Leakage models are a leaky abstraction: the case for cycle-level verification of constant-time cryptography

Anish Athalye
M. Frans Kaashoek
Nickolai Zeldovich
MIT CSAIL

Joseph Tassarotti
New York University

Abstract

We propose abandoning leakage models for verifying timing properties of cryptographic software, instead directly verifying software with respect to a hardware implementation at the RTL level. Early experiments include verifying that an Ed25519 implementation running on a 6-stage pipelined processor executes in a constant number of cycles. Many challenges remain, including scaling up to modern out-of-order speculative cores and extending the approach to reason about library code outside the context of a whole application.

1 Introduction

Formal verification is a promising approach for ruling out timing side channels in cryptography. The end goal is to ensure that cryptographic software runs in constant wall-clock time when executed by hardware, in order to defend against timing attacks [7].

Today's formally-verified cryptographic software falls short of this goal because it assumes a leakage model [2] where there is a gap between the leakage model and the actual hardware behavior. Fiat Crypto [10] uses a code generation approach that emits straight-line code using instructions that are assumed to have an input-independent execution time. HACL [20] uses the type system to only allow certain primitive operations on secrets, which are assumed to be implemented by the underlying hardware in constant time. Almeida et al. [1] define a leakage model that includes branches and memory access locations, and verification ensures that a leakage trace is independent of secrets. Vale [6] defines a similar leakage model but also includes the inputs to an assumed set of variable-latency instructions. EverCrypt [16] inherits the assumptions of HACL and Vale.

In proposing a leakage model, these approaches make simplifying assumptions about the underlying hardware that are either unverified or even untrue on modern processors. For example, HACL's interface for secrets omits integer division, because that operation is not constant-time on most architectures; but it does not avoid integer multiplication, which can also be variable-time on some ARM and i386 platforms.

The gap between the hardware model and the actual hardware behavior allows bugs to slip through. For example, OpenSSL contained crypto code that was provably secure under a baseline leakage model (where the adversary observes program counter addresses and memory access addresses) but insecure under a model that accounts for time-variable arithmetic operators [3].

Recent work has proposed more sophisticated leakage models [9, 11, 13] and validated hardware against these models through fuzzing [8, 14, 15], sometimes revealing gaps between leakage models and hardware implementations. Today's verified cryptographic software is not verified against these sophisticated leakage models. In concurrent work, Wang et al. [19] formally verify simple open-source RISC-V processors against leakage contracts.

We propose an alternative approach for obtaining end-to-end timing guarantees: verifying the timing behavior of software directly against the hardware. This cycle-level approach verifies software against a particular processor implementation for which the complete cycle-level behavior is known and made visible to the verification tool.

To evaluate the approach, we built Chroniton, a tool that uses full-circuit symbolic execution to analyze timing behavior of crypto code running on hardware. Early experiments show promise: Chroniton can verify that an off-the-shelf implementation of Ed25519 running on the biRISC-V signs messages in a constant number of cycles, independent of the private key. A major open challenge remains: how to use such a cycle-level approach to verify timing behavior of individual functions, e.g., a signature operation, that can be used in an application that is unknown at verification time. We have open-sourced Chroniton and the case studies at https://github.com/anishathalye/chroniton.

2 Approach

We propose analyzing the timing behavior of software with respect to a particular hardware *implementation* rather than a model of the hardware, eschewing the traditional hardware/software separation enabled by the ISA boundary and leveraged by most verification approaches. The benefit is that our approach does not require making assumptions about the hardware's timing behavior. Because the analysis needs to be repeated for every hardware target, we aim for a highly-automated approach.

We reason about a processor running a complete program that includes some cryptographic code, like Figure 1.

Tools like Icarus Verilog and Verilator are capable of cycle-accurate simulations of processors running such code. These simulators operate on concrete values: every bit is a 0 or a 1,

```
#include "ed25519.h"

#define MSG_SIZE 100
unsigned char pk[32], sk[64], buf[MSG_SIZE], sig[64];

void main() {
  ed25519_sign(sig, buf, sizeof(buf), pk, sk);
}
```

Figure 1. A program that invokes cryptographic functionality. Before `main()` runs, assembly code boots up the processor and sets up an environment for C code.

so they are not directly useful for verifying timing properties, such as how execution time depends on the private key.

We use a cycle-accurate *symbolic* hardware simulator to reason about processors executing cryptographic code. Using a symbolic simulator, we can mark state elements in the circuit, e.g., locations in the processor's memory such as those corresponding to `sk[]` and `pk[]`, as symbolic variables, and then symbolically execute the entire circuit and determine whether the circuit's execution time depends on these symbolic variables.

Symbolic simulation can reason about the number of cycles a circuit, starting from any symbolic starting state, takes to reach a state satisfying a property such as `main()` returning. We apply this to reason about the execution of a program from start to finish, where the processor starts from a fairly constrained state or is reset before running the program to completion. This fits certain classes of devices, such as hardware security modules (HSMs) or cryptographic accelerators like the OpenTitan Big Number Accelerator[1] (OTBN). The OTBN clears state and runs each cryptographic operation without interruption from start to finish. HSMs can also be architected and verified in this way [5].

The approach requires that the complete hardware implementation (e.g., Verilog code) of the processor is available to the verification tool. It only reasons about a single clock domain. Reasoning about timing in terms of cycles does not rule out vulnerabilities arising from dynamic voltage and frequency scaling, like Hertzbleed [18].

3 Implementation

We implemented a verification tool called Chroniton, which consists of about 100 lines of Rosette [17] code. It builds on the Verilog-to-Rosette toolchain from Notary [4, 12], which translates circuits written in Verilog into cycle-accurate Rosette models supporting symbolic execution.

4 Evaluation

We applied Chroniton to verifying constant-time execution of an off-the-shelf Ed25519 signature routine[2] on a variety of simple off-the-shelf RISC-V processors / SoCs:

- PicoRV32 (https://github.com/YosysHQ/picorv32): an extremely simple size-optimized CPU
- biRISC-V (https://github.com/ultraembedded/biriscv): a 6-stage pipelined dual-issue embedded CPU

We set up the processors to run the code in Figure 1, marking the circuit state corresponding to the private key as symbolic, and used Chroniton to symbolically execute over millions of simulated hardware cycles to verify that the code runs in a constant number of cycles, regardless of the private key. Chroniton, on the slowest example (the biRISC-V), took 24 hours to verify the constant-time property. We also used Chroniton to verify an X25519 implementation on a simplified version of the OTBN accelerator, where Chroniton takes 10 hours to verify the constant-time property. We have not yet applied our tool/approach to modern desktop/server-class processors, which are significantly more complicated than the simple embedded processors we used.

5 Discussion and Open Problems

Our approach reasons about the timing properties of a circuit, including its hardware and software, starting from a particular (partially symbolic) state. We apply it to whole programs, which include cryptographic code, and reason about execution from start to finish.

We hope to extend the approach to cryptographic routines in isolation, such that we can verify library code against hardware and then use it in an application that is unknown at verification time. Currently, Chroniton naturally supports bug-finding but not verification in this setting: a crypto library developer (e.g., the developer of an Ed25519 library) can verify that a particular application that invokes library code runs in constant time (like the code in Figure 1). Chroniton proves that the application runs in constant time, which increases confidence in the constant-timedness of the library code, but it doesn't prove that the library code runs in constant time on the given hardware *in all contexts*.

Chroniton supports marking parts of a circuit's state as symbolic, so e.g., a processor's branch predictor state could be made symbolic, but for sophisticated processors, it's not clear how to set up the circuit state to capture all possible contexts without being under-constrained. We hope to extend Chroniton such that it supports *proving* properties about timing behavior of library code on hardware.

[1] https://opentitan.org/book/hw/ip/otbn/index.html

[2] https://github.com/orlp/ed25519

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Proceedings of the 23rd International Conference on Fast Software Encryption (FSE)*. Bochum, Germany, 163–184.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, 53–70.

[3] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-grained Constant-time Policies. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, 83–96.

[4] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: A Device for Secure Transaction Approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, 97–113.

[5] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, 503–519.

[6] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada, 917–934.

[7] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium*. Washington, DC, 1–13.

[8] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. 2021. Validation of Side-Channel Models via Observation Refinement. In *Proceedings of the 42th IEEE/ACM International Symposium on Microarchitecture*. Athens, Greece, 578–591.

[9] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, 913–926.

[10] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, 73–90.

[11] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. Virtual conference, 1868–1883.

[12] Noah Moroze, Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. rtlv: push-button verification of software on hardware. In *Proceedings of the 5th Workshop on Computer Architecture Research with RISC-V (CARRV)*. Virtual conference.

[13] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. New York, NY, 72–86.

[14] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 226–239.

[15] Oleksii Oleksenko, Marco Guarnieri, Boris Kopf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*. San Francisco, CA, 1737–1752.

[16] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. San Francisco, CA, 983–1002.

[17] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 530–541.

[18] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *Proceedings of the 31st USENIX Security Symposium*. Boston, MA, 679–697.

[19] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. https://arxiv.org/abs/2305.06979.

[20] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.