Formally-Verified, Tight Timing Constraints for Machine Code

Charles Averill

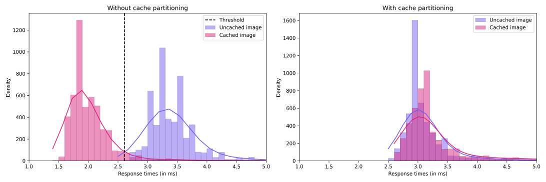
The University of Texas at Dallas
Dartmouth College

June 18, 2025

What's the Deal?

Often we'd like to know **exactly** how long code takes to run:

- In **real-time systems**, critical functions have strict timing constraints
- In **cryptographic systems**, differences in execution time of instructions or control-flow branches can expose secret data to attackers



What do we want?

There exist techniques to mitigate timing failures in real-time and cryptographic systems:

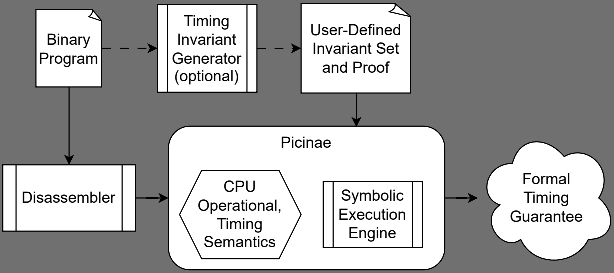
1. **Worst-Case Execution Time (WCET) Analysis** is a family of techniques that compute upper bounds for the execution time of code
2. **Constant-Time Cryptography** is the practice of writing sensitive cryptographic routines such that secret data is only used as an operand if it does not impact the resulting resource/time usage

Neither of these approaches gives any **formal guarantee** about the timing behavior of code.

For critical systems, we would like to be able to write a **machine-checked proof** of **exact timing behavior** at **arbitrary levels of precision**.
Enter the Picinæ Timing Module.

Picinæ

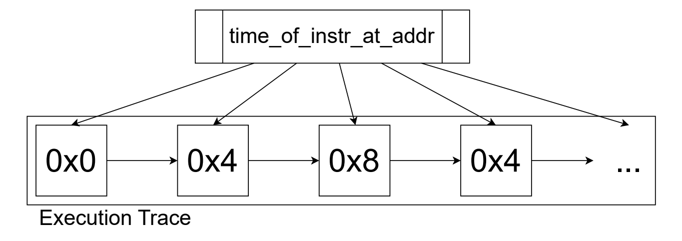
- Picinæ is an existing framework in Rocq for proving arbitrary properties of **machine code**
- Supports separation logic and linear temporal logic
- Lifts machine code into a Rocq-defined IL from Ghidra P-code, so **architecture-agnostic**
- **Full code coverage** ensured by introducing proof goals for all execution branches



Picinæ Timing Module

- We define a function mapping **instructions** to **cpu cycle counts**
- We map the instruction timing function **onto an LTL trace**
- The resulting property is a statement that **the number of clock cycles** that have occurred up to a point equals some **arithmetic expression**

```
cycle_count_of_trace t = time_memaccess +
    (if branch0_cond then 5 else 10) + ...
```



Timing Proofs

Picinæ proofs require the use of an **invariant set** - timing invariants just detail the time up to a certain point in execution.
Proofs of these invariant sets typically only require

1. Stepping forward to the next invariant
2. Simplifying
3. Proving an equality over natural numbers

Consider the following timing proof for an imperative implementation of Peano addition.

Addloop

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

If computing $x + y$, the loop should iterate exactly **x** times before execution completes.
At address 0, exactly $x - \texttt{t0}$ iterations should have occurred so far.

Addloop Proof

```
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 addloop timing_invs exits trace.
Proof.
  - repeat step; psimpl; subst; lia.
  - whammer. whammer.
  - whammer.
Qed.
```

Evaluation

- **vTaskSwitchContext** from FreeRTOS Kernel - timing postcondition references static memory values and contains complex arithmetic operations such as CLZ
  - Several more FreeRTOS timing proofs have since been written
- **ChaCha20** constant-time encryption cipher - shown to be **immune to timing attacks** because postcondition is **parametrized only by plaintext length**
  - Specification and proof performed by team of 4 first-year graduate students with 8 hours of instruction on Rocq and Picinæ, provided only with binary and assembly source

Future/Ongoing Work

- Automate creation of timing invariants due to their limited scope
- Compare timing proofs with real execution trace timing
- Utilize trace property composition to incorporate cpu caching behavior

Follow ongoing work at **www.charles.systems**