound), we cannot directly compare against these approaches. However, Section 5.2.1 showed that even if substantial parts of checks can be shown in bounds statically, the runtime gains in our setup and on our benchmarks were rather small. The replacement of checks, which we found most effective to reduce runtime overhead, is not supported by these tools.

Check Generation without Check Placement. Popeea et al. [2008] present a static analysis approach using Presburger arithmetic that can be used to remove redundant checks and to place runtime checks at memory accesses. They first translate programs to a simple language called *IMP* that implements loops as recursive functions and allows to access memory through arrays. They derive dependent types for the translated program. The dependent types encode array sizes in the types of arrays and are used to derive that accesses are in bounds. They show that they can successfully prove all checks in bounds for 12 out of 14 benchmarks (including bubble sort, hanoi tower, Linpack, and FFT) and 75% on the remaining two. The approach of Popeea et al. [2008] is a full memory safety instrumentation for a subset of programs that can be translated to the *IMP* language. They do not report the overhead introduced by this transformation and the additional dependent type information that needs to reside in the program to perform runtime checks. The results from Cyclone [Grossman et al. 2005] show that translating C programs to a safer subset can already increase the execution time by a lot even without checking for out-of-bounds accesses. Our approach can summarize accesses and place checks at different locations in the program, whereas the checks computed by Popeea et al. [2008] will always be right before the corresponding access.

The approach of Dillig et al. [2014] allows the programmer to specify locations at which checks for spatial memory safety are generated. The checks fail whenever executing the guarded region would lead to an out-of-bounds access. The regions that can be encapsulated by these checks are limited to not include calls and uses of values loaded from memory. These constraints guarantee that expressions are recomputable, which is necessary to compute precise checks at a user-specified location. The check placement is not dealt with automatically but is to be done by the programmer.

In contrast to these two approaches, PICO automatically computes well-suited check locations, allows for limited program knowledge, and scales to large, real-world applications.

```
bool wpChk(char *p_lb, char *p_ub,
           char *p_bs, char *p_bd) {
  return p_lb < p_bs || p_ub > p_bd;
}
```

Fig. 16. WPBound check [Ye et al. 2014].

Optimizations Introducing Fast Paths. Some memory safety instrumentation optimizations precompute sufficient conditions for accesses within loops to be in bounds, such that a fast path can avoid the execution of checks within the loop at runtime. Akritidis et al. [2009] proposed this idea as an optimization for their memory safety instrumentation. Ye et al. [2014] implement this optimization (that they call WPBound), extend it, and present a full evaluation.

Figure 15 (left) shows the example from Figure 6 (for simplicity without the third access) when instrumented with WPBound. The precomputed `wpChks` evaluate to false whenever the access is in bounds (see Figure 16). If the check evaluates to true, then there may be an out-of-bounds access and an additional precise SoftBound check is executed. If this function is used to initialize an array whose allocation starts at `A` and has `n` elements, then `wp1` evaluates to `true` and `wp2` evaluates to `false`. The fail-safe SoftBound check for `A[i-1]` will always be executed, whereas the SoftBound check for `A[i]` can be avoided entirely.

WPBound reduces the overhead of SoftBound from 1.71× to 1.45× on selected SPEC CPU 2000/2006 benchmarks. WPBound works on a no longer available SoftBound version with an older compiler version and uses -O2 as its baseline. Its code is not publicly available, we can therefore not directly compare against WPBound. The most notable difference is visible in `sjeng`. As seen in the example, it is possible that the WPBound check is executed in addition to the SoftBound check, which can lead to higher instead of lower execution times. WPBound slows down `sjeng` from 1.55× to 1.75×, while PICO decreases the execution time overhead from 1.49× to 1.27×.

Figure 15 (right) shows the `init` function optimized by PICO. The (byte-precise) PICO check (shown as assertion) is true if and only if all accesses it guards are valid. No fallbacks are required, which enables it to remove the `sbChk` calls. By construction, WPBound increases the bit code size (from 1.72× for SoftBound to 2.12× for WPBound on average reported in their paper). PICO however, decreases the binary file sizes substantially from 3.58× for SoftBound to 2.97× on average.

7 CONCLUSIONS

We presented PICO, an optimization for compiler-based memory safety instrumentations. PICO computes Presburger formulas for in-bounds conditions of memory accesses. In case the formula is valid, the access is proven in bounds. Otherwise, it can be used as a runtime check for the access. PICO can synthesize different variants of a check for different code locations and summarize checks of multiple accesses into one. Our experimental evaluation shows that we can significantly reduce the execution time and code size overhead of SoftBound, a state-of-the-art memory safety instrumentation for C, at the cost of increased compile time. Furthermore, we found that these improvements can mostly be attributed to our replacement of checks that depend on runtime values, rather than the simple removal of redundant checks.

REFERENCES

Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *SSYM'09*. 51–66. https://dl.acm.org/citation.cfm?id=1855772.

Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. 1994. Chains of recurrences—A method to expedite the evaluation of closed-form functions. In *ISSAC'94*. 242–249.

E. T. Bell. 1938. The iterated exponential integers. *Ann. Math.* 39, 3 (1938), 539–557. http://www.jstor.org/stable/1968633.

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating array bounds checks on demand. In *PLDI'00*. Association for Computing Machinery, New York, NY, 321–333. https://doi.org/10.1145/349299.349342

Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *CGO'11*. IEEE Computer Society, 213–223. https://doi.org/10.1109/CGO.2011.5764689

M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 1999. The jalapeño dynamic optimizing compiler for Java. In *JAVA' 99*. ACM, 129–141. https://doi.org/10.1145/304065.304113

Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE'06*. 162–171. https://doi.org/10.1145/1134285.1134309

Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. 2014. Optimal guard synthesis for memory safety. In *Computer-aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 491–507.

Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic loop optimization. In *CGO'17*. IEEE, 292–304. https://doi.org/10.1109/CGO.2017.7863748

Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *CC'16*. 132–142. https://doi.org/10.1145/2892208.2892212

Dietmar Ebner, Bernhard Scholz, and Andreas Krall. 2009. Progressive spill code placement. In *CASES'09*. ACM, New York, NY, 77–86. https://doi.org/10.1145/1629395.1629408

Robert van Engelen. 2001. Efficient symbolic analysis for optimizing compilers. In *CC'01*. Springer, London, UK, 118–132. http://dl.acm.org/citation.cfm?id=647477.727776.

Manuel Fähndrich and Francesco Logozzo. 2010. Static contract checking with abstract interpretation. In *FoVeOOS'10*. Springer, Berlin, 10–30. https://doi.org/10.1007/978-3-642-18070-5_2

Alain Finkel and Jérôme Leroux. 2002. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST TCS'02*. Springer, Berlin, 145–156. https://doi.org/10.1007/3-540-36206-1_14

Michael J. Fischer and Michael O. Rabin. 1998. Super-exponential complexity of presburger arithmetic. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 122–135. https://doi.org/10.1007/978-3-7091-9459-1_5

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly—Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Proc. Lett.* 22, 4 (2012), 27. https://doi.org/10.1142/S0129626412500107

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation is more than scanning polyhedra. *ACM Trans. Prog. Lang. Syst.* 37, 4 (July 2015). https://doi.org/10.1145/2743016

Dan Grossman, Michael Hicks, Trevor Jim, Greg Morrisett, James Cheney, and Yanling Wang. 2005. Cyclone: A type-safe dialect of C. *C/C++ Users J.* 23, 1 (2005), 112–139.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG'97*. 13–26.

Tina Jung. 2015. *A Hybrid Approach for Parametric Memory Dependence Analysis*. Bachelor's Thesis. Saarland University.

Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: Buffer overflow checks without the checks. In *EuroSys'18*. ACM. https://doi.org/10.1145/3190508.3190553

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

LLVM. 2020. LLVM Block Frequency Analysis. https://llvm.org/doxygen/classllvm_1_1BlockFrequencyInfoImpl.html#details.

Francesco Logozzo and Manuel Fähndrich. 2008. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *SAC'08*. ACM, New York, NY, 184–188. https://doi.org/10.1145/1363686.1363736

Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound input filter generation for integer overflow errors. In *POPL'14*. ACM, New York, NY, 439–452. https://doi.org/10.1145/2535838.2535888

T. J. McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

MITRE Corporation. 2019. Top 25 Most Dangerous Software Errors 2019, Common Weakness Enumeration CWE. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

Santosh Nagarakatte. 2016. SoftBound + CETS for LLVM-3.9. https://github.com/santoshn/SoftBoundCETS-3.9.

Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything you want to know about pointer-based checking. In *SNAPL'15*, Vol. 32. Dagstuhl, Germany, 190–208. https://doi.org/10.4230/LIPIcs.SNAPL.2015.190

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI'09*. ACM, 245–258. https://doi.org/10.1145/1542476.1542504

H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira. 2014. Validation of memory accesses through symbolic analyses. In *OOPSLA'14*. ACM, New York, NY, 791–809. https://doi.org/10.1145/2660193.2660205

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *POPL'02*. 128–139. https://doi.org/10.1145/1065887.1065892

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Not.* 42, 6 (2007), 89–100. https://doi.org/10.1145/1250734.1250746

Sebastian Pop, Albert Cohen, and Georges-André Silber. 2005. Induction variable analysis with delayed abstractions. In *HiPEAC'05*. Springer, Berlin, 218–232. https://doi.org/10.1007/11587514_15

Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. 2008. A practical and precise inference and specializer for array bound checks elimination. In *PEPM'08*. ACM, New York, NY, 177–187. https://doi.org/10.1145/1328408.1328434

William Pugh. 1994. Counting solutions to Presburger formulas: How and why. In *PLDI'94*. ACM, New York, NY, 121–134. https://doi.org/10.1145/178243.178254

Rust. 2020. Rust language manual. https://www.rust-lang.org/.

Olatunji Ruwase and Monica S. Lam. 2004. A practical dynamic buffer overflow detector. In *NDSS'04* . 159–169. https://doi.org/10.1145/780822.781150

D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. 2019. SoK: Sanitizing for security. In *SP'19*. IEEE, 1275–1295. https://doi.org/10.1109/SP.2019.00010

Standard Performance Evaluation Corporation SPEC. 2000. SPEC CPU 2000 benchmarks. https://www.spec.org/cpu2000/.

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Xiaodong Song. 2013. SoK: Eternal war in memory. In *S&P'13*. 48–62. https://doi.org/10.1109/SP.2013.13

David Tarditi, Archibald Samuel Elliott, Andrew Ruef, and Michael Hicks. 2018. Checked C: Making C safe by extension. In *SecDev'18*. IEEE, 53–60. https://doi.org/10.1109/SecDev.2018.00015

Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. 2013. The Speedup-Test: a statistical methodology for programme speedup analysis and computation. *Concurr. Comp. Pract. Exp.* 25, 10 (2013), 1410–1426. https://doi.org/10.1002/cpe.2939

Sven Verdoolaege. 2010. *isl*: An integer set library for the polyhedral model. In *ICMS'10*. Springer, Berlin, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

Jingling Xue and Jens Knoop. 2006. A fresh look at PRE as a maximum flow problem. In *Compiler Construction*, Alan Mycroft and Andreas Zeller (Eds.). Springer, Berlin, 139–154. https://doi.org/10.1007/11688839_13

Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBOUND: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *ISSRE'14*. IEEE, Washington, DC, 88–99. https://doi.org/10.1109/ISSRE.2014.20

Received September 2020; revised February 2021; accepted April 2021