Formally-Verified, Tight Timing Constraints for Machine Code

Anonymous

Abstract

A dependently typed, machine-checked framework is introduced for verifying timing properties of raw (stripped) machine code binaries within the Rocq interactive theorem-proving environment. By formalizing instruction timings and integrating them with an abstract interpreter, it provides high-assurance, high-precision timing guarantees that are applicable to a wide range of systems. This verifies that real-time systems and cryptographic algorithms meet their critical performance and security requirements.

1 Problem and Motivation

Many mission-critical computing systems operate under stringent timing constraints, where deviations in execution time can have severe consequences. Two important categories include: (1) real-time systems, which must meet strict timing deadlines to ensure correct behavior, and (2) cryptographic systems, which must guard against information leakage through timing-based side channels.

1.1 Real-Time Systems

Real-time systems underpin a vast array of mission-critical applications, including avionics, automotive control systems, industrial automation, and medical devices. In these domains, control loops and other real-time functions must execute within precise time intervals to maintain system stability and correctness. Failure to meet these timing constraints can result in catastrophic failures, such as aircraft control loss or medical device malfunctions.

Ensuring the correctness of such systems requires not only functional correctness but also formal guarantees on worst-case execution time (WCET) and schedulability. Traditional WCET analysis techniques [11], such as control flow analysis and measurement-based execution time analysis, provide valuable insights, but they forgo formal guarantees in favor of code coverage and automation.

1.2 Constant-Time Cryptography

Cryptographic implementations present a different but equally pressing challenge. Timing side-channel attacks exploit variations in execution time to infer secrets, breaking confidentiality even when cryptographic algorithms are mathematically sound.

Classic attacks, such as Kocher's timing attacks on RSA [5], demonstrate that even minute timing differences in modular arithmetic operations can leak secret key bits. More recent work has extended these attacks to cache-based timing attacks [1, 9] and branch prediction or speculative execution attacks, such as Spectre and Meltdown [4]. Defenses against timing attacks require ensuring that execution time remains independent of secret data, but achieving this property in practice is difficult due to microarchitectural effects that are often poorly documented and hardware-specific.

1.3 Motivation

Traditional verification techniques focused on bug-finding are often insufficient for obtaining airtight, machine-checkable formal guarantees about timing properties. For example, real-time verification methods rely on conservative WCET bounds that may not accurately reflect true execution behavior. Cryptographic verification techniques, such as constant-time programming methodologies [7], require careful manual implementation and do not inherently prove the absence of timing leaks. The dearth of comprehensive formal methods for timing verification of binaries leaves many safety-critical and security-critical systems vulnerable, calling into question their reliability and trustworthiness.

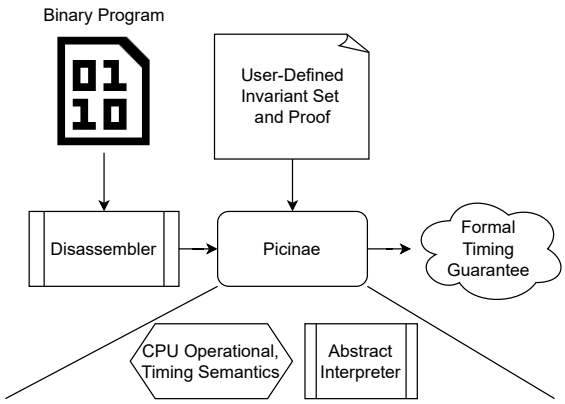

Figure 1: Picinæ Timing Module Pipeline

Addressing this problem requires new formal verification techniques capable of reasoning about execution time in a mathematically rigorous manner. Such techniques must account for hardware-level execution behaviors while remaining applicable to real-world software development workflows. The development of precise, automated proof techniques for timing correctness will not only improve safety in real-time systems but also enhance security in cryptographic implementations, ensuring that these systems can be trusted even in adversarial environments.

2 Background and Related Work

2.1 Picinæ

Our work builds upon Picinæ [3], a framework within the Rocq interactive theorem prover (ITP) for the development of functional correctness proofs for arbitrary (e.g., non-compiled) machine code.

2.1.1 Lifting. Picinæ operates on a low-level intermediate language (PicinæIL) formalized within the Rocq proof assistant. PicinæIL is similar to other ISA-modeling ILs, such as BIL [2] and Ghidra P-code [6], but is dependently typed and strongly normalizing to comply with Rocq's foundations in the calculus of inductive constructions. It represents machine instructions as structured, effect-preserving transformations of an abstract state. Programs are lifted to a Rocq-readable format expressed as a partial map from memory addresses a to IL fragments q that encode the operational effect of the instruction at a on an abstract cpu state. The IL encodes effects via fundamental constructs such as assignments, jumps, conditionals, and bounded repetitions. Instruction-internal loops are explicitly bounded to guarantee strong normalization, eliminating the need for termination proofs of loop-free code fragments.

Expressions in the IL comprise state element reads, memory operations, and modular arithmetic. Certain operations, such as

those that affect architecture-specific flags, are modeled as nondeterministic assignments to account for undefined behavior in the underlying ISA. Picinæ's memory model is purely functional—memory updates are immutable transformations that return a new state rather than destructively modifying a global state.

2.1.2 Invariants. To facilitate formal reasoning about lifted code, Picinæ introduces an *invariant set* framework. Invariant sets define untrusted, machine-verified properties asserted at specific points in the program's execution, forming a basis for inductively proving security and correctness guarantees. They are implemented as a partial map from virtual addresses to cpu state propositions, where propositions may range over Rocq's full higher-order, dependent propositional specification language.

2.1.3 Abstract Interpretation. Picinæ includes a verified symbolic interpreter to analyze lifted machine code within a Rocq proof context. This interpreter enables stepwise execution of an abstract machine state, incorporating Rocq proof meta-variables where necessary to model unknowns. It leverages dependent typing to automatically attach ISA-specific properties to untyped binary state elements within each proof context. For example, w-bit register values have Σ-type $\{n : \mathbb{N} \mid n < 2^w\}$. This affords machine-checked proofs of code properties that rely on ISA-specific properties.

Because the interpreter introduces proof goals corresponding to all possible cases of each branch, complete code coverage is guaranteed—any coverage lapse yields an unprovable proof goal (e.g., a branch to an address with no invariants). Invariant sets thereby prove coverage completeness.

2.1.4 Traces. Program traces in Picinæ are constructed by following execution paths within the lifted IL representation. They capture the sequence of state transitions induced by instruction execution, providing a formal basis for reasoning about control flow and program behavior. By integrating trace analysis with invariant reasoning, Picinæ facilitates proofs of temporal correctness, security, and reachability properties expressible in LTL [10].

2.2 Existing Timing Approaches

2.2.1 Abstract Interpretation statically approximates the behavior of a program by interpreting it with abstract program values. The goal is to determine an upper bound for execution times by using control-flow analysis, where paths through the program are modeled as a set of constraints. In practice, abstract interpretation often faces challenges due to processor-specific models and the complexity of accurately modeling control flow and timing anomalies.

2.2.2 Measurement-Based Analysis involves executing the code on the actual hardware or a simulation, and measuring the execution time for various input sets, recording the maximum execution time observed. Although this can more easily produce anecdotal results for complex systems, it is not a comprehensive search over code paths and offers no formal guarantees. The method's precision can be improved by collecting more measurements, including varying the initial processor state or by analyzing multiple test cases.

3 Approach and Novelty

Our approach extends Picinæ with a *timing module* that models a cpu's timing behavior to provide machine-checkable reasoning power for timing properties. It provides high-assurance timing guarantees for machine code through a rigorous pipeline that prevents false assurances and is approachable to a wider user base than standard formal verification tasks.

3.1 Instruction Timing

3.1.1 Units. We select cpu cycle counts as our unit of time because they provide a granular, consistent measure of execution that is translatable to specific hardware behavior. Unlike higher-level abstractions, such as wall-clock time, cycle counts capture the precise cost of instruction execution, including factors like pipeline stages and memory accesses. This enables a more accurate analysis of performance and resource utilization, crucial for ensuring predictable behavior in real-time systems. Additionally, cycle counts constitute a standard unit that is universally applicable across many different processors and configurations, facilitating actionable comparisons and optimizations.

3.1.2 CPU Selection. We choose the NEORV32 RISC-V cpu for our analysis because of its focus on high-reliability, timing-sensitive computations. The NEORV32's timing behavior is documented as a detailed datasheet [8] that emphasizes analysis-amenable properties, such as non-speculative execution.

3.1.3 Implementation. We encode instruction timings as a Rocq function that maps an instruction's type, arguments, and additional parameters like memory latency, to its cycle count. The cycle count computation takes into account special instructions such as CLZ (count leading zeros) and shift operations, which require the analysis of the immediate value or register values for determining their respective latencies.

In this way our approach offers generalized timing guarantees as a formula whose parameters can include hardware-specific conditions and tolerances when necessary. Such a formula reveals how the parameters must be constrained to achieve desired timing properties, such as worst-case bounds or zero information leakage.

3.2 Trace Timing

Extending Picinæ's symbolic interpreter to model timing properties entails designing and implementing a new instruction timing function onto the cpu trace, yielding a list of cycle counts. This list is then summed, providing the total number of cycles taken to reach the exit point of a function starting from an entry point. Timing properties are universal quantifications over traces, thereby expressing properties of all possible executions of the code.

In our timing proofs, the traces are abstract cpu state histories. This allows for the trace of the entire program to carry information about all possible inputs and control flow paths, ensuring that the trace timing functionality is comprehensive.

3.3 Proof Structure

Picinæ timing proofs tend to be considerably more amenable to proof automation than full functional correctness proofs. Figure 2 illustrates via an example loop that implements Peano addition, and Figure 3 shows a suitable invariant set for the code, consisting of a precondition, loop invariant, and postcondition.

The precondition summarizes the timing behavior of the two instructions before the loop. The loop invariant characterizes the loop's timing behavior and tracks critical information for loop termination. It proposes that the cycle count up to the loop's initial branch is equal to $t_0 + (c_0 - c)t$, where t_0, c_0, c, and t are the pre-loop time, the initial loop counter, the current loop counter, and the time of the loop body, respectively. Finally, the postcondition asserts that the total time taken is equal to $t_0 + c_0 t$.

```
start:
    ori     t2, zero, 1          ; 0
    andi    t3, t3, 0            ; 4
add:
    beq     t0, t3, end          ; 8
    addi    t1, t1, 1            ; 12
    sub     t0, t0, t2           ; 16
    beq     t3, t3, add          ; 20
end:
    ...                          ; 24
```

Figure 2: Peano addition assembly code implementation

```
(* x, y are the addition operands *)
Definition timing_invs (p:addr) (x y:N) (t:trace) :=
 let tb := 5+(ML-1) in (* time of a taken branch *)
 match t with (Addr a, s) :: t' ⇒ match a with
 | 4 ⇒ Some (s R_T0 = x ∧ s R_T2 = 1 ∧
       cycle_count t' = 2 + 2)
 | 8 ⇒ Some (s R_T2 = 1 ∧ s R_T3 = 0 ∧ s R_T0 ≤ x ∧
       (* 2+2+(x-T0) * (3+2+2+(5+(ML-1)) *)
       cycle_count t' = 4 + (x - T0) * (7 + tb))
       (* 2+2+(5+(ML-1)) + x * (3+2+2+(5+(ML-1))) *)
 | 24 ⇒ Some (cycle_count t' = 4 + tb + x * (7 + tb))
 | _ ⇒ None end | _ ⇒ None end.
```

Figure 3: Invariant set for the addloop code in Fig. 2

```
Theorem addloop_timing:
  ∀ s p t,
  satisfies_all
    lifted_addloop                    (* lifted code *)
    (timing_invs p (s R_T0) (s R_T1)) (* invariants *)
    addloop_exit                      (* exit point *)
    t.                                    (* trace *)
Proof.
    (* Address 4 *) repeat step; psimpl; subst; lia.
    (* Address 8 (break/loop cases) *) whammer.
    (* Postcondition *) whammer.
Qed.
```

Figure 4: Abbreviated proof of Fig. 3 timing properties

The structure of the invariant set directly follows from the control flow graph (CFG) of `addloop`. The proof's structure is isomorphic to the invariant set's structure, requiring only standard machinery of Rocq's proof system and Picinæ 's automatic binary arithmetic simplifier, and follows directly from the CFG.

Picinæ's abstract interpreter provides the `step` tactic, which advances the cpu state by one instruction. The core of most timing proofs consists of stepping forward until an invariant is reached (`repeat step`), auto-simplifying the binary arithmetic expressions accumulated during the steps (`psimpl`), and then generating a proof by reflexivity of the invariant-defined timing expression's equality to the proof-generated timing expression, often via Rocq's solver for linear integer arithmetic, `lia`. All three of these steps are largely automated—we provide a tactic `whammer` that performs these actions automatically, as well as a lower-level tactic `hammer` that makes fewer assumptions about the goal. These commonalities between timing proofs, in combination with our automation tactics, result in a straightforward path toward high-assurance timing proofs for arbitrary machine code.

4 Results and Contributions

Our Picinæ timing module allows software developers to obtain high-assurance timing guarantees for mission-critical machine code via an extension of the Picinæ system. Its timing guarantees are easily interpreted by a developer familiar with assembly languages, and proofs are easily developed even by those with limited knowledge of formal verification and ITPs. To demonstrate, we next present two examples of real-world code for which we have developed timing proofs.

4.1 FreeRTOS Context Switcher

FreeRTOS's `vTaskSwitchContext` prepares the cpu for a context switch between tasks. This function contains several branch conditions that appear in the final timing expression, as well as checks for stack overflows that block further execution when triggered. Additionally, the conditions of these branches and checks dereference memory, so rudimentary memory safety subproofs were required for the timing expression to be parametrized only over the initial memory state, for which a near-comprehensive automation tactic was developed. The timing expression for this function is parametrized by several values in static memory.

4.2 ChaCha20 Encryption Cipher

Our secondary example is the `ChaCha20` encryption cipher. This proof was completed in one month by a team of four first-year graduate students who received roughly eight hours of training on Rocq and Picinæ. Due to limited availability of SSL libraries that compile to RISC-V, our ChaCha20 implementation is written by hand from the RFC [7]. This implementation contains a loop with a constant iteration count, as well as a function call, which required verifying the timing properties of call-return semantics. Its timing expression is parametrized only by the length of the plaintext, proving that the implementation of the algorithm is immune to timing attacks, assuming it is run on a cacheless, non-speculatively executing RISC-V processor.

4.3 Future Work

In future work we plan to automate the creation of timing invariants for a subset of common-case binary codes. This automation, accomplished by locating basic blocks and analyzing CFGs, will reduce the workload required to write new timing proofs, and automate most or all of the proof process for simple examples. Given their predictable structure, we expect that these invariants could even be generated by large language models. Because invariants remain untrusted (since they undergo machine verification), this sacrifices no assurance for the end-user.

The memory safety subproofs required for `vTaskSwitchContext` hint that a standardized representation of static memory layout, accompanied by supporting proof automation, could greatly reduce the proof load for memory-sensitive code.

Integration with common static analysis tools such as Ghidra will further simplify the interface for these proofs, offering the capability of high-assurance timing proofs for a larger audience.

Finally, comparing the times revealed in timing proofs against experiments run on real hardware will further support the abstract conclusions derived by our system.

References

[1] Daniel J. Bernstein. 2005. *Cache-timing Attacks on AES.* Technical Report. The University of Illinois at Chicago. cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[2] 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.

[3] Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. 2019. Source-free Machine-checked Validation of Native Code in Coq. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*. 25–30.

[4] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre Attacks: Exploiting Speculative Execution. *Communications of the ACM (CACM)* 63, 7 (2020), 93–101.

[5] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*. 104–113.

[6] National Security Agency. 2017. *P-Code Reference Manual.* spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html.

[7] Yoav Nir and Adam Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539.

[8] Stephan T. Nolting. 2025. The NEORV32 RISC-V Processor - Datasheet. stnolting.github.io/neorv32.

[9] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the the Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*. 1–20.

[10] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*. 46–57.

[11] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008).