Formally-Verified, Tight Timing Constraints for Machine Code

Charles Averill
charles@utdallas.edu
University of Texas at Dallas
Dallas, Texas, USA

Abstract

We introduce a dependently typed, machine-checked framework 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 are real-time systems, which must meet strict timing deadlines to ensure correct behavior, and 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 [13].

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 [14], 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 [7], 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, 11] and branch prediction or speculative execution attacks, such as Spectre and Meltdown [6]. 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.

PLDI SRC '25, June 16–20, 2025, Seoul, SK

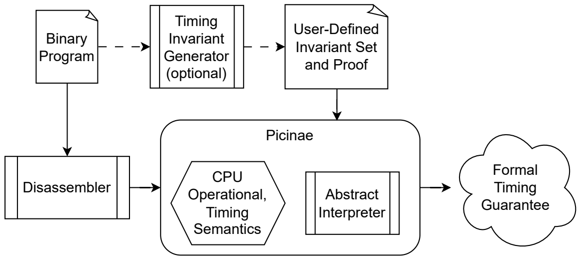
Binary Program
Timing Invariant Generator (optional)
User-Defined Invariant Set and Proof
Picinae
Disassembler
CPU Operational, Timing Semantics
Abstract Interpreter
Formal Timing Guarantee

Figure 1: Picinæ Timing Module Pipeline

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 might not accurately reflect true execution behavior. Cryptographic verification techniques, such as constant-time programming methodologies [9], 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.

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 [4], 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 (Picinæ IL) formalized within the Rocq proof assistant. Picinæ IL is similar to other ISA-modeling ILs, such as BIL [2] and P-code [8], 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 that encode the operational effect of the instruction at a on an abstract cpu state. The IL encodes effects via 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. Due to the genericity of these constructs, lifting to Picinæ IL is achievable for all forms of machine code, making Picinæ completely source language-agnostic.

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 non-deterministic 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 [12].

2.2 Existing Timing Approaches

2.2.1 Abstract Interpretation statically approximates program behavior by interpreting it with abstract values. This can determine an upper bound for execution times using control-flow analysis, where paths through the program are modeled as a set of constraints.

2.2.2 Measurement-Based Analysis involves executing the code on bare hardware or a simulator, and measuring the execution time for various input sets, recording the maximum execution time observed. Although this can more easily produce statistical 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. This enables a precise analysis of performance and resource utilization, crucial for ensuring predictable behavior in timing-sensitive 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 [10] that emphasizes analysis-amenable properties, such as non-speculative execution. Our approach is not suitable for systems using out-of-order execution. Instead, we target areas such as flight control, where manufacturers choose simpler cpus due to their predictability [3, 5].

3.1.3 Implementation. We encode instruction timings as a Rocq function that maps a machine 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. Because the computation maps to machine instructions, the abstraction of PicinæIL is bypassed, ensuring not only that the proof provides guarantees about the original binary, but that the approach is architecture-agnostic.

Instruction latency may also depend on the instructions that executed prior. Manufacturer-provided instruction WCET documentation expresses such timings as formulas over relevant cpu state elements, which are then incorporated into our Rocq timing function. 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 mapping the 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 universally quantify over traces, expressing properties of all possible executions.

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 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 current iteration is equal to $(c_0 - c)t$, where c_0, c, and t are the initial loop counter, the current loop

```
add:
    beqz  t0, end    ; 0  - goto end if t0 == 0
    addi  t1, t1, 1  ; 4  - increment t1
    addi  t0, t0, -1 ; 8  - decrement t0
    j     add        ; 12 - goto add
end:                 ; 16
```

Figure 2: Peano addition assembly code implementation

```
Definition timing_invs (p:addr) (x y:N) (t:trace) :=
 let tb := 5+(ML-1) in   (* time of a taken branch *)
 let ft := 3 in (* time of a fallen-through branch *)
 match t with (Addr a, s) :: t' ⇒ match a with
 | 0 ⇒ Some (s R_T0 ≤ x ∧
       cycle_count t' = (x - T0) * (ft + 2 + 2 + tb))
 | 16 ⇒ Some (
       cycle_count t' = tb + x * (ft + 2 + 2 + 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.                           (* abstract 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

counter, and the time of the loop body, respectively. The postcondition asserts that the total time taken is equal to $t_0 + c_0 t$.

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 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 present 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 and preservation subproofs were required for the timing expression to be parametrized only over the initial memory state. These subproofs are sufficiently difficult and small in scope that an automation tactic was necessary to reduce workload on the user. 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 [9]. 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 by plaintext length, proving that the implementation is immune to timing attacks, assuming it is run on a cacheless, non-speculatively executing RISC-V processor.

4.3 Future Work

Ongoing research into Picinæ timing proofs includes automating the creation of timing invariants for a subset of common-case binary codes. This automation, accomplished via CFG analysis and symbolic execution, 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.

Common patterns in the memory safety subproofs required for vTaskSwitchContext indicate that a standardized representation of memory layout with stronger supporting proof automation would 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] Advanced Micro Devices. [n.d.]. Am29000 and Am29005 Streamlined Instruction Microprocessors. https://datasheets.chipdb.org/AMD/29K/00x_ds.pdf.

[4] 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.

[5] Honeywell. [n.d.]. Pegasus FMS for Airbus A330 A320 Technical summary. https://aerospace.honeywell.com/content/dam/aerobt/en/documents/learn/platforms/brochures/C61-1647-000-000_ATR_TechnicalSummary_PegasusFMS_Airbus_A330_A320.pdf

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] DOLORES WALLACE and D. Kuhn. 2002. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering* 08 (07 2002). https://doi.org/10.1142/S021853930100058X

[14] 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).