Formally-Verified, Tight Timing Constraints for Machine Code

Charles Averill
https://www.charles.systems/ charles@utdallas.edu
Co-Authors: Ilan Buzzetti, Wei-Cheng Wu Advisors: Kevin Hamlen, Christophe Hauser
The University of Texas at Dallas, Dartmouth College

Motivation

Many mission-critical computing systems operate under tight timing constraints, where deviations in execution time can have severe consequences. Two important categories are **real-time systems**, which must meet timing deadlines to ensure correct behavior, and **cryptographic systems**, which must guard against timing side-channel attacks.

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, 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.

Background

Our work builds upon **Picinæ**, a framework within the Rocq prover for the development of functional correctness proofs for arbitrary (e.g., non-compiled) machine code. Picinæ lifts machine code into an intermediate language (IL) **formalized in Rocq**. The IL models instructions as state transformations using constructs like assignments, jumps, and bounded loops. It ensures strong normalization and is **source language-agnostic**, supporting all machine code forms. Invariant sets at specific program points enable inductive proofs of correctness and security. A symbolic interpreter executes IL abstractly, modeling unknowns via proof meta-variables. Dependent types enforce ISA-specific constraints, like bit-width bounds on registers. **Full code coverage** is ensured by introducing proof goals for all execution branches. Execution traces capture IL state transitions, enabling formal reasoning about temporal properties.

Abstract Interpretation approximates program behavior by interpreting it with abstract values. This can determine an upper bound for WCET using control-flow analysis, control-flow paths are modeled as constraint sets.

Measurement-Based Analysis involves executing the code on bare hardware or a simulator, and measuring the execution time for some inputs and recording the maximum time observed. This can more easily produce results for complex systems, but is not a comprehensive search over code paths and offers **no formal guarantees**. This method's precision can be improved by collecting more measurements by varying the initial processor state.

Instruction Timing

Picinæ measures instruction timing using cpu **cycle counts** for granular, consistent analysis. The **NEORV32 RISC-V** cpu is chosen for its detailed timing documentation and predictable, non-speculative execution, ideal for timing-sensitive domains like flight control.

Instruction timing is implemented as a Rocq function that maps each machine instruction's type, arguments, and state to a cycle count. Timing computations bypass Picinæ IL, operating directly on decoded machine instructions to maintain accuracy and **architecture-agnosticism**. Instruction latency can depend on previous instructions, so timing formulas incorporate cpu state variables as specified by cpu WCET documentation.

This approach is less feasible for complex architectures (e.g., x86), but could be generalized using disassemblers to **map instruction addresses to instruction times**. The resulting timing model is a **input state-parameterized formula** incorporating hardware-specific conditions and tolerances. Such formulas enable proving tight worst-case execution time bounds and verifying timing properties like absence of information leakage.

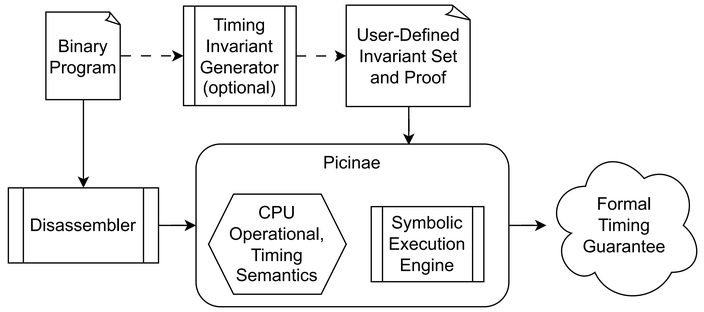

Figure 1. Picinæ Timing Module Pipeline

```
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

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 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.

```
Definition timing_invs (a : addr) (x y : N) (s : store) (t' : trace) :=
 match a with
 | 0 ⇒ Some (s R_T0 ≤ x ∧
   cycle_count t' = (x - s R_T0) * (t_fall + 4 + t_branch))
 | 16 ⇒ Some (cycle_count t' = t_fall + x * (t_fall + 4 + t_branch))
 | _ ⇒ None end.

Theorem addloop_timing:
 ∀ s trace, satisfies_all lifted_addloop invariants exit_point trace.
Proof.
  (* 0x4 - unfolded 'whammer' *) repeat step; psimpl; subst; lia.
  (* 0x8 (break/loop cases) *) whammer. whammer.
  (* 0x16 *) whammer. Qed.
```

Figure 3. Invariant set and proof for the `addloop` code in Fig. 2

```
Definition time_of_vTaskSwitchContext (t : trace) (gp : N) (mem : memory) :=
  if uxSchedulerSuspended =? 0 then
    cycle_count_of_trace t = (* total number of cycles equals... *)
    25 + 3 * time_branch + 17 * time_mem
      + (if (mem[4 + mem[gp - 920 + (31 - clz uxTopReadyPriority) * 20]])
          =? ((gp - 916) + (31   clz uxTopReadyPriority) * 20)
        then 22 + (clz uxTopReadyPriority) + 5 * time_mem
        else 19 + time_branch + (clz uxTopReadyPriority) + 3 * time_mem)
  else cycle_count_of_trace t = 5 + time_branch + 2 * time_mem.
```

Figure 4. Timing postcondition for `vTaskSwitchContext`

Evaluation

To demonstrate, we present examples of real-world code for which we have developed timing proofs.

Our first example, 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 seen in Fig. 4, as well as checks for stack overflows that block further execution when triggered. This timing expression is parametrized by several values in static memory.

Our second 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. Its timing expression is **parametrized only 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.

Future Work

Ongoing research into Picinæ timing proofs includes **automating the creation of some timing invariants**. This automation will, via CFG analysis and symbolic execution, reduce the workload required to write timing proofs, and automate most of the proof process for simple examples. Because **invariants remain untrusted**, this sacrifices no assurance for the end-user.

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.

Comparing the times revealed in timing proofs **against experiments run on real hardware** will further support the conclusions derived by our system.