Relational Hoare Logic for Realistically Modelled Machine Code

Denis Mazzucato[1], Abdalrhman Mohamed[2], Juneyoung Lee[3](✉), Clark Barrett[2], Jim Grundy[3], John Harrison[3], and Corina S. Păsăreanu[1]

[1] Carnegie Mellon University, Pittsburgh, USA
{dmazzuca,pcorina}@andrew.cmu.edu
[2] Stanford University, Stanford, USA
{abdal,barrettc}@stanford.edu
[3] Amazon Web Services, Seattle, USA
{lebjuney,jmgruj,jargh}@amazon.com

Abstract. Many security- and performance-critical domains, such as cryptography, rely on low-level verification to minimize the trusted computing surface and allow code to be written directly in assembly. However, verifying assembly code against a realistic machine model is a challenging task. Furthermore, certain security properties—such as constant-time behavior—require relational reasoning that goes beyond traditional correctness by linking multiple execution traces within a single specification. Yet, relational verification has been extensively explored at a higher level of abstraction. In this work, we introduce a Hoare-style logic that provides low-level, expressive relational verification. We demonstrate our approach on the s2n-bignum library, proving both constant-time discipline and equivalence between optimized and verification-friendly routines. Formalized in HOL Light, our results confirm the real-world applicability of relational verification in large assembly codebases.

AQ1

Keywords: Relational Verification · Machine Code · Mechanized Proofs

1 Introduction

Verification of low-level program properties is paramount for security-critical systems. This applies to microkernels [25], where processors and other hardware may have effects that are not captured by high-level abstractions, as well as cryptographic libraries [6,13], which aim to minimize the trusted computing base and build-toolchain dependencies. Additionally, performance-critical code is often written directly in assembly to maximize performance.

Many challenges arise when verifying low-level code, as programs execute on machines with finite memory, bounded integers, unstructured control flow, and memory space that is shared between data and code. In contrast, high-level

D. Mazzucato and A. Mohamed—Contributed equally to this work.

verification uses abstractions that simplify reasoning and hide hardware-specific details. For instance, low-level verification must consider that primitives, when storing data, need to have free memory space. Furthermore, the verification process becomes much harder when dealing with *relational properties* [16]. Relational properties link multiple execution traces together within a single property specification. They are necessary for critical applications, such as proving that a cryptographic routine runs in constant time with respect to secret data, or that two versions of a program are functionally equivalent.

In this work, we target the s2n-bignum library,[1] a cryptographic library written in assembly for ARM and x86 architectures. It includes mathematical operations on large integers, such as modular multiplication, as well as more cryptographic-oriented operations, such as elliptic curve operations. As part of the AWS's TLS/SSL implementation, these arithmetic routines are both performance- and security-critical. The library features both highly optimized routines that are hard to verify as well as verification-friendly variants that are easier to verify but slower in practice. By verifying the latter and proving that they are functionally equivalent to the former, we can ensure that the high-performance versions do not compromise correctness. We also aim to ensure that the high-performance routines execute in constant time, a property necessary to prevent timing side-channel attacks, which could compromise sensitive data.

A number of works previously studied Hoare-style logics for realistically modelled machine code [4,14,29,37,42,48]. Hoare-style reasoning has also been pervasively studied for relational properties [9,12,45]. However, relational verification for low-level code remains underexplored, especially for machine code with realistic features such as finite memory and unstructured control flow. Ideally, a binary verification toolkit would use a robust Hoare-style logic that supports realistic machine code, can express relational properties, provides sound and complete proof rules, and retains key properties that users naturally expect. These include, for instance, commutativity, as well as the ability to weaken and strengthen pre- and postconditions and to unify contracts across different contexts. Such *natural properties* enable modular reasoning, support multiple proof strategies, and make the framework practical for real-world applications. To the best of our knowledge, no existing work has presented a relational Hoare logic for realistically modelled machine code that satisfies all these properties.

In this paper, we fill that gap by introducing a novel Hoare-style logic for low-level, relational verification. Our framework, fully formalized in the HOL Light theorem prover [18,19], offers proof rules designed to meet users' natural expectations. We demonstrate our approach via two major case studies: in the first, we show how our framework can be used to verify constant-time behavior of various routines; in the second, we show it can be used to prove the functional equivalence of two different implementations of the same routine (e.g., one optimized for speed and the other optimized for verifiability). These case studies are conducted on the s2n-bignum cryptographic library. Results show that our logic scales to large assembly programs and yields practical value.

[1] https://github.com/awslabs/s2n-bignum.

While our primary application is the s2n-bignum library, the generality of our relational Hoare logic extends beyond cryptographic code. It supports low-level features including indirect branches and self-modifying code, even though cryptographic libraries such as s2n-bignum may not employ these features.

We summarize our contributions as follows:

i) A novel relational Hoare logic tailored to realistically modelled machine code, formalized in HOL Light.
ii) A first case study on constant-time behavior of cryptographic routines in the s2n-bignum library, including the copy and modular inversion routines.
iii) A second case study involving equivalence proofs between optimized and verification-friendly implementations of s2n-bignum routines.

2 Related Work

Hoare-Style Reasoning for Realistically Modelled Machine Code. Verifying realistically modelled machine code is challenging due to unstructured control flow, which traditional Hoare logics [20] struggle to handle. While Affeldt [2] uses Hoare logic to verify low-level arithmetic routines, their work is limited to assembly fragments with structured control flow. Several approaches address unstructured control flow, such as the inductive assertion method [39, Section 2] used by Barthe et al. [7] and Lehner and Muller [28] to generate verification conditions, and the program logic by Tan et al. [47] based on continuation-passing style reasoning [8]. However, these methods often fail to specify pre- and postconditions with shared continuation labels.

Other notable efforts include a logic for total correctness of communicating unstructured programs [4, 21, 22], formalized in Isabelle/HOL, and one for reasoning about MIPS assembly in Coq [30]. However, these logics are compositional only for nonoverlapping fragments. Full compositionality is critical for modular reasoning, which, in turn, is needed for scalability.

Wang [48] proposes a logic for total correctness of unstructured programs with multi-exit postconditions, but it does not guarantee postconditions upon first encounter. Unstructured programs may, in fact, go through the last instruction, jump back, and then meet the postcondition later. While this might seem misleading, we argue that functional specifications are generally confined to function boundaries, where the final instruction is a return statement. This ensures the program cannot continue execution and revisit the postcondition later, effectively solving the issue of first-met postconditions.

Myreen and Gordon [38] introduce a logic for unstructured code applied in the verified CakeML compiler [27], leveraging decompilation into logic [33–37]. Despite its major impact in verifying seL4 compilation [46] and realistic executables [47], it lacks a conjunction rule, a property that is naturally expected from a Hoare-style logic in order to unify contracts over different postconditions. The lack of a conjunction rule significantly increases the proof burden.

For instance, this assembly program increments x0 by 1 until it reaches 3, then halts. Let $P = (\texttt{x0} = 1)$, $Q = (\texttt{x0} = 2)$, and $Q' = (\texttt{x0} = 3)$. The program satisfies Q and Q' separately, on line 3, after two and three iterations, respectively, but Q and Q' cannot possibly hold simultaneously. In Sect. 5, we show how to handle such cases.

```
0    mov x0,xzr
1  loop:
2    add x0,x0,#1
3    cmp x0,#3
4    bne loop
```

Ray et al. [43] address conjunction rules and first-met postconditions by tracking execution steps, while Lundberg et al. [29] extend this to handle multi-exit locations, ensuring postconditions hold at the first encounter. However, their approaches assume deterministic semantics, incompatible with architectures like x86. Jensen et al. [23] addresses this gap by using separation logic [40,44] but only for a subset of x86 code. Furthermore, EverCrypt [42] verifies cryptographic primitives using a Hoare-style logic through C and assembly code interoperability, but it does not support ARM architecture. Similar to our embedding of relational to unary Hoare triples, exploiting the event list, EverCrypt is able to prove constant-time. Fiat-Crypto [17] generates verified cryptographic code from high-level specifications with applications to big-number arithmetic, in scope similar to the s2n-bignum library. All these approaches lack a robust foundation for generic relational properties—not only the ones reducible to unary properties.

Relational Hoare Logics. Relational Hoare logics [39] extend unary Hoare triples to reason about multiple execution traces. Benton [9] gives a relational Hoare logic for two execution traces, later generalized by Blatter et al. [12] to any number of traces. While Benton also covers relational properties of low-level unstructured code [8], their logic relies on an idealized computational model. We propose a generic framework for manual proof of relational properties.

In credible compilation, Rinard [45] developed relational logics for pointer allocation, and Benton [10] proposes a sound-but-incomplete fully automatic tool for equivalence preservation of compiled programs with minor differences. They verify HHVM bytecode, which is not a high-level language but not as low as assembly; for instance, they do not handle physical registers. Instead, our approach sacrifices automation, gaining both soundness and completeness.

Barthe et al. [5] propose product program constructions for equivalence reasoning, later extended [3,11] to support equivalence across multiple programs. Kang et al. [24] describe a relational logic for LLVM code which lacks support for indirect branches and self-modifying code. Pit-Claudel et al. [41] further extend relational verification to low-level stack machines. Our logic handles relational properties but does not trade off any low level features of assembly code.

3 Running Example

We use the program `compare`, shown in Fig. 1 (left), as a running example for the rest of the paper. It compares byte-by-byte the contents of a key buffer $\boldsymbol{k}$ and a data buffer $\boldsymbol{x}$ of length n. It takes as input the buffer length n and the memory addresses of the buffers k and x, provided via the registers `n`, `k`, and

Program 1.1. compare

```
  cbz n, eq
loop:
  sub n, n, #1
  ldr kn, [k, n, lsl #3]
  ldr xn, [x, n, lsl #3]
  cmp kn, xn
  bne neq
  cbnz n, loop
eq:
  mov res, #1
  ret
neq:
  mov res, xzr
  ret
```

Program 1.2. cst-compare

```
  mov diff, xzr
  cbz n, end
loop:
  sub n, n, #1
  ldr kn, [k, n, lsl #3]
  ldr xn, [x, n, lsl #3]
  eor temp, kn, xn
  orr diff, diff, temp
  cbnz n, loop
end:
  cmp diff, xzr
  cset res, eq
  ret
```

Fig. 1. Two programs to perform byte-by-byte comparison of buffers. The program `compare` (left) is not constant-time, while the program `compare-constant` (right) is.

`x`, respectively. Temporary values are stored in the registers `kn`, `xn`, `diff`, and `temp`, while the result is stored in `res`. The private data is the content of the key buffer. The program `compare` iterates backwards, comparing corresponding elements from both buffers. If a mismatch is detected, the program jumps to the label `neq` and sets `res` to 0. Otherwise, if all elements match, it reaches the label `eq` and sets `res` to 1. This behavior results in variable execution time depending on the buffer contents. An attacker can exploit this timing variation to deduce the position of mismatches and reconstruct the secret buffer $\boldsymbol{k}$ in linear time.

To address this issue, program `cst-compare` in Fig. 1 (right) implements a constant-time comparison. It always iterates over the entire buffer length, regardless of mismatches, accumulating possible differences in `diff`. The program's execution time is constant for any buffer content, ensuring no timing leaks and preventing attackers from inferring secret information. Furthermore, the two programs are functionally equivalent.

4 Unary Hoare Logic $\mathcal{L}_1$

This section provides background on an unary Hoare logic used in the verification framework described in [29,38]. We refer to this logic as $\mathcal{L}_1$. The definitions and theorems of $\mathcal{L}_1$ have been fully mechanized in HOL Light by previous researchers.

States. Let Σ denote a set of machine states, represented as the set of functions mapping observable resources $\mathbb{L}$ (e.g., memory, registers, program counter) to their values. For example, in the ARM architecture, the resources $\mathbb{L}_{\text{ARM}}$ include a 64-bit program counter `pc`, 32 general-purpose registers $\texttt{regs}_i$, flags $\texttt{flags}_k$, and memory $\texttt{memory}_h$, indexed accordingly by i, k, and h. Similarly, the X86 architecture has resources like an instruction pointer (`rip`) and extended flags. To generalize across different architectures, the label `instr` refers to the address of the next instruction, where $\texttt{instr} = \texttt{pc}$ for ARM and $\texttt{instr} = \texttt{rip}$ for X86. We use $s(l)$ for the value of resource $l \in \mathbb{L}$ in the state $s \in \Sigma$ and $s[l \mapsto v]$ for the updated state. Resource values depend on the architecture; e.g., for ARM, $s(\texttt{pc}) \in \texttt{int64}$, $s(\texttt{regs}_i) \in \texttt{int64}$, and $s(\texttt{memory}_h) \in \texttt{byte}$, where $\texttt{byte} \overset{\text{def}}{=} \{0,1\}^8$ and $\texttt{int}n \overset{\text{def}}{=} \{0,1\}^n$.

Properties. A property P is a subset of machine states Σ. A state s satisfies the property $P \subseteq \Sigma$ if $s \in P$. The execution of a single instruction is modelled as the small-step operational semantics $\tau \subseteq \Sigma \times \Sigma$, where $s \xrightarrow{\tau} s'$ describes the fetch-decode-execute cycle, updating state s to s' and advancing $\texttt{instr}$. The composition of two relations τ_1, τ_2 is defined as $\tau_1 \circ \tau_2 \overset{\text{def}}{=} \{(s, s'') \mid \exists s'.\ s \xrightarrow{\tau_1} s' \xrightarrow{\tau_2} s''\}$. The n-th composition of τ is τ^n. The decoding function $\textsc{decode}^\tau(s,i)$ maps bytes in the memory at address i either to an instruction or to $\bot$ if undecodable. ARM instructions have a length of 4 bytes and are 4-byte aligned. X86 instructions have variable lengths. We write $\textsc{length}^\tau(C)$ for the number of bytes that the program C occupies in the memory without padding for alignment. Execution halts when the first undecodable instruction is encountered, denoted by $\textsc{end}^\tau(s,i) \overset{\text{def}}{\Longleftrightarrow} s(\texttt{instr}) = i \wedge \textsc{decode}^\tau(s,i) = \bot$.

We use $\textsc{align}^\tau(s, i_0, C)$ in this paper to denote that the program C is stored in memory starting from the address i_0 where $s(\texttt{instr}) = i_0$ and i_0 satisfies the alignment constraint of a program if the architecture is ARM. The predicate $\textsc{align}^\tau(s, i_0, C)$ may appear as a conjunctive clause in P to describe the program of interest. The notation $\texttt{prog}(P)$ refers to the program C constrained by P.

The `eventually` *Property.* Assume that a machine state $s \in \Sigma$ satisfies a precondition. A postcondition $Q \subseteq \Sigma$ must eventually hold after a finite number of steps from s. To represent such s, $\texttt{eventually}^\tau(Q)$ defines the set of states from which Q eventually holds along every possible path through τ.

Definition 1 (Eventually). *Given an operational semantics $\tau \subseteq \Sigma \times \Sigma$ and a property $Q \subseteq \Sigma$, the property $\texttt{eventually}^\tau(Q) \subseteq \Sigma$ is defined inductively as:*

$$\frac{s \in Q}{s \in \texttt{eventually}^\tau(Q)} \qquad \frac{\exists s'.\ s \xrightarrow{\tau} s' \qquad \forall s'.\ s \xrightarrow{\tau} s' \implies s' \in \texttt{eventually}^\tau(Q)}{s \in \texttt{eventually}^\tau(Q)}$$

The second inference rule expands $\texttt{eventually}$[2] if every next state s' is in $\texttt{eventually}^\tau(Q)$. This notion of `eventually` is essential for reasoning about nondeterministic operational semantics, such as in X86, where certain instructions exhibit nondeterministic behavior. For instance, the `mul` instruction[3] nondeterministically sets the `SF` flag to either 0 or 1. A simplified small-step semantics for `mul` is as follows:

$$\frac{s(\texttt{instr}) = i \quad \textsc{decode}^{\tau_{\text{x86}}}(s,i) = \texttt{mul}\ r \quad r \in \texttt{int16} \quad s(r) = x \quad sf \in \{0,1\}}{s \xrightarrow{\tau_{\text{x86}}} s\,[\texttt{EAX} \mapsto s(\texttt{AX}) \cdot x,\ \texttt{SF} \mapsto sf,\ \texttt{instr} \mapsto i + \textsc{length}^\tau(\texttt{mul}\ r)]}\ \textsc{mul}$$

[2] We omit the τ symbol when the operational semantics is clear from the context.
[3] https://www.felixcloutier.com/x86/mul#flags-affected.

Example 1. Consider a program C consisting of the instructions `mul ax`, `sets dl`, and `imul edx, eax`. The program starts at instruction register i_0, where `AX` (the least significant 16 bits of `EAX`) is multiplied by itself, setting `SF` to 0 or 1. In the second instruction, the least significant byte of `EDX`, referred to as `DL`, is set to `SF`. After then, the values of `EAX` and `EDX` are multiplied, and the truncated result up to 32 bits is stored in `EDX`. The program terminates at $i_0 + 9$ because each of the x86 instructions is 3 bytes, and `EDX` is equal to either 0 or x^2. The postcondition can be expressed as: $\big\{s' \mid \textsc{end}^{\tau_{x86}}(s', i_0 + 9) \wedge (s'(\texttt{EDX}) = x^2 \vee s'(\texttt{EDX}) = 0)\big\}$. It holds under a precondition requiring `AX` to initially be equal to x and `EDX` to 0.

Unary Hoare Triple. Reasoning about machine code differs from reasoning about high-level languages in several ways. First, the machine code is not represented as a syntactic program but instead as a set of instructions in the memory space. Second, a machine code may modify anything during its execution, including itself and callee-save registers. To denote unwanted modifications after the program execution, a *frame* condition $F \subseteq \Sigma \times \Sigma$ bounds allowed changes of state components between the input and output states. We explain the formal definition of the predicate `ensures` which is the Hoare triple in $\mathcal{L}_1$. The notation used in this paper follows the convention you may find in the s2n-bignum library.

Definition 2 (Ensures). *Given an operational semantics $\tau \subseteq \Sigma \times \Sigma$, a precondition $P \subseteq \Sigma$, a postcondition $Q \subseteq \Sigma$, and a frame condition $F \subseteq \Sigma \times \Sigma$, we define the predicate* $\texttt{ensures}^{\tau}(P, Q, F)$ *as follows:*

$$\begin{aligned}&\texttt{ensures}^{\tau}(P, Q, F) \overset{\text{def}}{\iff} \\ &\quad \forall s.\ s \in P \implies s \in \texttt{eventually}^{\tau}(\{s' \mid s' \in Q \wedge (s, s') \in F\})\end{aligned}$$

Example 2. Consider the program C from Example 1, starting from the precondition $\{s \mid \textsc{align}^{\tau_{x86}}(s, i_0, C) \wedge s(\texttt{AX}) = x \wedge s(\texttt{EDX}) = 0\}$, ensuring that the memory is aligned with C and `AX` is equal to x. By application of the operational semantics, we eventually satisfy the postcondition where C terminates with `EDX` equal to 0 or x^2.

During execution, C may modify `EAX`, `EDX`, and the sign flag `SF`. We denote by $\textsc{mayChange} : \wp(\mathbb{L}) \to \wp(\Sigma \times \Sigma)$ the resources that the program may modify. Formally, $\textsc{mayChange}(L) \overset{\text{def}}{=} \{(s, s') \mid \forall l \in \mathbb{L}.\ l \notin L \implies s(l) = s'(l)\}$. Thus, the frame condition can be written as $\textsc{mayChange}(\{\texttt{instr}, \texttt{EAX}, \texttt{EDX}, \texttt{SF}\})$. The correctness of C is captured by:

$$\texttt{ensures}^{\tau_{x86}}\left(\begin{array}{l}\{s \mid \textsc{align}^{\tau_{x86}}(s, i_0, C) \wedge s(\texttt{AX}) = x \wedge s(\texttt{EDX}) = 0\}, \\ \big\{s \mid \textsc{end}^{\tau_{x86}}(s, i_0 + 12) \wedge \big(s(\texttt{EDX}) = x^2 \vee s(\texttt{EDX}) = 0\big)\big\}, \\ \textsc{mayChange}(\{\texttt{instr}, \texttt{EAX}, \texttt{EDX}, \texttt{SF}\})\end{array}\right)$$

Recall that the frame rule in separation logic [40,44] states that if $\{P\}\ C\ \{Q\}$ holds, then for a disjoint memory region R, $\{P * R\}\ C\ \{Q * R\}$ also holds. Similarly, in our logic, if R is invariant under $\textsc{mayChange}(L)$, i.e., $\forall s, s'.\ (s, s') \in$

$\textsc{mayChange}(L) \implies (s \in R \iff s' \in R)$ then $\texttt{ensures}(P, Q, \textsc{mayChange}(L))$ implies $\texttt{ensures}(P \cap R, Q \cap R, \textsc{mayChange}(L))$. Therefore, $\mathcal{L}_1$ supports modular verification while preserving the simplicity of first-order predicates, enabling efficient proof automation.

The logic $\mathcal{L}_1$ is equipped with the usual derivation rules for reasoning about the program execution [32, Appendix A]. The logic core and tactics are implemented in 10k lines of HOL Light [18]. It is currently used to verify functional safety properties of the s2n-bignum library, comprising 615 arithmetic routines written in ARM and x86 assembly languages for P-256/384/521, x25519/ed25519 and RSA. A total of 1013 functional properties have been verified, amounting to 860k lines of proofs.

5 Program Logic $\mathcal{L}_2$ for Relational Verification

In this section, we first introduce a stronger variant of the `eventually` predicate. Then, we present the relational logic $\mathcal{L}_2$ as a natural extension of $\mathcal{L}_1$. We show how to prove a unary Hoare triple from a relational one and vice versa. This last step is essential in demonstrating the robustness of our logic and allows proofs to transition between $\mathcal{L}_1$ and $\mathcal{L}_2$. We highlight the main extensions that allow us to prove relational properties and leave a discussion about the details of the challenges in Appendix B [32].

5.1 Unary Hoare Triples with Number of Steps

Building on [43], we propose a stronger `eventually` operator that explicitly specifies the number of steps required to reach a given postcondition.

Definition 3 (Stronger Eventually). *Given an operational semantics $\tau \subseteq \Sigma \times \Sigma$ and a number of steps $n \in \mathbb{N}$, for any postcondition $Q \subseteq \Sigma$, we define:*

$$\texttt{eventuallyn}^{\tau}_{n}(Q) \overset{\text{def}}{=} \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall s'.\ s \xrightarrow{\tau^n} s' \implies s' \in Q \;\wedge \\ \forall s', l \in \mathbb{N}.\ l < n \wedge s \xrightarrow{\tau^l} s' \implies \exists s''.\ s' \xrightarrow{\tau} s'' \end{array} \right\}$$

That is, it defines the set of states such that for all states reachable in n steps, the postcondition Q must hold, and for all states reachable in less than n steps, there must exist a successor state.

There are two merits in specifying the number of steps n. First, it makes the conjunction rule sound. In low-level languages, a program execution that failed to satisfy the postcondition at `instr` may continue as long as it encounters decodable instructions, and then branch back prior to `instr` and eventually satisfy the postcondition. Therefore, writing multiple postconditions at `instr` that hold at different steps but not together would break the conjunction rule as shown in Sect. 2. Explicitly stating the exact number of steps to arrive at the postcondition as an additional constraint resolves such problem. Second, it retains soundness of the commutativity and composition of nested `eventuallyn` operators, which are similarly important for proving natural properties of relational Hoare triples. When a low-level program exhibits nondeterministic behavior, each trace may meet the postcondition after different numbers of steps.[4] The definition of `eventuallyn` is stronger than `eventually`, cf. Definition 1.

Lemma 1. $\forall Q \subseteq \Sigma, n \in \mathbb{N}.\ \mathtt{eventuallyn}_n(Q) \subseteq \mathtt{eventually}(Q)$

The stronger eventually operator supports the following properties:

Conjunction. As we require postconditions to hold after exactly n steps, we can unify contracts stating different postconditions on the final states.

$$\frac{s \in \mathtt{eventuallyn}_n(Q) \qquad s \in \mathtt{eventuallyn}_n(Q')}{s \in \mathtt{eventuallyn}_n(Q \cap Q')}\ \textsc{conj}$$

Commutativity. Nested eventually operators commute, implying that the order of the two programs specified by the relational property will not matter. Whenever $Q^\times \subseteq \Sigma \times \Sigma$ is eventually satisfied in n_0 and n_1 steps, for the first and second components of $Q^\times$, the inverse $\{(s_1, s_0) \mid (s_0, s_1) \in Q^\times\}$ is satisfied in n_1 and n_0 steps, respectively.

$$\frac{s_0 \in \mathtt{eventuallyn}_{n_0}\left(\left\{s'_0 \,\middle|\, s_1 \in \mathtt{eventuallyn}_{n_1}\left(\dot{Q}^\times_{\pi_0 = s'_0}\right)\right\}\right)}{s_1 \in \mathtt{eventuallyn}_{n_1}\left(\left\{s'_1 \,\middle|\, s_0 \in \mathtt{eventuallyn}_{n_0}\left(\dot{Q}^\times_{\pi_1 = s'_1}\right)\right\}\right)}\ \textsc{comm}$$

Here, $\dot{Q}^\times_{\pi_i = s_x} \subseteq \Sigma$ contains all states that satisfy Q together with s_x in the i-th component, i.e., $\dot{Q}^\times_{\pi_i = s_x} \overset{\text{def}}{=} \{\pi_{1-i}(s, s') \mid \pi_i(s, s') = s_x \wedge (s, s') \in Q^\times\}$. The projection π_i retrieves the i-th component of a pair of states (zero indexed). Projections are lifted to sets of states by $\pi_i(Q^\times) \overset{\text{def}}{=} \{\pi_i(s, s') \mid (s, s') \in Q^\times\}$.

Composition. Two fragments reaching $Q^\times$ and $R^\times$ in n_0, n_1 and m_0, m_1 steps, respectively, can be composed to reach $R^\times$ in $n_0 + m_0$ and $n_1 + m_1$ steps.

$$\frac{\begin{array}{c} s_0 \in \mathtt{eventuallyn}_{n_0}\left(\left\{s \,\middle|\, s_1 \in \mathtt{eventuallyn}_{n_1}\left(\dot{Q}^\times_{\pi_0 = s}\right)\right\}\right) \\ \forall s'_0, s'_1 . (s'_0, s'_1) \in Q \implies s'_1 \in \mathtt{eventuallyn}_{m_0}\left(\left\{s \,\middle|\, s'_0 \in \mathtt{eventuallyn}_{m_1}\left(\dot{R}^\times_{\pi_0 = s}\right)\right\}\right) \end{array}}{s_0 \in \mathtt{eventuallyn}_{n_0 + m_0}\left(\left\{s \,\middle|\, s_1 \in \mathtt{eventuallyn}_{n_1 + m_1}\left(\dot{R}^\times_{\pi_0 = s}\right)\right\}\right)}\ \textsc{comp}$$

With the three properties of `eventuallyn` (cf. CONJ, COMM, and COMP), we can define a unary Hoare triple that maintains the properties that users would naturally expect from a Hoare logic. To do so, we employ a step function $fn : \Sigma \to \mathbb{N}$ to make the number of steps dependent on a given state.

[4] https://github.com/awslabs/s2n-bignum/blob/c747b1b66801e3975a8da502e18962838d3be945/common/relational2.ml#L86-L243.

Definition 4 (Stronger Ensures). *Given an operational semantics $\tau \subseteq \Sigma \times \Sigma$, a precondition $P \subseteq \Sigma$, a postcondition $Q \subseteq \Sigma$, a frame condition $F \subseteq \Sigma \times \Sigma$, and a step function $fn : \Sigma \to \mathbb{N}$, a* unary Hoare triple *is a statement of the form* $\texttt{ensuresn}_{fn}(P, Q, F)$*, where:*

$$\begin{aligned}&\texttt{ensuresn}_{fn}(P, Q, F) \overset{\text{def}}{\iff}\\ &\quad \forall s.\ s \in P \implies s \in \texttt{eventuallyn}^{\tau}_{fn(s)}\big(\{s' \mid s' \in Q \land (s, s') \in F\}\big)\end{aligned}$$

Whenever precondition P holds for state s, postcondition Q holds for any state s' that is related by $fn(s)$ steps of the execution of the program $\texttt{prog}(P)$, and $\texttt{prog}(P)$ modifies only the memory locations specified by the frame condition F.

As a consequence of Lemma 1, the unary Hoare triple $\texttt{ensuresn}_{fn}(P, Q, F)$ is stronger than $\texttt{ensures}$, cf. Definition 2.

Theorem 1. $\forall P, Q, F, fn.\ \texttt{ensuresn}_{fn}(P, Q, F) \implies \texttt{ensures}(P, Q, F)$

The other direction of the implication is not always true; in fact, it holds only for deterministic programs. The reason is that the program may branch based on a nondeterministic choice, and the postcondition may hold in a different number of steps than the one specified in the Hoare triple.

Theorem 2. For any operational semantics τ, precondition P, postcondition Q, frame condition F, if τ is deterministic, then:

$$\texttt{ensures}^{\tau}(P, Q, F) \implies \exists fn.\ \texttt{ensuresn}^{\tau}_{fn}(P, Q, F)$$

5.2 Relational Hoare Triples

We now define the relational Hoare triple $\texttt{ensures2}_{fn_0,fn_1}(P^\times, Q^\times, F^\times)$ that allows us to reason about the behavior of two programs. Whenever the precondition $P^\times \subseteq \Sigma \times \Sigma$ holds for a pair of states (s_0, s_1), the postcondition $Q^\times \subseteq \Sigma \times \Sigma$ should eventually hold for any pair of states (s'_0, s'_1) that are related by respectively fn_0 and fn_1 steps of the execution of the two programs C_0 and C_1. As for the logic $\mathcal{L}_1$, the two programs are not explicitly given but instead are constrained in the memory space by P, i.e., $C_0 = \texttt{prog}(\pi_0(P))$ and $C_1 = \texttt{prog}(\pi_1(P))$. The frame condition $F^\times \subseteq (\Sigma \times \Sigma) \times (\Sigma \times \Sigma)$ specifies the memory locations that can be modified by the two programs. Formally:

Definition 5 (Relational Ensures). *Given operational semantics $\tau \subseteq \Sigma \times \Sigma$, a precondition $P^\times \subseteq \Sigma \times \Sigma$, a postcondition $Q^\times \subseteq \Sigma \times \Sigma$, and a frame condition $F^\times \subseteq (\Sigma \times \Sigma) \times (\Sigma \times \Sigma)$, two step functions $fn_0, fn_1 : \Sigma \to \mathbb{N}$, a* relational Hoare triple *is a statement of the form* $\texttt{ensures2}_{fn_0,fn_1}(P^\times, Q^\times, F^\times)$*, where:*

$$\begin{aligned}&\texttt{ensures2}_{fn_0,fn_1}(P^\times, Q^\times, F^\times) \overset{\text{def}}{\iff} \forall s_0, s_1.\ (s_0, s_1) \in P^\times \implies s_0 \in M_{s_0,s_1}\\ &\quad \textit{where } M_{s_0,s_1} \overset{\text{def}}{=} \texttt{eventuallyn}_{fn_0(s_0)}\big(\{s'_0 \mid s_1 \in N_{s_0,s_1,s'_0}\}\big)\\ &\quad \textit{and } N_{s_0,s_1,s'_0} \overset{\text{def}}{=} \texttt{eventuallyn}_{fn_1(s_1)}\big(\{s'_1 \mid (s'_0, s'_1) \in Q^\times \land ((s_0, s_1), (s'_0, s'_1)) \in F^\times\}\big)\end{aligned}$$

The definitions of M_{s_0,s_1} and N_{s_0,s_1,s'_0} nest $\texttt{eventuallyn}$ requirements: M_{s_0,s_1} includes all the states where the program C_0 reaches a state s'_0 within $\mathit{fn}_0(s_0)$ steps, and N_{s_0,s_1,s'_0} includes all the states where the program C_1 reaches a state s'_1 where $(s'_0, s'_1) \in Q^\times$ and $((s_0, s_1), (s'_0, s'_1)) \in F^\times$ hold within $\mathit{fn}_1(s_1)$ steps.

As this definition is based on nested $\texttt{eventuallyn}$ operators, thanks to its properties CONJ, COMM, and COMP, it follows that the relational Hoare triple $\texttt{ensures2}$ commutes, is compositional, and allows contract unification.

Lemma 2 (Commutativity). *Given precondition $P^\times$, postcondition $Q^\times$, frame condition $F^\times$, and step functions $\mathit{fn}_0, \mathit{fn}_1$, the relational Hoare triple* commutes*:*

$$\texttt{ensures2}_{\mathit{fn}_0,\mathit{fn}_1}(P^\times, Q^\times, F^\times) \iff \texttt{ensures2}_{\mathit{fn}_1,\mathit{fn}_0}(P^S, Q^S, F^S)$$

where the swapped versions are defined as $X^S \overset{\text{def}}{=} \{(s_1, s_0) \mid (s_0, s_1) \in X^\times\}$.

This symmetry above ensures that the relational logic is invariant to the program orders, allowing their roles to be interchanged without affecting the triple's validity.

Lemma 3 (Compositional). *Given three properties $P^\times, R^\times, Q^\times$, two frame conditions $F_0^\times, F_1^\times$, and four step numbers n_0, n_1, m_0, m_1, it holds that two relational Hoare triples can be* composed *transitively:*

$$\begin{aligned}&\texttt{ensures2}_{\lambda s.n_0,\lambda s.m_0}(P^\times, R^\times, F_0^\times) \wedge \texttt{ensures2}_{\lambda s.n_1,\lambda s.m_1}(R^\times, Q^\times, F_1^\times)\\ &\quad\Longrightarrow \texttt{ensures2}_{\lambda s.n_0+n_1,\lambda s.m_0+m_1}(P^\times, Q^\times, F_0^\times \circ F_1^\times)\end{aligned}$$

Similarly, also the frame condition can be transitively composed. This is essential in Sect. 7 for the composition of program equivalences.

Lemma 4 (Compositional of Frame Conditions). *Given two preconditions P, P', two postconditions Q, Q', and three frame conditions F_0, F_1, F_2, and three step functions $\mathit{fn}_0, \mathit{fn}_1, \mathit{fn}_2$, it holds that two relational Hoare triples can be* composed *transitively with respect to the frame conditions:*

$$\frac{\begin{array}{c}\texttt{ensures2}_{\mathit{fn}_0,\mathit{fn}_1}(P, Q, \{((s_0,s_1),(s'_0,s'_1)) \mid (s_0,s'_0) \in F_0 \wedge (s_1,s'_1) \in F_1\})\\ \texttt{ensures2}_{\mathit{fn}_1,\mathit{fn}_2}(P', Q', \{((s_0,s_1),(s'_0,s'_1)) \mid (s_0,s'_0) \in F_1 \wedge (s_1,s'_1) \in F_2\})\end{array}}{\texttt{ensures2}_{\mathit{fn}_0,\mathit{fn}_2}(P \circ P', Q \circ Q', \{((s_0,s_1),(s'_0,s'_1)) \mid (s_0,s'_0) \in F_0 \wedge (s_1,s'_1) \in F_2\})}$$

Lemma 4 formalizes equivalence transitivity: when a program C_0 is equivalent to C_1 and C_1 is equivalent to C_2, then C_0 is equivalent to C_2. This Lemma is vital in the equivalence proofs because proving the correctness of each optimization step independently is easier than directly proving the equivalence of the original and optimized program.

Lemma 5 (**Conjunction**). *Given two preconditions* $P_0^\times, P_1^\times$*, two postconditions* $Q_0^\times, Q_1^\times$*, and a frame condition* $F^\times$*, two contracts can be unified with a* conjunction*:*

$$\texttt{ensures2}_{fn_0,fn_1}(P_0^\times, Q_0^\times, F^\times) \wedge \texttt{ensures2}_{fn_0,fn_1}(P_1^\times, Q_1^\times, F^\times)$$
$$\implies \texttt{ensures2}_{fn_0,fn_1}(P_0^\times \cap P_1^\times, Q_0^\times \cap Q_1^\times, F^\times)$$

All these properties of our Hoare triples enable us to reason about the behavior of two programs, while maintaining the natural properties of a Hoare logic. Appendix C [32] presents the additional properties of our program logic $\mathcal{L}_2$, including the weakening and strengthening of pre-, post-, and frame conditions. Implemented in HOL Light, the core of the relational verification amounts to 1704 lines of code.

5.3 Connection with Unary Hoare Triples

We compare the relational Hoare triple `ensures2` with the unary counterpart `ensuresn`, demonstrating two key transformations: (1) deriving relational Hoare triples from two unary ones, and (2) extracting a unary Hoare triple from a *hybrid* relational one. These transformations serve a dual purpose. First, deriving a relational triple from unary ones enables reasoning about the behavior of two programs by analyzing each independently:

Theorem 3. *Given two sets of pre-, post-, and frame conditions* P, P', Q, Q'*,* F, F'*, and two step functions* fn_0, fn_1*, it holds that:*

$$\texttt{ensuresn}_{fn_0}(P, Q, F) \wedge \texttt{ensuresn}_{fn_1}(P', Q', F')$$
$$\implies \texttt{ensures2}_{fn_0,fn_1}(P \times P', Q \times Q', F \times F')$$

Second, extracting a unary triple from a hybrid relational one allows results obtained in the unary logic to be seamlessly promoted to the relational framework. A hybrid relational triple is a relational triple where the pre-, post-, and frame conditions relate to unary pre-, post-, and frame conditions, respectively. The goal is to be able to extract a unary Hoare triple from a relational one; hence: (i) the relational precondition should always have a satisfying pair (s_0, s_1) when s_1 satisfies the unary precondition; (ii) if a pair (s_0, s_1) satisfies the relational postcondition, then s_1 should satisfy the unary postcondition; and (iii) the frame condition should be satisfied for the product relation whenever the second component satisfies the frame condition of the unary relation.

Definition 6 (**Hybrid Relational Ensures**). *Given the pre-, post-, and frame conditions for the product relation* $P^\times, Q^\times$*,* $F^\times$*, and unary pre-, post-, and frame conditions* P, Q*,* F*, and two step functions* $fn_0, fn_1 : \Sigma \to \mathbb{N}$*, a* hybrid *relational Hoare triple, written* $h\texttt{ensures}2_{fn_0,fn_1}(P^\times, Q^\times, F^\times \mid P, Q, F)$*, holds if:*

$$\begin{aligned}
&\texttt{ensures}2_{fn_0,fn_1}(P^\times, Q^\times, F^\times) \\
&\land \forall s_1.\ s_1 \in P \implies \exists s_0.\ (s_0, s_1) \in P^\times && (i)\\
&\land \forall s_0, s_1, (s_0, s_1) \in Q^\times \implies s_1 \in Q && (ii)\\
&\land \exists F'.\ \forall s_0, s_1, s_0', s_1'.\ \begin{pmatrix} ((s_0', s_1'), (s_0, s_1)) \in F^\times \iff \\ (s_0', s_1') \in F' \land (s_0, s_1) \in F \end{pmatrix} && (iii)
\end{aligned}$$

Employing the hybrid relational triple $h\texttt{ensures}2$ (with the prefix h denoting "hybrid") simplifies the verification process and makes the logic more robust. For instance, it enables translating correctness proofs for one program to another, equivalent program without having to reprove them, saving time and effort. The next result shows that a hybrid relational Hoare triple can be transformed into a unary Hoare triple.

Theorem 4. $h\texttt{ensures}2_{fn_0,fn_1}(P^\times, Q^\times, F^\times \mid P, Q, F) \implies \texttt{ensuresn}_{fn_1}(P, Q, F)$.

6 Constant-Time Behavior

In this section, we show how our relational logic $\mathcal{L}_2$ can be applied to reason about constant-time behavior. As is customary in security analysis, we discriminate between public and private input data by partitioning the state labels into two disjoint sets, i.e., $\mathbb{L} = \mathbb{L}_{\text{pub}} \cup \mathbb{L}_{\text{pri}}$ and $\mathbb{L}_{\text{pub}} \cap \mathbb{L}_{\text{pri}} = \emptyset$. Public and private data induce equivalence relations on states, i.e., $\simeq_{\text{pub}}$ and $\simeq_{\text{pri}}$ respectively. The public data is accessible to the attacker, while the private data is kept secret. A program is constant-time if, for the same public input data, any two executions terminate with the same number of clock cycles. As a result, private data does not influence the execution time of the program.

Constant-Time via Events Accumulation. It is not practical to specify constant-time behavior by ensuring that the number of steps is equivalent in executions with the same public data, as it is highly dependent on the underlying hardware. Due to microarchitectural effects, such as memory access patterns or branch prediction, the number of clock cycles can vary significantly between executions. Instead, we can safely reason about constant-time behavior by employing a stronger notion of timing security: a program is *constant-time* if—for the same public input data—any two executions of the program induce *the same trace of microarchitectural events.*

To observe these events, we extend the state space Σ with an `events` component in $\mathbb{L}_e \overset{\text{def}}{=} \mathbb{L} \cup \{\texttt{events}\}$. The `events` component records an ordered list of events, such as memory accesses (`load` x, n or `store` x, n; where x is the accessed address and n the operation size in bytes), and branch jumps (`branch` x, y; where

x and y are the current and the destination program counter, respectively), Any other variable-time instruction, such as division or floating point operations can also be included in the event trace. In our case studies, these operations are intentionally left unresolved by the operational semantics, and thus not included in the event trace. These events are public data, i.e., `events` $\in \mathbb{L}_{\text{pub}}$. The extended state space is Σ_{e} with operational semantics τ^{e}. For instance, loading a memory address x into a 16 bit register r collects a load event of 2 bytes:

$$\frac{\begin{array}{c} s(\texttt{instr}) = i \quad \text{DECODE}(s, i) = \texttt{load}\ r\texttt{,}\ x \\ s(\texttt{memory}_x) = v \quad s(\texttt{events}) = e \quad \text{LENGTH}(r) = 16 \end{array}}{s \xrightarrow{\tau^{\text{e}}} s\,[r \mapsto v,\ \texttt{events} \mapsto (e \mathbin{+\!\!+} \texttt{load}\ x, 2),\ \texttt{instr} \mapsto i + \text{LENGTH}^{\tau}(\texttt{load}\ r\texttt{,}x)]}\ \text{LOAD}$$

Therefore, we are now able to specify constant-time behavior by ensuring that the list of microarchitectural events is the same in both executions. Our approach can be easily extended to include other side-channels, such as power consumption.

While we do not include opcode-level information in our events, instruction opcodes can influence the number of cycles (e.g., the `cbz` and `b.ne` instructions in ARM). This relies on an assumption that a program is public information, and therefore the events do not need to carry opcode information. This assumption can be broken if a program runs assembly instructions that are separately stored in a private input buffer. We prove that such things do not happen individually.

Definition 7 (Constant-Time via Event Accumulation). *Let $\tau^{\text{e}} \subseteq \Sigma_{\text{e}} \times \Sigma_{\text{e}}$ be an operational semantics that collects the microarchitectural events, $P \subseteq \Sigma_{\text{e}}$ be a precondition, $Q \subseteq \Sigma_{\text{e}}$ a postcondition, $F \subseteq \Sigma_{\text{e}} \times \Sigma_{\text{e}}$ a frame condition, and $fn_0, fn_1 : \Sigma_{\text{e}} \to \mathbb{N}$ two step functions. The program* $\texttt{prog}(P)$ *is* constant-time *with respect to private data $\mathbb{L}_{\text{pri}}$ if it holds that:*

$$\texttt{ensures2}^{\tau^{\text{e}}}_{fn_0, fn_1}\begin{pmatrix} \{(s_0, s_1) \in P \times P \mid s_0(\mathbb{L}_{\text{pub}}) = s_1(\mathbb{L}_{\text{pub}})\}\,, \\ \{(s_0, s_1) \in Q \times Q \mid s_0(\texttt{events}) = s_1(\texttt{events})\}\,, \\ F \times F \end{pmatrix}$$

Note that, by constraining the public data to be equal in the precondition, we also require that states share the same event trace before executing the program.

Example 3. The program `cst-compare` in Fig. 1 (right) is constant-time with respect to the microarchitectural events of Definition 7. Indeed, `cst-compare` first branches on the length n of the buffers if $n = 0$ at Line 2; otherwise, it compares the buffers byte-by-byte. Assuming registers of 32 bits, each iteration collects two 4-bytes load events: one for each buffer at Lines 5 and 6. Then, it branches to start the next iteration at Line 9 until the end of the buffers, no matter what the comparison result is. Hence, for any public input value, `cst-compare` induces the same event trace.

In contrast, the program `compare` in Fig. 1 (left) is not constant-time since the event trace may be different for two executions. Consider the following counterexample, where $n = 1$ and the buffers are $k = 10$ and $x = 20$. In memory, the two executions contain $s_0(\texttt{memory}_{10}) = 0$ and $s_0(\texttt{memory}_{20}) = 0$; and $s_1(\texttt{memory}_{10}) = 0$ and $s_1(\texttt{memory}_{20}) = 1$ respectively. The two traces differ at the first mismatch, as the loop in the second execution is terminated early. For brevity, the following event traces are simplified omitting the address of branch instructions with the evaluation of the condition:

$$s_0(\texttt{events}) = [\texttt{branch}\ \text{FALSE}, \texttt{load}\ 10, 4, \texttt{load}\ 20, 4, \texttt{branch}\ \text{FALSE}, \texttt{branch}\ \text{FALSE}]$$
$$s_1(\texttt{events}) = [\texttt{branch}\ \text{FALSE}, \texttt{load}\ 10, 4, \texttt{load}\ 20, 4, \texttt{branch}\ \text{TRUE}]$$

Constant-Time via Unary to Relational Embedding. We can employ unary Hoare logic to prove constant-time behavior by showing that private data does not influence the event trace generated during program execution. In other words, it is sufficient to provide a witness trace that depends only on public data.

Definition 8 (Constant-Time via Unary to Relational Embedding). *Let τ^{e} be an operational semantics that collects the microarchitectural events, P be a precondition, Q a postcondition, and F a frame condition. The program* $\texttt{prog}(P)$ *is* constant-time *with respect to private data $\mathbb{L}_{\mathrm{pri}}$ if there exists a function $f : \Sigma_{\mathrm{e}}(\mathbb{L}_{\mathrm{pub}}) \to \mathbb{E}$ such that:*

$$\forall v_{\mathrm{pub}}, e_0.\ \texttt{ensuresn}^{\tau^{\mathrm{e}}}_{fn}\begin{pmatrix}\{s \in P \mid v_{\mathrm{pub}} = s(\mathbb{L}_{\mathrm{pub}}) \wedge e_0 = s(\texttt{events})\}\,, \\ \{s \in Q \mid s(\texttt{events}) = e_0 +\!\!+ f(v_{\mathrm{pub}})\}\,, \\ F\end{pmatrix}$$

where $\Sigma_{\mathrm{e}}(\mathbb{L}_{\mathrm{pub}})$ is the partial projection of states Σ_{e} on public data $\mathbb{L}_{\mathrm{pub}}$, and $+\!\!+$ is the list concatenation.

This approach eliminates the need to run the symbolic simulation tactic twice, but requires providing an explicit witness for the event trace function f. Since this approach proves a statement about a single program execution, the proof structure is very similar to the correctness proof. Therefore, we can merge the two proofs for correctness and constant-time behavior into a single one; thus eliminating the computational effort of checking each proof separately and greatly reducing the overhead of writing and maintaining them. We can retrieve the relational definition by instantiating Theorem 3 with two instances of the same `ensuresn` proof, renamed accordingly.

Example 4. Using list comprehension, for a given public input v_{pub}, the witness f for the program `cst-compare` in Fig. 1 is defined as:

$$[\texttt{branch}\ (v_{\mathrm{pub}}(\texttt{n}) = 0)] +\!\!+ \left[\begin{array}{l|l}\texttt{load}\ (v_{\mathrm{pub}}(\texttt{x}) + v_{\mathrm{pub}}(\texttt{n}) - 1 - i), 4 & \\ \texttt{load}\ (v_{\mathrm{pub}}(\texttt{y}) + v_{\mathrm{pub}}(\texttt{n}) - 1 - i), 4 & i \in [0, v_{\mathrm{pub}}(\texttt{n})) \\ \texttt{branch}\ (i < v_{\mathrm{pub}}(\texttt{n})) & \end{array}\right]$$

where `n`, `x`, and `y` are public data and therefore accessible in v_{pub}.

Note that, routines in the s2n-bignum library can be proven constant-time by instantiating either Definition 7 or Definition 8, the two are equivalent.

7 Equivalence Checking

In this section, we demonstrate the application of our relational Hoare logic framework to equivalence checking between performance and verification-friendly implementations of the same routine in the s2n-bignum library.

Equivalence Between Two Programs. Two programs are considered functionally equivalent if they produce the same output states starting from equivalent input states. When dealing with assembly-level programs, we must carefully define what it means for two states to be "equal". For instance, two equal input states should not require the exact same code in memory; otherwise, only identical programs could be compared. Similarly, because the calling convention allows callee-save registers to hold different values, the value of these registers should not be constrained.

On the output side, certain registers or memory regions may differ if they are not designated as outputs. For example, eliminating dead stores to the stack frame is a valid optimization because the stack frame is not used after function returned. Two equivalent output states must allow those parts of memory to contain different data.

As a consequence, the equivalence checking takes as a parameter the equivalence relations $\simeq_{\text{in}} \subseteq \Sigma \times \Sigma$ and $\simeq_{\text{out}} \subseteq \Sigma \times \Sigma$ that define when input and output states are considered equivalent. This relation has to be defined manually for each pair of programs to be compared.

Example 5. Consider the two programs `compare` and `cst-compare` in Fig. 1. Assuming a proof of correctness for `compare` already exists, our goal is to prove that the secure constant-time version is functionally equivalent to the original program, without needing to reprove the correctness of `cst-compare` from scratch. To do so, we define the input equivalence $\simeq_{\text{in}}$, relating the program counter, input registers, and relevant part of the memory as follows:

$$\simeq_{\text{in}} = \textsc{mayChange}\left(\mathbb{L} \setminus \left(\begin{array}{l}\{\texttt{instr}, \texttt{n}, \texttt{x}, \texttt{y}\} \cup \\ \{\texttt{memory}_i \mid i \in [x, x+n) \vee i \in [y, y+n)\}\end{array}\right)\right)$$

Note the use of the $\textsc{mayChange}$ operator to define a relation that allows two states to differ in all labels but the ones specified. For output equivalence $\simeq_{\text{out}}$, we relate only the output register, i.e., $\simeq_{\text{out}} = \textsc{mayChange}(\mathbb{L} \setminus \{\texttt{res}\})$.

Definition 9 (Equivalence). *Let $P_0, P_1 \subseteq \Sigma$ be two preconditions, $Q_0, Q_1 \subseteq \Sigma$ two postconditions, $F_0, F_1 \subseteq \Sigma \times \Sigma$ two frame conditions, and $fn_0, fn_1 : \Sigma \to \mathbb{N}$ two step functions. Given the input and output equivalences $\simeq_{in}, \simeq_{out} \subseteq \Sigma \times \Sigma$, the programs* $\texttt{prog}(P_0)$ *and* $\texttt{prog}(P_1)$ *are* equivalent *if it holds that:*

$$\texttt{ensures2}_{fn_0, fn_1}\left(\begin{array}{l}\{(s_0, s_1) \in P_0 \times P_1 \mid s_0 \simeq_{\text{in}} s_1\}, \\ \{(s_0, s_1) \in Q_0 \times Q_1 \mid s_0 \simeq_{\text{out}} s_1\}, \\ F_0 \times F_1\end{array}\right)$$

Example 6. We can prove that the constant-time program `cst-compare` is equivalent to the original program in `compare` by applying Definition 9 with the input and output equivalences defined in Example 5. Along the lines of the pre- and postconditions defined in Sect. 6, we define:

$$P' = \left\{ s \,\middle|\, \begin{array}{l} s(\mathtt{n}) = n \wedge s(\mathtt{x}) = x \wedge s(\mathtt{y}) = y \wedge \\ \forall i \leq n.\ s(\mathtt{memory}_{x+i}) = \boldsymbol{x}_i \wedge s(\mathtt{memory}_{y+i}) = \boldsymbol{y}_i \end{array} \right\},$$

$$P_0 = \left\{ s \in P' \,\middle|\, \textsc{align}^{\tau}(s, i_0, \mathtt{cst\text{-}compare}) \right\}, P_1 = \left\{ s \in P' \,\middle|\, \textsc{align}^{\tau}(s, i_0, \mathtt{compare}) \right\},$$

$$Q_0 = \{ s \,|\, \textsc{end}^{\tau}(s, \textsc{length}^{\tau}(\mathtt{cst\text{-}compare})) \}, Q_1 = \{ s \,|\, \textsc{end}^{\tau}(s, \textsc{length}^{\tau}(\mathtt{compare})) \},$$

$$F_0 = \textsc{mayChange}(\{\mathtt{instr}, \mathtt{n}, \mathtt{xn}, \mathtt{yn}\}),$$

$$F_1 = \textsc{mayChange}(\{\mathtt{instr}, \mathtt{n}, \mathtt{xn}, \mathtt{yn}, \mathtt{diff}, \mathtt{temp}\}),$$

$$fn_0(s) = \textsc{largestPrefix}_n(s, x, y), \text{ and } fn_1(s) = s(\mathtt{n}),$$

where $\textsc{largestPrefix}_n(s, x, y)$ is the length of the largest prefix among the two given memory addresses x and y of length n. In conclusion, Definition 9 provides the specification for the equivalence proof between the two programs.

Composition of Program Equivalences. We slightly abuse notation and define `eqensures` as a shorthand for the equivalence of two programs C_0, C_1 with $\simeq_{\text{in}}$ in the precondition starting from $\mathtt{pc}_0, \mathtt{pc}_1$, eventually reaching $\simeq_{\text{out}}$ in the postcondition at $\mathtt{pc}'_0, \mathtt{pc}'_1$: $\mathtt{eqensures}_{\mathtt{pc}_0, \mathtt{pc}'_0, \mathtt{pc}_1, \mathtt{pc}'_1}(C_0, C_1, \simeq_{\text{in}}, \simeq_{\text{out}})$. Notably, Lemma 3 proves that the sequential composition of two equivalence proofs is sound if $\forall s, s'.\ s \simeq_{\text{out}} s' \implies s \simeq_{\text{in}}' s'$. Formally, the *sequential composition* of two equivalences is defined as follows:

$$\frac{\mathtt{eqensures}_{\mathtt{pc}_0, \mathtt{pc}'_0, \mathtt{pc}_1, \mathtt{pc}'_1}(C_0, C_1, \simeq_{\text{in}}, \simeq_{\text{out}}) \quad \mathtt{eqensures}_{\mathtt{pc}'_0, \mathtt{pc}''_0, \mathtt{pc}'_1, \mathtt{pc}''_1}(C_0, C_1, \simeq_{\text{in}}', \simeq_{\text{out}}')}{\mathtt{eqensures}_{\mathtt{pc}_0, \mathtt{pc}''_0, \mathtt{pc}_1, \mathtt{pc}''_1}(C_0, C_1, \simeq_{\text{in}}, \simeq_{\text{out}}')}$$

Lemma 4 instead proves the soundness of the transitive composition of two equivalences, only if the result input and output equivalences preserve the existence of an intermediate state, i.e., $s \simeq_{\text{in}}' s' \iff \exists s''.(s \simeq_{\text{in}} s'' \wedge s'' \simeq_{\text{in}}' s')$ and $s \simeq_{\text{out}}' s' \iff \exists s''.(s \simeq_{\text{out}} s'' \wedge s'' \simeq_{\text{out}}' s')$. Formally, the *transitive composition* of two equivalences is defined as follows:

$$\frac{\mathtt{eqensures}_{\mathtt{pc}_0, \mathtt{pc}'_0, \mathtt{pc}_1, \mathtt{pc}'_1}(C_0, C_1, \simeq_{\text{in}}, \simeq_{\text{out}}) \quad \mathtt{eqensures}_{\mathtt{pc}_1, \mathtt{pc}'_1, \mathtt{pc}_2, \mathtt{pc}'_2}(C_1, C_2, \simeq_{\text{in}1}, \simeq_{\text{out}1})}{\mathtt{eqensures}_{\mathtt{pc}_0, \mathtt{pc}'_0, \mathtt{pc}_2, \mathtt{pc}'_2}(C_0, C_2, \simeq_{\text{in}}', \simeq_{\text{out}}')}$$

Combining Equivalence and Correctness Proofs. In the following, we show how to reuse a correctness proof of an original program to obtain a correctness proof of an optimized program through program equivalence. Indeed, given the functional correctness of the original program in the form of an `ensuresn` proof, we can apply it to the optimized program by proving the equivalence of the two via the relational Hoare triple `ensures2`. The correctness proof of the optimized program is given in the form of a hybrid relational Hoare triple h`ensures2`, presented in Definition 6.

Theorem 5 (Transfer of Correctness through Equality).

$$\texttt{ensuresn}^{\tau}_{fn_0}(P, Q, F) \wedge \texttt{ensures2}^{\tau}_{fn_0,fn_1}(P^{\times}, Q^{\times}, F^{\times})$$
$$\implies h\texttt{ensures2}^{\tau}_{fn_0,fn_1}\left(\begin{array}{l|l} \{(s_0, s_1) \in P^{\times} \mid s_0 \in P\}, & P, \\ \{(s_0, s_1) \in Q^{\times} \mid s_0 \in Q\}, & Q, \\ \{((s_0, s_1), (s_0', s_1')) \in F^{\times} \mid (s_0, s_0') \in F\} & F \end{array}\right)$$

Let $P_0 \subseteq \Sigma$ be the precondition, $Q_0 \subseteq \Sigma$ the postcondition, $F_0 \subseteq \Sigma \times \Sigma$ the frame condition, and $fn_0 : \Sigma \to \mathbb{N}$ the step function. We state functional correctness as: $\texttt{ensuresn}_{fn_0}(\{s \in P_0 \mid s(\texttt{pc}) = x_0\}, \{s \in Q_0 \mid s(\texttt{pc}) = x_\omega\}, F_0)$. Afterwards, from Definition 9, given the two input-output equivalences $\simeq_{\text{in}}$ and $\simeq_{\text{out}}$, the equivalence between two programs is achieved by proving:

$$\texttt{ensures2}_{fn_0,fn_1}\left(\begin{array}{l} \{(s_0, s_1) \in P_0 \times P_1 \mid s_0 \simeq_{\text{in}} s_1\}, \\ \{(s_0, s_1) \in Q_0 \times Q_1 \mid s_0 \simeq_{\text{out}} s_1\}, \\ F_0 \times F_1 \end{array}\right)$$

where P_1, Q_1, F_1 are the pre-, post-, and frame conditions of the second program, respectively. Theorem 5 transfers the correctness and equivalence proofs to the following hybrid relational Hoare triple:

$$h\texttt{ensures2}_{fn_0,fn_1}\left(\begin{array}{l|l} \{(s_0, s_1) \in P_0 \times P_1 \mid s_0 \simeq_{\text{in}} s_1 \wedge s_0(\texttt{pc}) = x_0\}, & \{s \in P_1 \mid s(\texttt{pc}) = x_0\}, \\ \{(s_0, s_1) \in Q_0 \times Q_1 \mid s_0 \simeq_{\text{out}} s_1 \wedge s_0(\texttt{pc}) = x_\omega\}, & \{s \in Q_1 \mid s(\texttt{pc}) = x_\omega\}, \\ F_0 \times F_1 & F_1 \end{array}\right)$$

Finally, by applying Theorem 4, we obtain the correctness proof of the new program: $\texttt{ensuresn}_{fn_1}(\{s \in P_1 \mid s(\texttt{pc}) = x_0\}, \{s \in Q_1 \mid s(\texttt{pc}) = x_\omega\}, F_1)$. In Appendix D [32], we provide the steps required to promote a correctness proof that was originally written via the `ensures` operator—without an explicit number of steps—to a proof that uses the `ensuresn` operator. The majority of functional correctness proofs already available in the s2n-bignum library are written using the `ensures` operator. In total, the core of the equivalence checking proofs is 2629 lines of HOL Light code.

8 Obtaining Proofs for the Hol-Bignum Library

8.1 Case Study: Bignum Copy and Inversion Modulo Routine

We apply the constant-time verification to the s2n-bignum library, notably on the copy program of large integers, cf. `bignum_copy`, and the inversion modulo a prime $p = 2^{255} - 19$, cf. `bignum_inv_p25519`. The following should provide guidance on which proof approach to apply depending on the program size and complexity.

The `bignum_copy` routine is relatively small, comprising 16 instructions that copy the content of buffer k to the buffer z, padding z with zeros if it is bigger than k. Despite its size, `bignum_copy` has the most complex program flow in the library, making it a good candidate for constant-time verification. The functional correctness proof is 180 lines. The constant-time proof, using Definition 7, is 276 lines: it does not require an explicit event trace and is fairly easy to prove correct. On the other hand, the unary constant-time proof using Definition 8 is 245 lines, and requires an explicit event trace. Although the event trace is small and intuitive, this parameter makes the proof more complex as it requires a nontrivial induction on list comprehensions. Notably, we can combine correctness and constant-time proofs together via Theorem 3 in a single, 277-line proof, which yields the lowest proof size overhead.

The `bignum_inv_p25519` routine instead is a 1033-instruction program that finds the inverse of a big integer modulo a prime $p = 2^{255} - 19$. The functional correctness proof is 2303 lines long. The constant-time proof, using the unary embedding of Definition 8 combining correctness and constant-time proofs, is 2633 lines long. Most of the additions in the proof are due to the explicit definition of the event trace, which contains 90 memory events alone. However, after defining the event trace, extending the correctness proof with the constant-time proof was effortless. All the mechanized proofs are available in the artifact.[5] In future work, we plan to automate the generation of the event trace, which will significantly reduce the required level of manual effort.

8.2 Case Study: Elliptic Curves and Montgomery Reduction

We utilized program equivalence to verify the functional correctness of optimized implementations for (1) *field and point operations* of NIST elliptic curves (specifically, curves P-256, P-384, and P-521), and (2) *Montgomery reduction*, an algorithm that allows efficient modular arithmetic when the modulus is large. These optimizations were achieved using an *autovectorizer*, a constraint solver-based instruction scheduler called SLOTHY [1], and the point operations of NIST curves were optimized using a custom memory instruction optimizer for the ARM architecture. We also have similar equivalence checking tactics for the X86 architecture. Overall, we checked the equivalence for 15 pairs of arithmetic routines, amounting to a total of 19k lines of proofs.

The autovectorizer replaces sequences of 64-bit scalar multiplication instructions, such as `mul` and `umulh`, with their equivalent NEON vector instructions. This optimization targets the ARM Neoverse N1 architecture, whose microarchitecture contains only one multiplication pipeline. The `mul`/`umulh` instructions stall this pipeline for a few cycles when executing scalar multiplication instructions. SLOTHY employs a constraint solver and cost model to find the optimal instruction scheduling, significantly reducing these stalls. Specifically, SLOTHY improves the scheduling of straight-line code in the main basic blocks of NIST curves' field operations, and also improves the software pipelining optimization in the main loop of the Montgomery reduction. The memory instruction optimizer performs two key tasks in the ARM architecture: store-to-load forwarding and dead store elimination. Store-to-load forwarding replaces load instructions with

[5] https://doi.org/10.5281/zenodo.15309209.

stored values, eliminating redundant memory accesses. Dead store elimination removes store instructions with results that are never used.

Tactics for Program Equivalence Proofs. To automate the writing of equivalence proofs, we developed proof tactics that can be used for two different classes of optimizations: small localized updates and instruction reordering.

For local optimizations that update only small portions of the original program, such as autovectorization, we implemented the tactic `EQUIV_STEPS_TAC`. This tactic takes as input a list of line ranges and annotations describing whether each range is optimized or left identical. For the identical portion, the tactic performs lock-step symbolic simulation and eagerly abbreviates the common outputs of the instructions with fresh variables to avoid exponential explosion of the sizes of the output expressions. For optimized ranges, the tactic employs stuttering simulation, which executes the corresponding sections of each program step-by-step. To help `EQUIV_STEPS_TAC` automatically converge on complex cases, users can register custom bit-vector equality theorems for output expressions.

For optimizations involving instruction reordering, we implemented two additional tactics: `STEPS_ABBREV_TAC` and `STEPS_REWRITE_TAC`. The first tactic performs stuttering symbolic simulation for the first program, storing the symbolic output expressions to an OCaml array. The second tactic takes as input an instruction index mapping between the two programs, along with the symbolic output generated by `STEPS_ABBREV_TAC`. Then, it simulates the second program step-by-step, proving that the symbolic output of each instruction is equal to the symbolic expression in the first program, according to the instruction mapping.

Software Pipelining of Montgomery Reduction. The Montgomery reduction is heavily used in cryptographic operations performing modular exponentations. Its original implementation in the s2n-bignum library includes a nested loop structure, where the outer loop consists of three basic blocks: loop entry, inner loop (which consists of a single basic block), and loop exit. A faster version was achieved by: caching repetitive calculations, vectorizing `mul` and `umulh` in all basic blocks, applying software pipelining optimizations to the inner loop, and rescheduling instructions using SLOTHY.

We verified the functional correctness of the optimized Montgomery reduction by *transitively* composing equivalence proofs with the original correctness proof after each optimization stage. For each optimization, we applied *sequential* composition of equivalences between each basic block pair and induced the equivalence of the whole loop. In the case of software pipelining, which transforms the control flow graph by adding loop prologue and epilogue blocks, equivalence composition has been applied between each block.

Overall, the optimized field operations of NIST curves achieved throughput speedups up to 38%, and integrating these improvements into point operations alongside memory optimizations resulted in up to 23% throughput gains. These enhancements demonstrate the substantial impact of the new optimizations.

9 Conclusion

This work presents a novel relational Hoare logic framework for verifying realistically modelled machine code, while preserving natural properties expected from Hoare-style reasoning. Fully formalized in HOL Light, the framework is applied in two case studies involving the s2n-bignum cryptographic library, a key component of a TLS/SSL implementation. Our results show that the logic scales to large assembly programs and yields practical value in the verification of cryptographic codebases.

While Mazzucato et al. [31] have investigated constant-time verification for libraries similar to s2n-bignum, their approach relies on abstraction-dependent methods through an untrusted computing base to decompile assembly into C. In contrast, our framework operates directly on the assembly level, ensuring higher reliability of the verification results as it reduces the trusted computing base to the minimal core of the HOL Light theorem prover and to the operational semantics implementations. As future work, we plan to increase coverage of relational properties on the s2n-bignum library and improve proof automation to handle repetitive tasks. As a natural extension to constant-time proofs, we aim to address speculative execution vulnerabilities [15,26].

Acknowledgements. We would like to thank Hanno Becker for his help in improving the Montgomery reduction implementation, and the anonymous reviewers of CAV 2025 for their valuable feedback.

Disclosure of Interests. Research reported in this publication was supported by an Amazon Research Award, Fall 2023.

References

1. Abdulrahman, A., Becker, H., Kannwischer, M.J., Klein, F.: Fast and Clean: Auditable high-performance assembly via constraint solving. Cryptology ePrint Archive, Paper 2022/1303 (2022)
2. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC 2012, pp. 1326–1331. Association for Computing Machinery, New York (2012). ISBN 978-1-4503-0857-1. https://doi.org/10.1145/2245276.2231986
3. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 362–375. Association for Computing Machinery, New York (2017). ISBN 978-1-4503-4988-8. https://doi.org/10.1145/3062341.3062378
4. Bartels, B., Jähnig, N.: Mechanized, compositional verification of low-level code. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 98–112. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_8
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17

6. Barthe, G., Gregoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF 2018), pp. 328–343. IEEE (2018)
7. Barthe, G., Rezk, T., Saabas, A.: Proof obligations preserving compilation. In: Dimitrakos, T., Martinelli, F., Ryan, P., Schneider, S. (eds.) FAST 2005. LNCS, vol. 3866, pp. 112–126. Springer, Heidelberg (2006). https://doi.org/10.1007/11679219_9
8. Benton, N.: A typed, compositional logic for a stack-based abstract machine. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 364–380. Springer, Heidelberg (2005). https://doi.org/10.1007/11575467_24
9. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, pp. 14–25. Association for Computing Machinery, New York (2004). ISBN 978-1-58113-729-3. https://doi.org/10.1145/964001.964003
10. Benton, N.: Semantic equivalence checking for HHVM bytecode. In: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, pp. 1–8. Association for Computing Machinery, New York (2018). ISBN 978-1-4503-6441-6. https://doi.org/10.1145/3236950.3236975
11. Beringer, L.: Relational decomposition. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 39–54. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22863-6_6
12. Blatter, L., Kosmatov, N., Prevosto, V., Le Gall, P.: Certified verification of relational properties. In: Integrated Formal Methods: 17th International Conference, IFM 2022, Lugano, Switzerland, 7–10 June 2022, Proceedings, pp. 86–105. Springer, Heidelberg (2022). ISBN 978-3-031-07726-5. https://doi.org/10.1007/978-3-031-07727-2_6
13. Bond, B., et al.: Vale: verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 917–934 (2017). ISBN 978-1-931971-40-9
14. Bosamiya, J., Gibson, S., Li, Y., Parno, B., Hawblitzel, C.: Verified transformations and hoare logic: beautiful proofs for ugly assembly language. In: Christakis, M., Polikarpova, N., Duggirala, P.S., Schrammel, P. (eds.) NSV/VSTTE -2020. LNCS, vol. 12549, pp. 106–123. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63618-0_7
15. Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., Stefan, D.: SoK: practical foundations for software spectre defenses. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 666–680. IEEE Computer Society (2022). ISBN 978-1-66541-316-9. https://doi.org/10.1109/SP46214.2022.9833707
16. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium, pp. 51–65 (2008). https://doi.org/10.1109/CSF.2008.7
17. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1202–1219 (2019). https://doi.org/10.1109/SP.2019.00005
18. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
19. Harrison, J.: HOL Light Tutorial (for Version 2.20) (2011)

20. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
21. Jähnig, N., Gothel, T., Glesner, S.: A denotational semantics for communicating unstructured code (2015)
22. Jähnig, N., Göthel, T., Glesner, S.: Refinement-based verification of communicating unstructured code. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 61–75. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_5
23. Jensen, J.B., Benton, N., Kennedy, A.: High-level separation logic for low-level code. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, pp. 301–314. Association for Computing Machinery, New York (2013). ISBN 978-1-4503-1832-7. https://doi.org/10.1145/2429069.2429105
24. Kang, J., et al.: Crellvm: verified credible compilation for LLVM. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, pp. 631–645. Association for Computing Machinery, New York (2018). ISBN 978-1-4503-5698-5. https://doi.org/10.1145/3192366.3192377
25. Klein, G., et al.: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220 (2009)
26. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19 (2019). ISSN 2375-1207. https://doi.org/10.1109/SP.2019.00002
27. Kumar, R., Myreen, M., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. ACM SIGPLAN Not. **49**(1), 179–191 (2014)
28. Lehner, H.: Muller: formal translation of bytecode into BoogiePL. Electron. Notes Theor. Comput. Sci. **190**(1), 35–50 (2007)
29. Lundberg, D., Guanciale, R., Lindner, A., Dam, M.: Hoare-style logic for unstructured programs. In: de Boer, F., Cerone, A. (eds.) SEFM 2020. LNCS, vol. 12310, pp. 193–213. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58768-0_11
30. Marti, N.: Formal verification of low-level software. Ph.D. thesis, University of Tokyo (2008)
31. Mazzucato, D., Campion, M., Urban, C.: Quantitative static timing analysis. In: 31st Static Analysis Symposium (SAS 2024), Roberto Giacobazzi and Alessandra Gorla and Marco Campion, Pasadena, CA, United States (2024). https://doi.org/10.1007/978-3-031-74776-2_11
32. Mazzucato, D., et al.: Relational hoare logic for realistically modelled machine code (2025). https://arxiv.org/abs/2505.14348
33. Myreen, M., Curello, G.: Formal verification of machine-code programs. In: Lundberg, D. (ed.) International Conference on Certified Programs and Proofs, p. 20. University of Cambridge, Computer Laboratory (2009)
34. Myreen, M., Gordon, M.: Verification of machine code implementations of arithmetic functions for cryptography. In: Theorem Proving in Higher Order Logics: Emerging Trends Proceedings, Department of Computer Science, University of Kaiserslautern (2007)
35. Myreen, M., Gordon, M., Slind, K.: Machine-code verification for multiple architectures-an application of decompilation into logic. In: 2008 Formal Methods in Computer-Aided Design, pp. 1–8. IEEE (2008)
36. Myreen, M., Gordon, M., Slind, K.: Decompilation into logic—improved. In: 2012 Formal Methods in Computer-Aided Design (FMCAD, pp. 78–81. IEEE (2012)

37. Myreen, M.O., Fox, A., Gordon, M.: Hoare logic for ARM machine code. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 272–286. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75698-9_18
38. Myreen, M.O., Gordon, M.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 568–582. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_44
39. Naumann, D.A.: Thirty-seven years of relational hoare logic: remarks on its principles and history. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12477, pp. 93–116. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_7
40. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of Computer Science Logic (2001)
41. Pit-Claudel, C., Philipoom, J., Jamner, D., Erbsen, A., Chlipala, A.: Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, pp. 918–933. Association for Computing Machinery, New York (2022). ISBN 978-1-4503-9265-5. https://doi.org/10.1145/3519939.3523706
42. Protzenko, J., et al.: EverCrypt: a fast, verified, cross-platform cryptographic provider. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 983–1002 (2020). ISSN 2375-1207. https://doi.org/10.1109/SP40000.2020.00114
43. Ray, S., Hunt, W.A., Matthews, J., Moore, J.S.: A mechanical analysis of program verification strategies. J. Autom. Reason. **40**(4), 245–269 (2008). ISSN 1573-0670. https://doi.org/10.1007/s10817-008-9098-1
44. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS) (2002)
45. Rinard, M.: Credible Compilation (1999)
46. Sewell, T., Myreen, M., Klein, G.: Translation validation for a verified OS kernel. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 471–482 (2013)
47. Tan, J., Tay, H.J., Gandhi, R., Narasimhan, P.: AUSPICE: automatic safety property verification for unmodified executables. In: Gurfinkel, A., Seshia, S.A. (eds.) VSTTE 2015. LNCS, vol. 9593, pp. 202–222. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29613-5_12
48. Wang, A.: An axiomatic basis for proving total correctness of goto-programs. BIT Numer. Math. **16**(1), 88–102 (1976). ISSN 1572-9125. https://doi.org/10.1007/BF01940782

Author Queries

Chapter 19

Query Refs.	**Details Required**	**Author's response**
AQ1	Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city and country names in all affiliations. Please check and confirm if the inserted city and country names are correct. If not, please provide us with the correct city and country names.	