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)
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)


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 [anish.io/chroniton](anish.io/chroniton)
An approximation: testing/fuzzing in RTL-level simulation

```
#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);
}
```

```
module riscv_core

    // Inputs
    input    clk_i
    input    rst_i
    input    [ 31:0] mem_d_data_rd_i
    input    mem_d_accept_i
    input    mem_d_ack_i
    input    mem_d_error_i
    input    mem_d_resp_tag_i
    input    mem_i_accept_i
    input    mem_i_valid_i
    input    mem_i_error_i
    input    mem_i_inst_i
    input    intr_i
    input    [ 31:0] reset_vector_i
    input    [ 31:0] cpu_id_i

    // Outputs
    output   [ 31:0] mem_d_addr_o
    output   [ 31:0] mem_d_data_wr_o
```

Software
- .c
- .s

Hardware
- .v

Compile
- firmware.hex

RTL simulator

Test if it runs in constant time (on a specific concrete input)
Approach: exhaustive testing using symbolic execution

```c
#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);
}
```

Software
.c, .s

Hardware
.v

compile

Binary firmware.hex

Chroniton (symbolic RTL simulation)

✅ / ❌ formally verify that it runs in constant time (for all inputs)

```vhdl
module riscv_core
  // Inputs
  input clk_i,
  input rst_i,
  input [31:0] mem_d_data_rd_i,
  input mem_d_accept_i,
  input mem_d_ack_i,
  input mem_d_error_i,
  input mem_d_resp_tag_i,
  input mem_i_accept_i,
  input mem_i_valid_i,
  input mem_i_error_i,
  input mem_i_inst_i,
  input instr_i,
  input reset_vector_i,
  input [31:0] cpu_id_i,
  // Outputs
  output [31:0] mem_d_addr_o,
  output [31:0] mem_d_data_wr_o
```

$readmemh("firmware.hex", rom)
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
module counter (
    input clk,
    input en,
    output reg [31:0] counter
);

always @(posedge clk)
    if (en)
        counter <= counter + 32'h1;
endmodule

(struct state (...

(define (new-symbolic-state) ...

(define (step state) ...

(define (with-input state input) ...

(define (get-output state) ...)
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*

SoC state, including CPU and memory state
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

```c
#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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>Cycles</th>
<th>Verification time (single-threaded)</th>
<th>LOC of hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>PicoRV32</td>
<td>Ed25519</td>
<td>4,046,295</td>
<td>2 hours</td>
<td>0</td>
</tr>
<tr>
<td>biRISC-V</td>
<td>Ed25519</td>
<td>692,287</td>
<td>24 hours</td>
<td>10</td>
</tr>
<tr>
<td>OpenTitan Big Number Accelerator (OTBN)</td>
<td>X25519</td>
<td>114,490</td>
<td>10 hours</td>
<td>5</td>
</tr>
</tbody>
</table>
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");
}

Constant-time cryptography and parsing avoid branching on secrets, even when convenient

Verified constant-time on PicoRV32

Need different padding for biRISC-V

Code running on PicoRV32
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