Ahead-of-Time Detection of Instruction Cache Timing Attacks With Towers of Interpreters

Jacob Ginesin

April 25, 2025

1 Motivation

Timing side channels stemming from micro–architectural state (*caches, branch predictors, . . .*) can break constant-time security arguments. Formal tools such as CBMC reason at the *C* level and remain unaware of these features unless we inject a faithful model [1]. The **LMS–Koika** project, the full implementation of the PEPM '25 paper "Collapsing Towers [of Interpreters] for Side-Channel Security" [2] approaches the problem by *staging interpreters*: each semantic refinement is coded as a Scala trait; composition produces both an executable reference model *and* a residual C program amenable to model checking [3].

Data caches and speculative execution are already modelled, yet the instruction cache (I-cache) is absent. We add an I-cache layer and show that it surfaces a new timing leak. Our code is publicly available via the following link: https://github.com/JakeGinesin/lms-koika

2 A Primer on "Towers"

The idea of "collapsing" interpreters into a compiler is fundamentally based on the famous 2nd *futamura projection* [4]. Given a `PartialEvaluator : Program, Input -> Program` that *specializes* the entered program for the given input, the three futamura projections are defined as follows:

1. `PartialEvaluator(Interpreter, Program) -> Program`. Returns an interpreter specialized for only the given program, essentially compiling it.
2. `PartialEvaluator(PartialEvaluator, Interpreter) -> Compiler`. Returns a compiler for any program the inputted interpreter can evaluate.
3. `PartialEvaluator(PartialEvaluator, PartialEvaluator) -> Compiler Generator`. Specializes the second inputted `Partial Evaluator` into a compiler generator.

A *tower of interpreters* is a stack where every level is an interpreter written in the language of the level beneath it. **Collapsing** the tower in practice, via the 2nd futamura projection, with multi-stage programming means:

1. evaluate as much as possible at *code-gen time*;
2. leave unknowns as residual C code;
3. still reason in Scala at a comfortable level of abstraction.

Each concern is packaged as a trait; mixing traits simply stacks features.

3 Micro-Architectural Model

The `NanoRISC` interpreter implemented by **LMS–Koika** maintains a notion of hardware state, which is captured with the following struct. We extend this struct to support instruction cache keys.

State Extension

```
@CStruct                        // changed lines in gray
case class StateT(
  regs: Array[Int],  mem: Array[Int],
  saved_regs: Array[Int],
  cache_keys: Array[Int], cache_vals: Array[Int],
  icache_keys: Array[Int],    // <-- NEW
  timer: Int)
```

Behaviour

- 2-entry LRU queue (`icache_keys(0)` = MRU).
- +1 cycle on a hit, +20 cycles on a miss.
- Cache lookup happens exactly once per static program counter: the trampoline entry `call(pc, state)`.

Simplified Implementation (Scala)

```
trait ICache extends KoikaInterp.Cached { self: Dsl =>
  private def updateICache(s: Rep[StateT], pc: Rep[Int]): Unit = {
    val hit0 = s.icache_keys(0) == pc
    val hit1 = s.icache_keys(1) == pc
    s.timer += (if (hit0 || hit1) 1 else 20)

    if (!hit0) {                              // LRU maintenance
      s.icache_keys(1) = s.icache_keys(0)
      s.icache_keys(0) = pc
    }
  }
  abstract override def call(i: Int, s: Rep[StateT]): Rep[StateT] = {
    updateICache(s, i); super.call(i, s)
  }
}
```

4 Leaking Example

The program in Listing 1 branches on secret register `r0`. The cold path (`pc=1`) is absent from the cache on first execution, incurring a +20 cycle penalty.

Listing 1: Branch-dependent timing

```
val branchICacheLeak = Vector(
  Mov(r2, Imm(0)),                  // pc 0 : always executed
  Mov(r1, Imm(42)),                 // pc 1 : only if r0 != 0  (miss)
  Mov(r1, Imm(99)),                 // pc 2
  B(Some((Eq, r0, Imm(0))), Addr(4)), // pc 3
  Mov(r1, Imm(123))                 // pc 4
)
```

5 Verification with CBMC

The automated driver runs the program twice (*secret kept constant*) and asserts equal `timer`. CBMC (command below) finds a counterexample where the branch outcome differs:

Listing 2: CBMC run

```
$ cbmc -DCBMC icache_icache_leak.c
** Results:
timing leak function main ... failed
VERIFICATION FAILED
```

6 Conclusion

In this work we modeled an instruction cache in Scala, folding it into the existing **LMS–Koika** infrastructure to compose it with existing layers, thus making its timing effects visible to implementation-level symbolic model checking. The work highlights the advantage of the tower approach: micro-architectural refinements are local, combinable, and automatically propagated to the verification artefact. In the future, we hope to extend this work in a few ways, namely (1) emitting Rust code as opposed to C code, (2) implementing a more detailed and more realistic notion of hardware state, (3) integrating our work with existing interpreters for hardware description languages, and (4) folding the towers methdology into existing program verification pipelines.

References

[1] D. Kroening, P. Schrammel, and M. Tautschnig, "CBMC: The C Bounded Model Checker," no. arXiv:2302.02384, Feb. 2023, arXiv:2302.02384 [cs]. [Online]. Available: http://arxiv.org/abs/2302.02384

[2] C. Wong, M. Abdullah, Y. Yang, M. Yan, A. Chlipala, and N. Amin, "Collapsing Towers for Side-Channel Security (Short Paper)," 2025.

[3] N. Amin, "lms-koika: Collapsing Towers for Side-Channel Security," https://github.com/namin/lms-koika, 2025.

[4] Y. Futamura, "Partial Evaluation of Computation Process, Revisited," *Higher-Order and Symbolic Computation*, vol. 12, no. 4, p. 377–380, Dec. 1999.