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

Anish Athalye¹, M. Frans Kaashoek¹, Nickolai Zeldovich¹, Joseph Tassarotti² ¹ MIT CSAIL ² New York University



Crypto code is vulnerable to timing side channels

Dedicated hardware provides strong security for applications

- Simpler than desktop/server-class processors
- Runs crypto code without sharing or interruption
- Eliminates many side channels by design

Remaining challenge: overall request timing

Remote timing attacks are practical (Brumley & Boneh, 2003)









Thales Luna PCIe HSM

Apple T2 security chip

YubiKey

Google OpenTitan **Big Number Accelerator**





State of the art: formal verification against leakage models

Prove that leakage is independent of secrets

Adversary observes PC addresses, memory access addresses, and operands of arithmetic ops

Most verified cryptography follows this approach (or similar)

Almeida et al. 2016, Vale (Bond et al. 2017), HACL (Zinzindohoué et al. 2017), Fiat Crypto (Erbsen et al., 2019), EverCrypt (Protzenko et al. 2020)

Downsides

Gap between leakage model and actual hardware behavior: might miss leakage, might be too conservative

Analysis of high-level language doesn't apply to systems code (access to control registers, peripherals)

Trust compiler to preserve constant-time behavior

Our plan: verify timing of software directly against hardware

Prove that...

a particular hardware implementation (RTL-level) runs a particular program (binary, memory image) in constant time (cycles)

for all inputs



Contributions

Chroniton, tool to verify software timing behavior against hardware RTL

Evaluation on off-the-shelf software + hardware: Ed25519 portable C implementation + {PicoRV32, Ibex, biRISC-V} X25519 OTBN assembly implementation + OpenTitan Big Number Accelerator

more info + code at <u>anish.io/chroniton</u>

An approximation: testing/fuzzing in RTL-level simulation





Approach: exhaustive testing using symbolic execution



The core: a symbolic RTL simulator

Compile Verilog HDL to Rosette (Torlak & Bodik 2014) Code

Rosette: solver-aided programming language built on top of Racket

Cycle-level circuit simulation, with concrete or symbolic state

Verilog to Rosette compilation

```
module counter (
    input clk,
    input en,
    output reg [31:0] counter
);
```

```
always @(posedge clk)
if (en)
counter <= counter + 32'h1;</pre>
```

endmodule

Verilog code

```
(struct state (...))
                          (define (new-symbolic-state)
                            ...)
                          (define (step state)
                            ...)
compile to state machine
representation in Rosette
                          (define (with-input state input)
                            ...)
                          (define (get-output state)
                            ...)
                                         Rosette code
```

Concrete evaluation of circuits

(define s (new-zeroed-state))

state { counter: (bv #x00000000 32) }

(step (with-input s (input 'en #t)))

state { counter: (bv #x00000001 32)

Symbolic evaluation of circuits

(define s (new-symbolic-state))

state { counter: counter\$4d1 }

(step (with-input s (new-symbolic-input)))

state {

counter: (ite en\$f7c (bvadd (bv 1 32) counter\$4d1) counter\$4d1)

Symbolic execution of software on hardware

Can have partially concrete, partially symbolic circuit state

Compiled binary loaded into circuit's ROM

What we are symbolically executing: circuit's step function

```
state {
 cpu.alu_out_q: (ite (bveq (bv #b1 1) soc.cpu.is_lui_auipc_jal_jalr_addi_add_sub$bd7) ...)
 cpu.cpu_state: (bv #x40 8)
 cpu.decoded_imm: (ite (&& (bveq (bv #b1 1) soc.cpu.decoder_trigger$caf) ...) ...)
 cpu.decoded_imm_j: soc.cpu.decoded_imm_j$4da
 cpu.decoded_rs2: soc.cpu.decoded_rs2$92e
  • • •
 cpu.cpuregs:
    0: soc.cpu.cpuregs[0]$e57
    1: soc.cpu.cpuregs[1]$a0f
    • • •
  ram:
    0: soc.ram.ram[0]$a12
    1: soc.ram.ram[1]$fe8
    • • •
  rom:
    0: (bv #x20001117 32)
    1: (bv #x80010113 32)
    2: (bv #x014000ef 32)
    3: (bv #x070000ef 32)
    4: (bv #x0ff00513 32)
    5: (bv #x05c000ef 32)
    • • •
```



Verifying timing behavior

Make input data symbolic

Just some bytes in data memory

Count cycles until hardware finishes executing

Check that completion time is independent of symbolic variables

That's all we need for basic examples!

Ed25519 on PicoRV32, verified to run in 4,046,295 cycles

#include "ed25519.h"

#define MSG SIZE 100 unsigned char pk[32], sk[64], msg[MSG_SIZE], sig[64];

void main() { ed25519_sign(sig, msg, sizeof(msg), pk, sk);

The case for cycle-level verification

Precise analysis

High confidence in non-leakage: verify directly against hardware / RTL

Not too conservative: directly verify timing behavior

Reason about any code (compiled binary)

Use any hardware features, CSRs, peripherals; eliminate trust in compiler

Limitations / open problems

Verify only simpler embedded CPUs (suits the application domain of HSMs and accelerators) Repeat verification for each hardware target (which is why we use automation) Programs verified end-to-end, can we say something about libraries / program fragments?

Case studies: high confidence in non-leakage

Hardware	Software	Cycles	Verification time (single-threaded)	LOC of hints
PicoRV32	Ed25519	4,046,295	2 hours	0
biRISC-V	Ed25519	692,287	24 hours	10
OpenTitan Big Number Accelerator (OTBN)	X25519	114,490	10 hours	5



Case studies: not overly conservative

```
if (secret) {
    *result = a + b;
    asm volatile(
        "beq zero, zero, 0f \n\t"
        "0: \n\t"
    );
} else {
    *result = *a - *b;
    asm volatile("nop");
}
```

Code running on PicoRV32

- Constant-time cryptography and parsing avoid branching on secrets, even when convenient
- Verified constant-time on PicoRV32
 - Need different padding for biRISC-V

Why it works: minimizing symbolic branching

- Code already written to be constant time
- Hardware has natural separation between control and data path
 - Conceptually, executing concrete program on symbolic data
 - Circuit's control state stays concrete while data is symbolic
 - Repeated symbolic evaluation of circuit's step function doesn't blow up

Rosette

- Hybrid symbolic evaluation best of both worlds of symbolic execution + model checking
- Rewrite rules optimizations that simplify terms



Hints minimize symbolic evaluation

Minimizing symbolic evaluation

overapproximate — often don't need to track precise expressions to reason about timing

Optimizing symbolic branching

concretize — to invoke the solver to concretize control state

case-split — perform case analysis, explore branches separately

Summary

Verify timing behavior of software directly against hardware (RTL)

Using symbolic execution + a bit of human guidance

- Examples include Ed25519 on biRISC-V (6-stage dual-issue RISC-V processor)

code, docs, and expository examples: anish.io/chroniton