Source-free, Machine-checked Validation of Native Code in Coq

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

Dakota Fisher
The University of Texas at Dallas
djf180000@utdallas.edu

Gilmore R. Lundquist
The University of Texas at Dallas
grl082000@utdallas.edu

ABSTRACT

Picinæ is an infrastructure for machine-proving properties of raw native code programs without sources within the Coq program-proof co-development system. This facilitates formal reasoning about binary code that is inexpressible in source languages or for which no source code exists, such as hand-written assembly code or code resulting from binary transformations (e.g., binary hardening or debloating algorithms). Preliminary results validating some highly optimized, low-level subroutines for Intel and ARM architectures using this new framework are presented and discussed.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Software reverse engineering*; • **Security and privacy →** *Logic and verification.*

KEYWORDS

formal methods, binary validation, automated theorem proving

ACM Reference Format:
Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. 2019. Source-free, Machine-checked Validation of Native Code in Coq. In *3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST '19), November 15, 2019, London, UK*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3338502.3359759

1 INTRODUCTION

Humans are notoriously error-prone when it comes to reasoning about code. For example, despite aggressive vetting by the open-source community, bugs in the Linux kernel persist for an average of almost 2 years before they are finally discovered [10]. An unprecedented 16,555 CVEs were tabulated by MITRE in 2018 alone.[1]

Reasoning about raw native code is even more difficult than analyzing code at the source level. While source code is designed to be human-readable, native code is designed to be machine-readable and is hence often unintuitive to humans. In particular, disassembled machine code typically lacks any structured control-flow idioms, and each assembly operation potentially implements a host of obscure side-effects on the state of the machine. For example, despite having no explicit parameters at the assembly level, the Intel x86 `AAD` instruction sets the `AL` register to $\texttt{AL}' := (\texttt{AL} + 10 * \texttt{AH}) \bmod 2^8$, zeros the `AH` register, replaces the low 16 bits of overlapping registers `AX` and `EAX`, assigns `SF` the 8th bit of $\texttt{AL}'$, sets or clears the `ZF` flag depending on whether $\texttt{AL}' = 0$, and sets or clears the `PF` flag depending on whether the number of 1's in the binary representation of $\texttt{AL}'$ is even or odd. Any or all of these effects can impact the behavior of subsequent instructions.

To obtain high confidence about the correctness and safety of complex code, *machine-checked formal methods* have long been championed as providing the highest attainable level of assurance for code validation. Rather than relying upon error-prone manual inspection of code, or upon semi-automated spot-checking regimens (e.g., fuzzing) that do not exhaustively cover the program's state space (which is often infinite), formal methods approaches construct machine-checked proofs that are universally quantified over the state space, and that explicitly formalize all assumptions. For example, the Coq program-proof co-development environment has been used to construct the only C-compiler [7] in which compiler bug-checkers (e.g., Csmith [14]) could find no errors.

Formal methods approaches typically entail developing proofs of source-level code properties, which are then reflected down to the object and native code levels by a *certifying compiler*—a compiler for which there exists a machine-checkable proof of *semantic transparency* assuring that semantic source-level properties are preserved by compilation (e.g., [7]). However, these top-down approaches cannot prove properties of codes for which sources do not exist or are unavailable. Mission-critical, source-free codes abound, including

- low-level binary runtime libraries (which often derive in part from hand-written assembly code) to which most source codes link at compile- or load-time;
- binary code resulting from low-level code transformations, such as VM instrumentation, source-free control-flow integrity [13, 16], program shepherding [6], or binary debloating [5, 11]; and
- closed-source libraries and components, which are building blocks for a majority of commodity software.

This paper introduces Picinæ (Platform In Coq for INstruction-level Analysis of Executables): a new infrastructure for developing machine-checkable proofs of native code behavioral properties without source code. Picinæ implements the syntax and semantics of an ISA-general intermediate language (IL) in Coq, to which a variety of native code languages can be lifted. To facilitate lifting, a small plug-in for the Binary Analysis Platform (BAP) [1] automatically translates BAP IL code into Picinæ IL, allowing the system to support all architectures currently supported by BAP. Once lifted, theorems and machine-checkable proofs can be developed in Coq to formally validate behavioral properties of the code, such as calling convention adherence, termination, safety (avoidance of "bad" states), and even full correctness.

[1] www.cvedetails.com/vulnerability-list/year-2018/vulnerabilities.html

FEAST '19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6834-6/19/11…$15.00
https://doi.org/10.1145/3338502.3359759

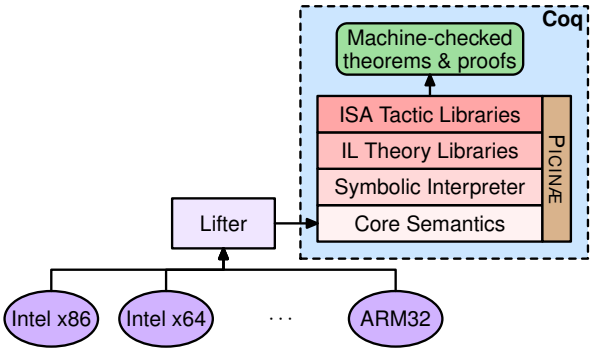

Figure 1: Picinæ workflow

$$q ::= 0 \mid v := e \mid \mathsf{J}\ e \mid \mathsf{X}\ i \mid q_1; q_2 \mid e\ ?\ q_1 : q_2 \mid q \times e$$
$$e ::= v \mid n_{\boxed{w}} \mid e_1[e_2]_{\boxed{w}} \mid e_1[e_2] \overset{\boxed{w}}{\longleftarrow} e_3 \mid e_1\ op\ e_2 \mid *_{\boxed{w}}$$
$$op ::= \oplus \mid \ominus \mid \otimes \mid \ll \mid \gg \mid \cdots$$

Figure 2: IL syntax (abbreviated)

As a preliminary case-study, we used Picinæ to machine-validate three heavily optimized subroutines from the GNU C standard libraries for Intel x86 and ARM. Our experiences show that although native code formal validation remains difficult, Picinæ's infrastructure affords sufficient expressive power to build machine-checkable proofs about code that is beyond the scope of traditional top-down formal methods.

2 OVERVIEW

Figure 1 illustrates Picinæ's high-level workflow. Raw native code is first lifted to an IL data structure expressed as a `.v` file that is directly readable into Coq. The lifter is not a full disassembler; it exhaustively decodes all valid opcode byte sequences in executable segments of the target program (or, optionally, within a specified address range) without attempting to decide whether the decoded instructions are reachable or aligned. This avoids problems associated with disassembly undecidability. Coq theorems can later declare (and prove) reachability properties of particular instruction sequences under particular conditions.

Lifted code is formalized in Coq as a partial function from addresses a to pairs (sz, q), where sz is the size of each instruction's encoding and q is an IL code block that details the instruction's effect upon the abstract machine state. Figure 2 summarizes this IL syntax, which closely resembles the syntaxes of ILs employed by related works [1, 2] to formalize ISA operational semantics. Specifically, IL statements q are no-operations 0, assignments :=, jumps J, numbered exceptions X, sequences $q_1; q_2$, conditionals $e\ ?\ q_1 : q_2$, or repetitions $\times$. Using repetitions in lieu of general while-loops guarantees that individual statements (and therefore individual instructions) always terminate. This avoids many spurious proof obligations related to termination. If an ISA includes an infinitely looping instruction, it can be encoded as a self-jump J.

Expressions are effect-free and consist of state element reads v, constants n, memory-reads $e_1[e_2]$, memory-writes $e_1[e_2] \leftarrow e_3$, modular binary operations op, and unpredictable outputs $*$. Memories are encoded as immutable array values; a memory-write returns a new array that is identical to array e_1 except with index e_2 assigned value e_3. Unpredictable expressions $*$ model instructions that have unspecified effects. For example, on Intel architectures the `xor` instruction has an unpredictable effect upon the adjust flag `AF`. Its IL encoding therefore contains assignment `AF := *`. Most expression forms are also parameterized by a bit width $\boxed{w}$, e.g. to distinguish a 32-bit zero from a 64-bit zero, and to facilitate modular arithmetic operations.

```
32 ↦ (2, AL := AL ⊕ AH ⊗ 10[8]; AH := 0[8];
         AX := AL; EAX := EAX ≫ 16[32] ≪ 16[32] ⊕ AX;
         SF := AL ≫ 7[8];
         ZF := AL ? 1[1] : 0[1];
         PF := ··· ; OF := *[1]; AF := *[1]; CF := *[1])
33 ↦ (2, ...)
34 ↦ (1, ESP := ESP ⊕ 4[32];
         J M[ESP ⊖ 4[32]][32])
```

Figure 3: Intel x86 program `AAD;RET` starting at address 32 lifted to Picinæ IL

As an example, Figure 3 lifts a two-instruction Intel x86 program consisting of an `AAD` instruction located at address 32 followed by a `RET` instruction at address 34. The lifted IL contains an additional instruction at address 33 because instruction encodings are unaligned in this ISA, and the final byte of the `AAD` encoding paired with the 1-byte `RET` opcode encodes a valid 8-bit `OR` instruction. Proving a property about this program can entail proving that this aliased instruction is unreachable, or proving that the property holds irrespective of whether it is reachable. The `AAD` instruction assigns to the `AL` and `AH` registers, modifying `AX` and `EAX` as a result, and changes a variety of status flags as side-effects, including unpredictable effects upon the `OF`, `AF`, and `CF` flags. The `RET` instruction increments stack pointer `ESP` by 4 and jumps to the value of the return address loaded from memory `M`.

Once the target program has been lifted to IL, it is loaded into Coq as a `.v` file in conjunction with a series of Picinæ libraries to build a suitable theorem-proving environment. In general, Picinæ's implementation can be stratified into four levels:

(1) Picinæ's *core* defines the IL syntax and its basic operational semantics.
(2) The *symbolic interpreter* elaborates these core definitions to build a more efficient transition system for the IL, which can be used within proofs to infer each successive machine state within a program being analyzed.
(3) A collection of *theory libraries* proves foundational properties of the IL that serve as building blocks for constructing proofs of program properties.
(4) At the highest level, a suite of ISA-specific *tactic libraries* automate and streamline common-case proof steps and notations particular to each architecture.

Together, these form a foundation for defining and proving machine-checkable theorems about source-free native code subroutines. In general, any property expressible in Coq's logic [3] can be reasoned about. This includes temporal properties of program

traces (e.g., LTL [4]), properties that quantify over sets of possible traces (e.g., CTL [12]), and even theorems that abstract all or part of the lifted code as an unknown with arbitrary content.

At its current stage of development, Picinæ does not model multicore concurrency; it is intended to validate properties of individual threads of synchronously executed instructions. However, it supports self-modifying code, dynamically changing memory access permissions, and input non-determinism, such as clock-reads or random number generation, which are expressed using $*$.

3 TECHNICAL APPROACH

3.1 Core Definitions

Picinæ's core definitions specify the IL's basic syntax and operational semantics in about 200 lines of Coq code. This part of the implementation is designed to be small, since it is the trusted foundation on which all other layers depend.

The operational semantics are defined in terms of three inductive propositions: one for expressions, one for IL statements, and one for whole-program execution. Expressions judgments $\langle e, \sigma\rangle \Downarrow u$ are large-step, and evaluate an expression e to a value u in the context of a machine state σ. Statement judgments $\langle q, \sigma\rangle \curvearrowright \langle \sigma', \chi\rangle$ are also large-step, computing the machine state σ' that results from executing statement q in state σ, along with an exit status χ (*viz.*, fall-through $\downarrow$, jump-to-address $\downarrow_a$, or exception $\uparrow_i$). Program judgments $\langle a, \sigma\rangle \rightsquigarrow_n \langle \sigma', \chi\rangle$ are small-step; they implement the reflexive transitive closure of statement judgments by executing n machine instructions starting at address a in state σ to reach state σ' with exit status $\chi \in \{\downarrow_a, \uparrow_i\}$.

Since expressions include unpredictable values $*$, all three operational semantics are *non-deterministic*. For example, judgment $\langle e, \sigma\rangle \Downarrow u$ asserts that expression e *might* evaluate to value u. Judgment $\langle *_{\boxed{w}}, \sigma\rangle \Downarrow n_{\boxed{w}}$ is therefore derivable for all $n \in [0, 2^w)$. Determinism theorems of the form $\langle e, \sigma\rangle \Downarrow n_1 \rightarrow \langle e, \sigma\rangle \Downarrow n_2 \rightarrow n_1 = n_2$ can establish that certain expressions e have only one possible value.

Machine states $\sigma : v \rightharpoonup u$ are mappings from state variables v to values u (*viz.*, binary numbers or arrays of binary numbers). Each ISA defines its own universe of state variables v and their types. This affords reasoning about many different ISAs or combinations of ISAs for cross-platform software analysis within a common logical framework.

3.2 Symbolic Interpreter

The inductive propositions that comprise Picinæ's core are succinct (in order to minimize the trusted computing base), but are expressed at a level too low for easy state manipulation in large proofs. To allow users to work at a higher level, a symbolic interpreter is implemented that steps an abstract machine state (possibly containing Coq proof meta-variables) by executing a specified number of machine instructions within a Coq proof context.

Interpreting code that branches introduces multiple proof goals to the context—one for each possible branch destination. Interpreting a computed jump leads to an abstract state that must be refined by the prover (e.g., by case distinction on the program counter address) in order to soundly reduce the destination set to a finite set of possible targets before interpretation can continue. Interpretation of programs containing non-deterministic expressions $*$ introduces proof meta-variables to represent the (unknown) values of those expressions, along with hypotheses that constrain the unknowns. For example, after symbolically interpreting the statement at address 32 in Fig. 3, the proof context contains new hypotheses of the form $x : \mathbb{N}$, $\sigma(\texttt{OF}) = x$, and $x < 2$, where x is a fresh Coq proof variable.

In total, the symbolic interpreter and its proof of correctness comprise about 1000 lines of Coq code. For efficiency, it is implemented as functional Gallina code launched using Coq's `vm_compute` tactic.

3.3 Theory Libraries

The power and ease of machine-assisted validation depends largely on the power and scope of the framework's library of proved theorems. Although Picinæ is still in a stage of early development, its proof library already consists of about 3,500 lines of Coq theorems, definitions, and proofs, divided into the following major sections:

- *Inductive schemas* provide proof principles for Floyd-Hoare style inductive reasoning about pre-conditions, invariants, and post-conditions.
- A *static semantics* proves type soundness of lifted IL and establishes appropriate bounds on numeric machine state element values.
- A library of *two's complement arithmetic* facilitates reasoning about signed and unsigned modular arithmetic operations and their effects upon the binary representations of numbers.
- A collection of *determinism lemmas* automatically identifies deterministic expressions and instructions and facilitates deterministic proof development in common cases.
- *Monotonicity* theorems allow proofs to soundly reason about architectures in which only a subset of state components are known, programs in which only a subset of instructions have been lifted, and machine states in which only a subset of state component values are known. This facilitates modular, incremental reasoning about ISAs, programs, and states.

In order to explain the case-studies that follow, we here limit our focus to describing the first of these sections.

In Picinæ, correctness theorems about subroutines are typically expressed as an *invariant set* $I : a \rightharpoonup (\sigma \rightarrow \text{Prop})$, which is a partial function from addresses to machine state propositions, paired with a *post-condition* $Q : \sigma \rightarrow \text{Prop}$ (a machine state proposition). Invariant set I includes the subroutine pre-condition $I(a_0)$, where a_0 is the subroutine's entry point address. The post-condition is a proposition that is asserted to be true whenever execution reaches the subroutine's *return address*, as defined by the architectural conventions of the ISA. For example, on 64-bit Intel architectures, a subroutine's return address is the address a satisfied by proposition $\text{Ret}(a) := \langle \texttt{M}[\texttt{ESP}]_{\boxed{64}}, \sigma_0\rangle \Downarrow a$, where σ_0 is the machine state on entry to the subroutine.

With this formalization, partial correctness theorems have the form:

$$I(a_0)\sigma_0 \rightarrow \big(\langle a_0, \sigma_0\rangle \rightsquigarrow_n \langle \sigma', \downarrow_a\rangle\big) \rightarrow \big(I(a) = P \rightarrow P\,\sigma'\big) \wedge \big(\text{Ret}(a) \rightarrow Q\,\sigma'\big) \quad (1)$$

which asserts that if the pre-condition is satisfied on entry, then whenever execution reaches any address a in any state σ' after any

n steps of computation, state σ' satisfies invariant $I(a)$ if a is an invariant point, and it satisfies post-condition Q if a is the return address. Total correctness theorems conjoin Equation 1 with

$$I(a_0)\sigma_0 \rightarrow \exists n\, a, \left(\langle a_0, \sigma_0\rangle \rightsquigarrow_n \langle \sigma', \downarrow_a\rangle\right) \wedge \mathrm{Ret}(a) \tag{2}$$

which asserts that satisfying the pre-condition guarantees termination after some number of steps n.

Picinæ's `prove_invs` theorem reduces such proof goals to a set of $|I(\mathbb{N})|$ proof cases—one case for each invariant in I. Each case starts symbolic execution at an invariant point in an abstract state satisfying the invariant, and challenges the user to prove that execution inevitably reaches another invariant point in a state satisfying the reached invariant, or exits the subroutine in a state that satisfies the post-condition. Thus, applying `prove_invs` to a partial or total correctness assertion launches an inductive proof that verifies that Q and all invariants in I are satisfied whenever they are reached. This is the core inductive schema for most correctness and safety proofs.

3.4 ISA Tactic Libraries

While the core IL semantics, symbolic interpretation engine, and IL theory libraries are powerful enough to prove facts about programs from arbitrary ISAs, each ISA has specialized definitions (e.g., calling conventions), common cases (e.g., IL expressions deserving auto-simplification), and notations (e.g., assembly syntaxes) that deserve specialized treatment. Each supported ISA therefore has an ISA-specific supporting module that defines the universe v of machine state components and their types, specializes the general IL machinery to ISA-specific tasks, introduces proof notations for improved readability, and adds auto-simplification heuristics for easier program analysis.

The ISA-specific tactic libraries for Intel x86 and ARM32 v7 ISAs consist of about 550 and 500 (respectively) lines of Coq definitions, theorems, proofs, and tactics. These two ISAs are the basis for the case-studies presented in the next section.

4 CASE STUDIES

As a preliminary evaluation of our approach, we machine-verified three heavily optimized native code subroutines extracted from the GNU standard C libraries for Intel x86 and ARM32 using Picinæ. Each was lifted to Picinæ IL using BAP with our BIL-to-Picinæ plug-in, and proof development was conducted interactively using Coq v8.8 for Windows with CoqIDE.

4.1 ARM String-length

Figure 4 lists the assembly code for the binary implementation of `strlen`. Like most aggressively optimized codes, it has an unintuitive structure. Instead of reading one byte at a time from memory, it reads four bytes at a time and tests each constituent byte of the loaded 32-bit word for nullity before proceeding to the next word.

To improve cache alignment, lines 1–4 first round input pointer `r0` *down* to the nearest word boundary and read four bytes from there into `r2`. Lines 5–10 then set all bits within `r2` that precede the start of the string, so that the main loop will disregard them. This works because memory access permission granularity is per-page,

```
1  bic r1, r0, #3
2  ldr r2, [r1], #4
3  ands r3, r0, #3
4  rsb r0, r3, #0
5  beq 11
6  orr r2, r2, #FF
7  subs r3, r3, #1
8  orrgt r2, r2, #FF00
9  subs r3, r3, #1
10 orrgt r2, r2, #FF0000
11 tst r2, #FF
12 tstne r2, #FF00
13 tstne r2, #FF0000
14 tstne r2, #FF000000
15 addne r0, r0, #4
16 ldrne r2, [r1], #4
17 bne 11
18 tst r2, #FF
19 addne r0, r0, #1
20 tstne r2, #FF00
21 addne r0, r0, #1
22 tstne r2, #FF0000
23 addne r0, r0, #1
24 bx lr
```

Figure 4: ARM32 `strlen` disassembly

and pages are multiples of the word size, ensuring that reading these extra bytes never raises a spurious access exception.

The main loop (lines 11–17) uses a chain of conditionally executed instructions to test each byte of the loaded word for nullity without explicit conditional branches. In particular, null detection sets the Z flag as a side-effect, preventing the remaining conditional instructions in the loop body from executing, and thereby exiting the loop. Concluding lines 18–24 use a similar strategy to determine which null byte caused the loop to exit, assign the computed length to `r0`, and return to the caller.

We proved total correctness of this subroutine in about 230 lines of Coq definitions, theorems, and proofs. More than half of the proof is devoted to proving abstract facts about bit arithmetic; only about 80 lines regard the computational aspects of the code. This highlights a recurring theme that we have experienced in many of our experiments: Correctness of optimized native codes frequently depends upon a host of obscure facts about binary arithmetic and logical operations that are not easily provable from any existing Coq library. Building a more comprehensive library of proved theorems relevant to assembly-level binary arithmetic is therefore important for scaling our approach to larger programs.

Our proof assumes as a pre-condition that initial state σ_0 satisfies the architectural calling conventions, and we assign a single invariant to line 11:

$$\begin{aligned}
&\exists k, \sigma(\mathtt{r0}) = p \oplus 4k - p \bmod 4 \wedge \\
&\quad \sigma(\mathtt{r1}) = p \oplus 4(k+1) - p \bmod 4 \wedge \\
&\quad \sigma(\mathtt{r2}) = m(p \oplus 4k - p \bmod 4) \mid (k\ ?\ 0 : 2^{8(p \bmod 4)} - 1) \wedge \\
&\quad \forall i,\ i < 4k - p \bmod 4 \rightarrow m(p \oplus i) \neq 0
\end{aligned} \tag{3}$$

where $p = \sigma_0(\mathtt{r0})$, $m = \sigma_0(\mathtt{M})$, and $\mid$ denotes bitwise-or. This asserts that `r0` points to the most recently read memory-word, `r1` points to the next word, `r2` contains the most recently read word (possibly with low-order bits set if $k = 0$), and all bytes between p and `r1` are non-null. Post-condition Q is defined by

$$m(p \oplus \sigma(\mathtt{r0})) = 0 \ \wedge\ \forall i,\ i < \sigma(\mathtt{r0}) \rightarrow m(p \oplus i) \neq 0 \tag{4}$$

which asserts that `r0` holds the length of string p on exit.

The use of 32-bit modular addition $\oplus$ in these predicates insulates them against a peculiar corner case: The subroutine's search for nulls could wrap around the end of the address space, leading to unusual conditions such as $\sigma(\mathtt{r1}) < p$. However, if one considers byte sequences that wrap around the end of the address space to be

legal strings, then the subroutine is nevertheless correct (but only because its memory-reads are all word-aligned, and therefore no individual load ever spans the address-space limit). We discovered this complication late in the validation effort, forcing us to change our proof strategy.

The main proof applies Picinæ's `prove_invs` tactic (see §3.3) to reduce the correctness theorem to two subgoals—one for the pre-condition and one for the invariant. Applying the symbolic interpreter to the latter yields five sub-goals: four for the possible exit paths from the loop and a fifth that cycles back to the invariant. All but the last of these subgoals is solved by proving a `null_terminate` lemma that reasons that finding a null in byte j of the loaded word equates to finding a null at index $4k + j$ of the string. (Verifying this property of bit arithmetic constitutes the majority of the proof logic.) The final subgoal is solved by proving that the state reached by the symbolic interpreter satisfies the loop invariant.

4.2 ARM Memset

We next verified a subroutine with even more complex optimizations: `memset`. The control flow for `memset` consists of three loops and an additional branch (which skips the first two loops if the buffer has fewer than 8 bytes), composing 31 instructions total. The first loop stores individual bytes to reach the first word boundary. This prepares the second loop, which stores two words per store instruction, aligned at word boundaries. The second loop is unrolled four times, with each iteration storing up to 8 words (32 bytes) total. The third loop performs the remainder of stores, and is also unrolled four times. Like `strlen`, `memset` uses conditional execution extensively to avoid branching in the unrolled loops; 20 of the 31 instructions are conditional.

All three loops use register `r2` to track of the number of remaining bytes. Register `r1` holds the character value to be stored in each byte. The setup for the second loop duplicates the least 8 bits of `r1` to fill the 32-bit register. Register `r1` is then copied to `r12` to store two words with one instruction, and `r3` holds the address for the next store in memory. Given starting address p, the character c to be stored, and the length n of the buffer, all three loop invariants share a common predicate:

$$\begin{aligned} &\sigma(\texttt{r3}) \oplus \sigma(\texttt{r2}) = p \oplus n \wedge \\ &\sigma(\texttt{r1}) \bmod 2^8 = c \wedge \\ &\forall i,\ i < \sigma(\texttt{r3}) \ominus p \rightarrow m(p \oplus i) = c \end{aligned} \tag{5}$$

The second loop's invariant conjoins this with:

$$\begin{aligned} &\sigma(\texttt{r1}) = \sigma(\texttt{r12}) = c \mid c \ll 8 \mid c \ll 16 \mid c \ll 24 \wedge \\ &\sigma(\texttt{r3}) \bmod 4 = 0 \end{aligned} \tag{6}$$

The postcondition of the program is:

$$\forall i,\ i < n \rightarrow m(p \oplus i) = c \tag{7}$$

Proving that the implementation is correct is straightforward except for two optimizations that raise significant complications: the duplication of c to obtain a word filled with c, and the second loop's unusual use of arithmetic underflow to conditionally exit. Both entail proving tricky properties of bit arithmetic.

The word containing copies of c is formed using bitwise-and to cast `r1` to a byte, and then using bitwise-or and shift-left to duplicate it. Validation must therefore prove that extracting any byte of the resulting word yields the original c. To conditionally exit, the second loop subtracts 8 from `r2` to modify an underflow status flag that conditions the execution of the remaining loop instructions. To restore `r2` after the subtraction without corrupting the underflow flag, a bitwise-and clears the least significant 3 bits. Proving that this actually restores `r2` contributes significant complexity to the proof.

In total, the `memset` correctness proof is composed of about 700 lines, of which about 500 are devoted to bit arithmetic properties. Thus, as with the `strlen` experiment, the bulk of the proof's complexity is dominated by proofs of obscure numerical properties of bit arithmetic, after which the computational aspects (e.g., control-flows) are relatively easy to validate.

4.3 Intel x86 String-compare

As a contrast to the previous two experiments, we validated a simpler subroutine on a more complex ISA: Intel x86. Most Intel instructions have many side-effects upon the machine state, potentially complicating proofs about them. However, the implementation of `strcmp` in the x86 GNU standard libraries is straightforward: Its 15 instructions implement a single loop that reads one byte at a time from both strings, comparing each and returning the difference between the first unequal pair. It therefore constitutes a good investigation of how proof complexity scales without the intrusion of problematic bit arithmetic.

Proving correctness of this subroutine was relatively easy. The proof consists of about 15 lines of definitions that formally specify lexicogrphical ordering of strings, followed by about 50 lines of proof. About 20 of those lines are boilerplate proof-initialization tactics that are mostly reused between proofs, leaving only 30 lines of genuine proof effort. These were completed by an expert user in about 1 hour of time.

This indicates that the scalability challenges of machine-checked native code validation mainly revolve around the presence of aggressive optimizations in the code being validated. Picinæ's automation features allow complexities of the ISA to be mostly ignored until they are leveraged by optimized code to implement unusual, extra efficient solutions to programming tasks.

5 RELATED WORK

Few prior works have attempted to machine-validate source-free native codes. RockSalt [8] models a reduced subset of x86 to prove safety of the Google Native Client [15] sandbox. XCAP [9] models a different x86 subset to verify OS-level context management subroutines. Bedrock [2] implements an IL similar to Picinæ's for validating compiler back-ends. Picinæ differs from these prior efforts in that it establishes a foundation for machine-validating *arbitrary* codes, including instruction sequences not emitted by any compiler.

6 CONCLUSION

Validating raw native code without sources is extremely difficult. There are presently few tools for doing so, and most rely on unvalidated software components (e.g., back-ends of compilers for which no machine-checked proof of correctness has ever been developed).

Picinæ fills this gap by providing a fully machine-validated framework for reasoning about binary code lifted to an ISA-agnostic IL. Preliminary experiments validating three low-level subroutines from the GNU C standard libraries show that although formal validation remains challenging, Picinæ's automation facilities nevertheless make it feasible to specify and prove correctness of highly optimized code for both RISC and CISC architectures.

Our experiences indicate that the primary roadblock for scalability of the approach is the inadequacy of existing proof libraries about binary arithmetic for reasoning about many bit-arithmetic properties leveraged by native code to realize optimized computations. Future work should therefore seek to identify and prove more comprehensive, automatable theory libraries for binary arithmetic in order to ease low-level code verification tasks.

ACKNOWLEDGMENTS

The research reported herein was supported in part by ONR Award N0014-17-1-2995, AFRL Award FA8750-15-C-0066 (PA #2019-4224), DARPA Award FA8750-19-C-0006, NSF Award #1513704, and an endowment from the Eugene McDermott family. Any opinions, recommendations, or conclusions presented are those of the authors and not necessarily of the aforementioned supporters.

REFERENCES

[1] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*. 463–469.

[2] Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 391–402.

[3] Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2–3 (1988), 95–120.

[4] Solange Coupet-Grimal. 2003. An Axiomization of Linear Temporal Logic in the Calculus of Inductive Constructions. *Journal of Logic and Computation* 13, 6 (2003), 801–813.

[5] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-flow Trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. forthcoming.

[6] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2011. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*. 191–206.

[7] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *Proceedings of the 8th European Congress on Embedded Real Time software and Systems (ERTS^2)*.

[8] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 395–404.

[9] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. 2007. Using XCAP to Certify Realistic Systems Code: Machine Context Management. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 189–206.

[10] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 305–318.

[11] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*. 1733–1750.

[12] Ming-Hsien Tsai and Bow-Yaw Wang. 2006. Formalization of CTL in Calculus of Inductive Constructions. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*. 316–330.

[13] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing Untrusted Code via Compiler-agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. 299–308.

[14] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294.

[15] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fulagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P)*. 79–93.

[16] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*. 337–352.