# The K2 Architecture for Trustworthy Hardware Security Modules

Anish Athalye[1], M. Frans Kaashoek[1], Nickolai Zeldovich[1], Joseph Tassarotti[2]

[1] MIT CSAIL [2] New York University

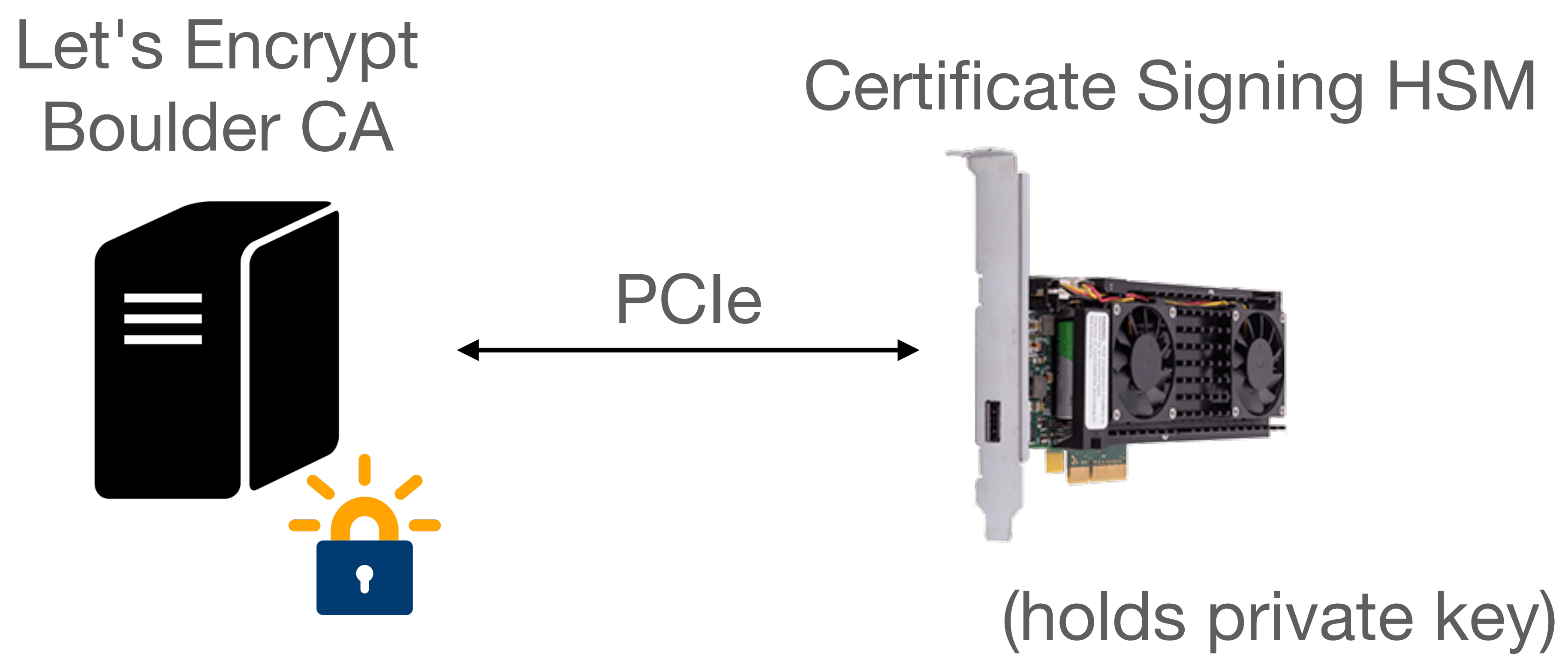
# HSMs: powerful tools for securing systems

Factor out core security operations

Provide security under host compromise

Many types of HSMs

- U2F token
- iPhone Secure Enclave
- PKCS#11 HSM
- WhatsApp Backup Key Vault



Let's Encrypt Boulder CA

PCIe

Certificate Signing HSM

(holds private key)

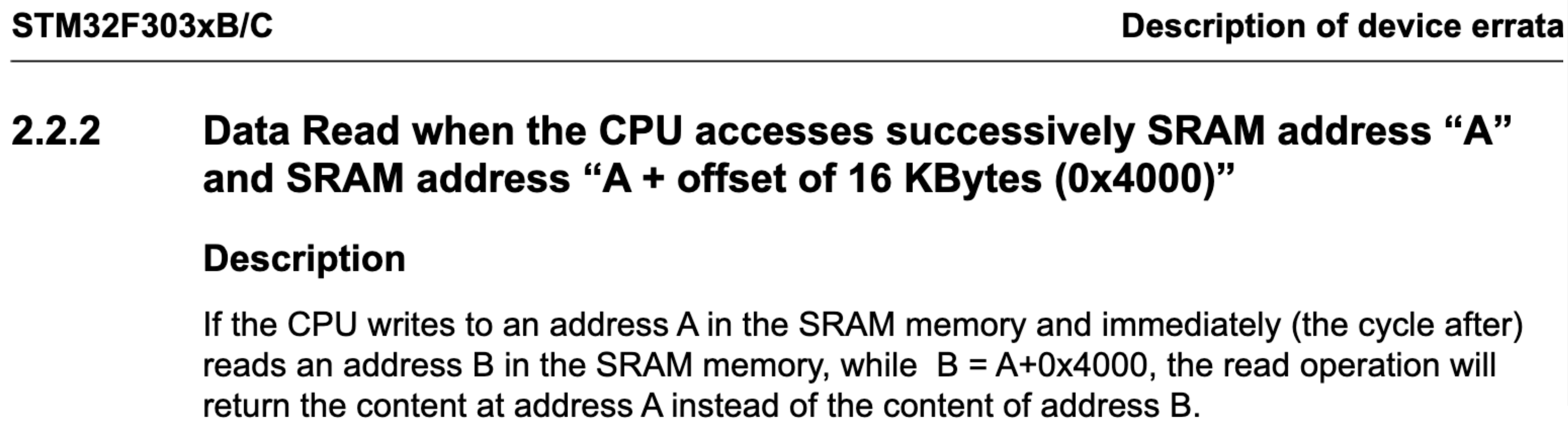
Hundreds of millions of deployed HSMs

# HSMs suffer from bugs

Hardware

Software

Timing side channels

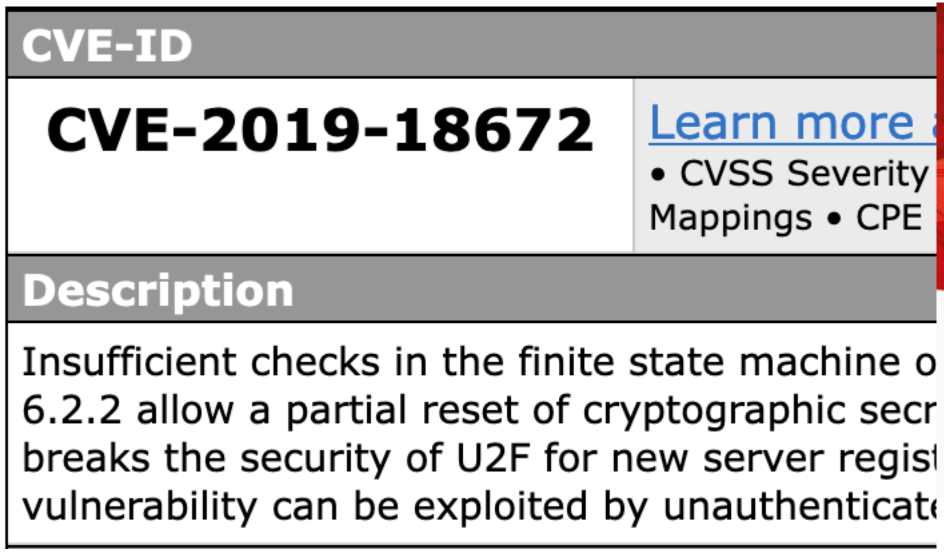


STM32F303xB/C Description of device errata

2.2.2 **Data Read when the CPU accesses successively SRAM address “A” and SRAM address “A + offset of 16 KBytes (0x4000)”**

**Description**

If the CPU writes to an address A in the SRAM memory and immediately (the cycle after) reads an address B in the SRAM memory, while B = A+0x4000, the read operation will return the content at address A instead of the content of address B.

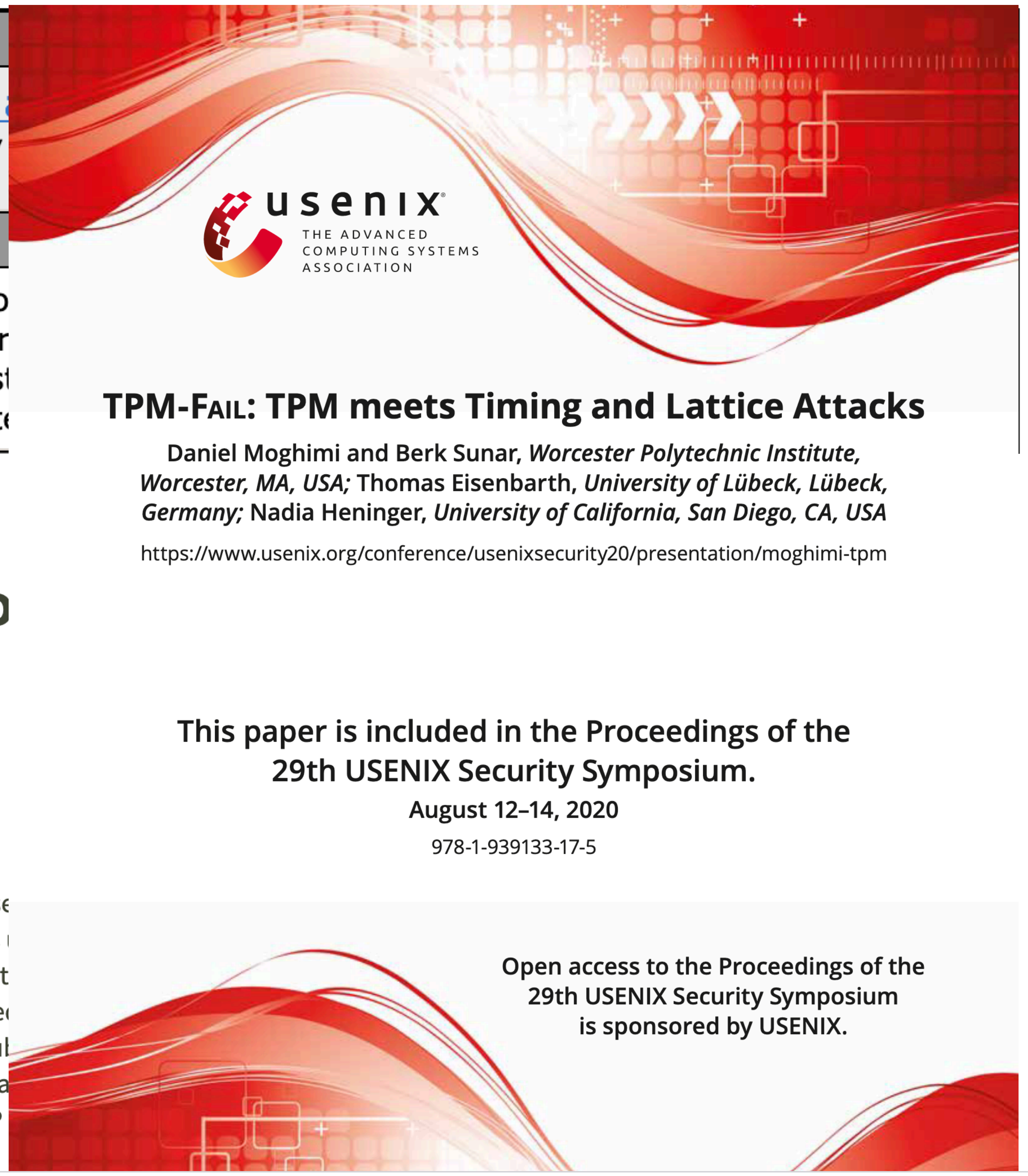


**CVE-ID**

**CVE-2019-18672** Learn more
• CVSS Severity
Mappings • CPE

**Description**

Insufficient checks in the finite state machine o
6.2.2 allow a partial reset of cryptographic secr
breaks the security of U2F for new server regis
vulnerability can be exploited by unauthenticat



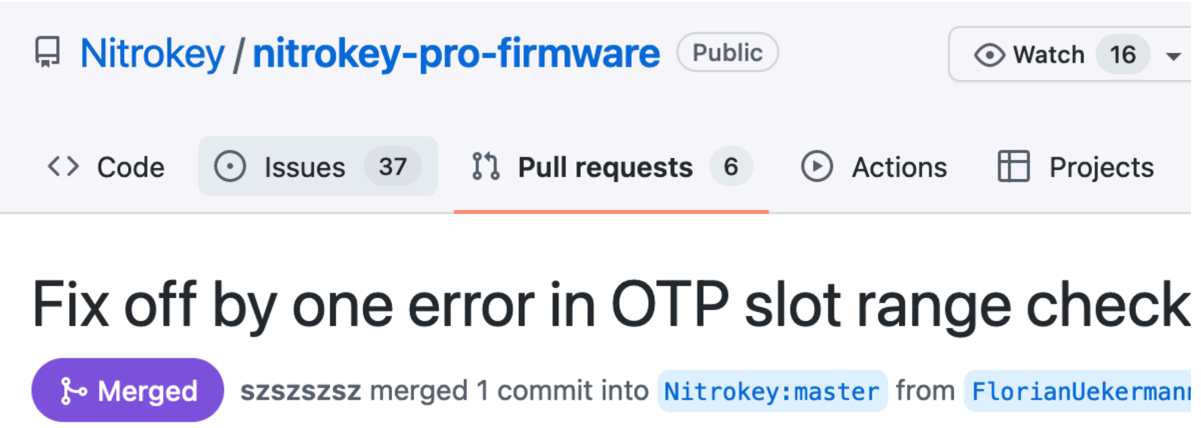
TPM-Fail: TPM meets Timing and Lattice Attacks

Daniel Moghimi and Berk Sunar, *Worcester Polytechnic Institute, Worcester, MA, USA;* Thomas Eisenbarth, *University of Lübeck, Lübeck, Germany;* Nadia Heninger, *University of California, San Diego, CA, USA*

https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm

This paper is included in the Proceedings of the 29th USENIX Security Symposium.
August 12–14, 2020
978-1-939133-17-5

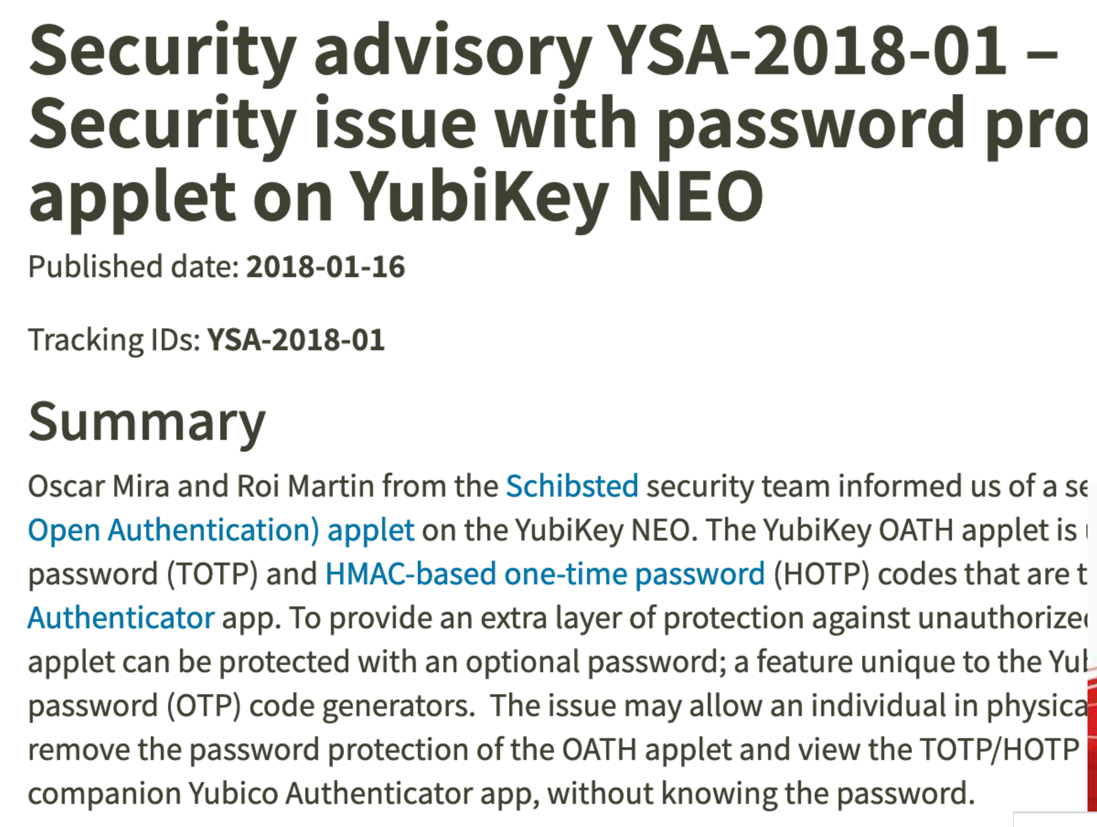
Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.



Nitrokey / nitrokey-pro-firmware Public

Fix off by one error in OTP slot range check

Merged szszszsz merged 1 commit into `Nitrokey:master` from `FlorianUekermann`



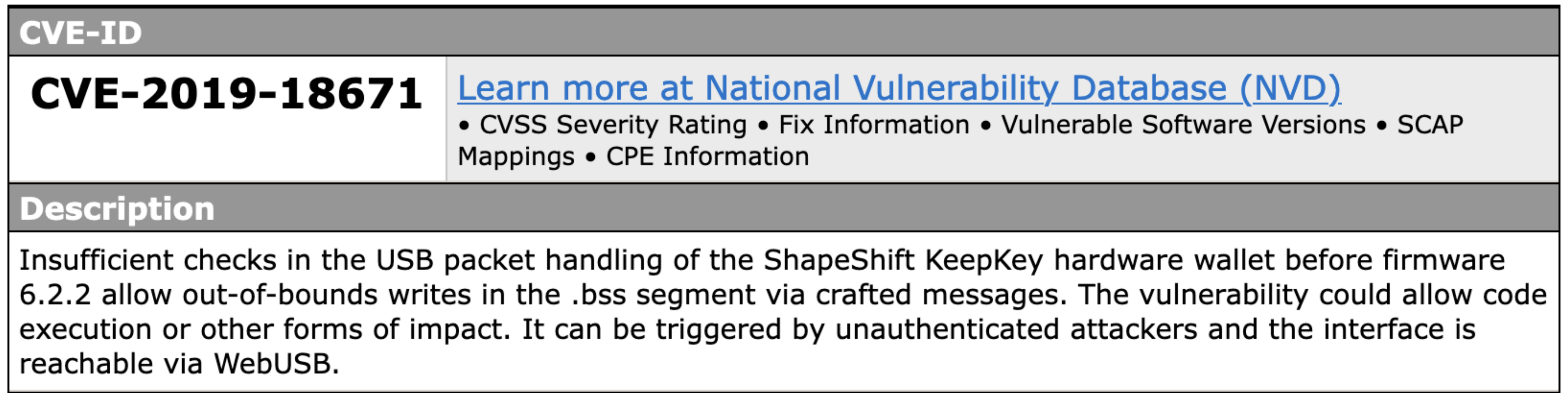
## Security advisory YSA-2018-01 – Security issue with password pro applet on YubiKey NEO

Published date: **2018-01-16**

Tracking IDs: **YSA-2018-01**

## Summary

Oscar Mira and Roi Martin from the Schibsted security team informed us of a se Open Authentication) applet on the YubiKey NEO. The YubiKey OATH applet is password (TOTP) and HMAC-based one-time password (HOTP) codes that are t Authenticator app. To provide an extra layer of protection against unauthorize applet can be protected with an optional password; a feature unique to the Yul password (OTP) code generators. The issue may allow an individual in physica remove the password protection of the OATH applet and view the TOTP/HOTP companion Yubico Authenticator app, without knowing the password.

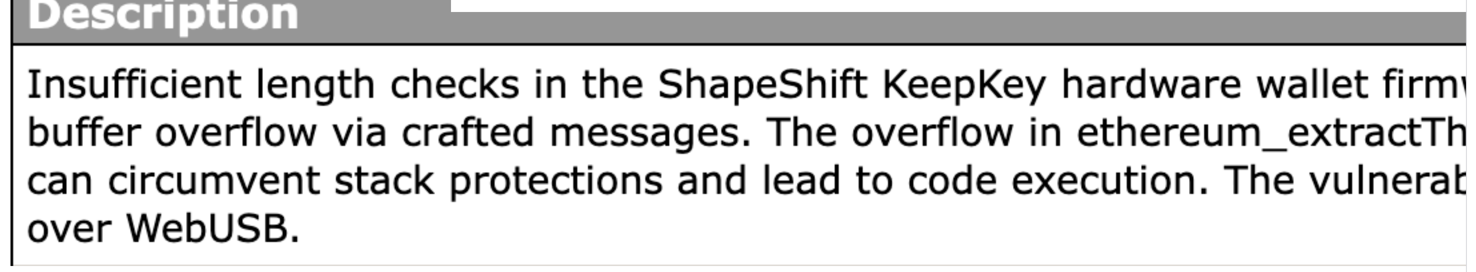


**CVE-ID**

**CVE-2019-18671** Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

**Description**

Insufficient checks in the USB packet handling of the ShapeShift KeepKey hardware wallet before firmware 6.2.2 allow out-of-bounds writes in the .bss segment via crafted messages. The vulnerability could allow code execution or other forms of impact. It can be triggered by unauthenticated attackers and the interface is reachable via WebUSB.

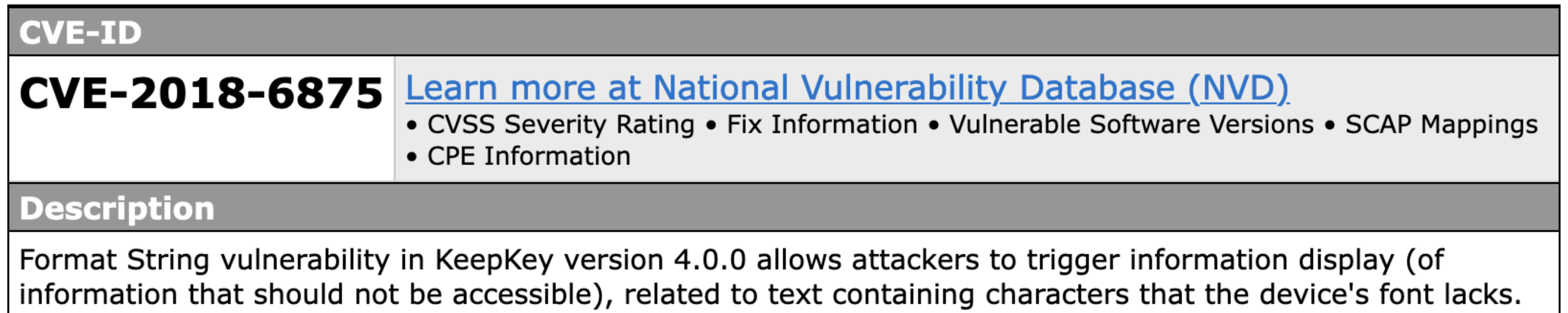


**CVE-ID**

**CVE-2021-31**

**Description**

Insufficient length checks in the ShapeShift KeepKey hardware wallet firm buffer overflow via crafted messages. The overflow in ethereum_extractTh can circumvent stack protections and lead to code execution. The vulnerab over WebUSB.

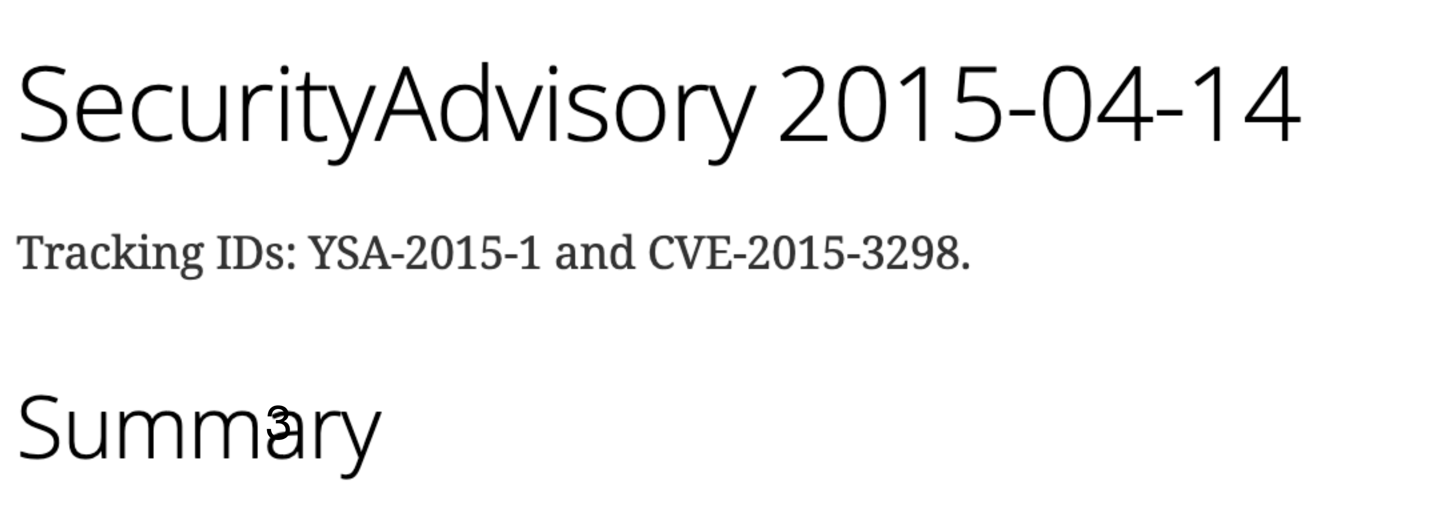


**CVE-ID**

**CVE-2018-6875** Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

**Description**

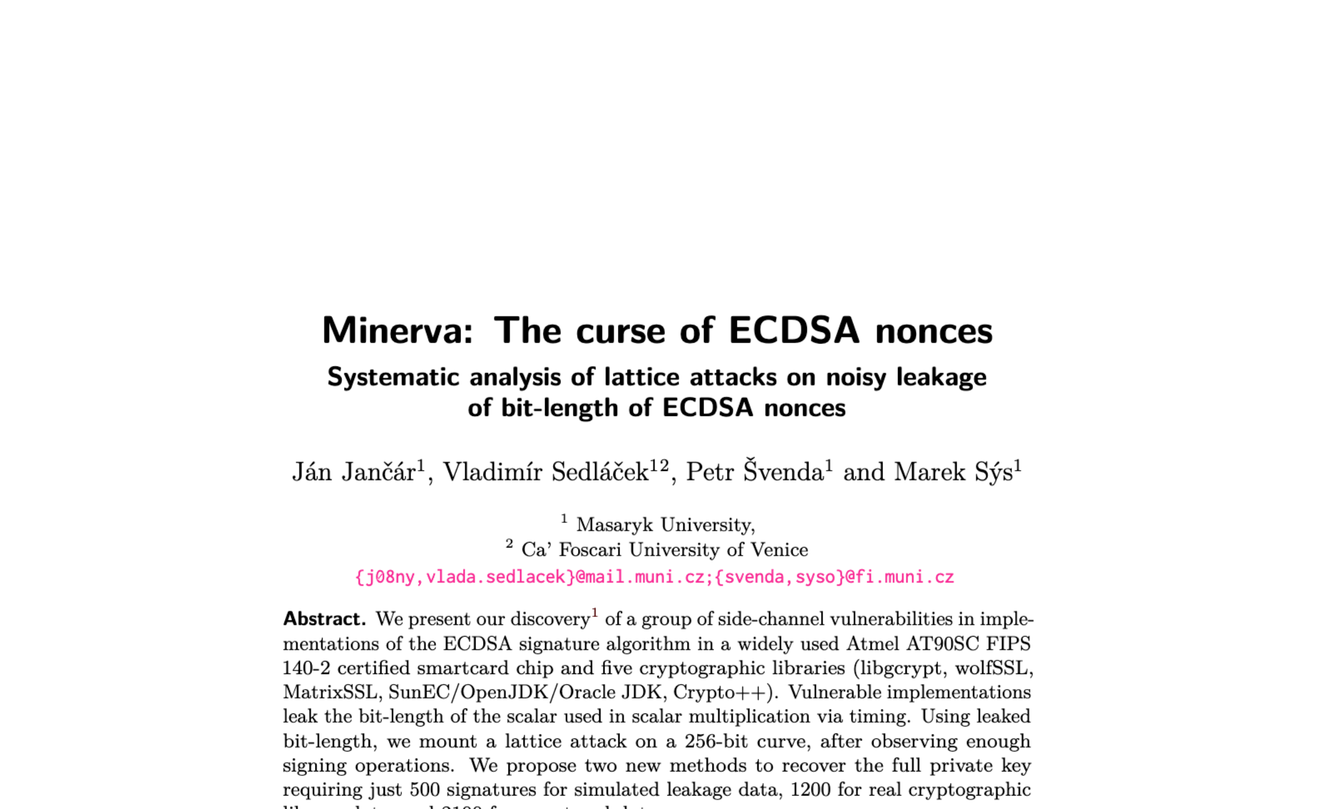
Format String vulnerability in KeepKey version 4.0.0 allows attackers to trigger information display (of information that should not be accessible), related to text containing characters that the device's font lacks.



# SecurityAdvisory 2015-04-14

Tracking IDs: YSA-2015-1 and CVE-2015-3298.

## Summary



**Minerva: The curse of ECDSA nonces**
**Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces**

Ján Jančár[1], Vladimír Sedláček[12], Petr Švenda[1] and Marek Sýs[1]

[1] Masaryk University,
[2] Ca’ Foscari University of Venice
{j08ny,vlada.sedlacek}@mail.muni.cz;{svenda,syso}@fi.muni.cz

**Abstract.** We present our discovery[1] of a group of side-channel vulnerabilities in implementations of the ECDSA signature algorithm in a widely used Atmel AT90SC FIPS 140-2 certified smartcard chip and five cryptographic libraries (libgcrypt, wolfSSL, MatrixSSL, SunEC/OpenJDK/Oracle JDK, Crypto++). Vulnerable implementations leak the bit-length of the scalar used in scalar multiplication via timing. Using leaked bit-length, we mount a lattice attack on a 256-bit curve, after observing enough signing operations. We propose two new methods to recover the full private key requiring just 500 signatures for simulated leakage data, 1200 for real cryptographic

# Goal: HSMs without security vulnerabilities

Rule out hardware, software, and timing side-channel vulnerabilities

Threat model

- Powerful adversary that gains control of host machine
- Full control over I/O interface to HSM
- Physical attacks and other side channels: out of scope

# Challenge: timing side channels at hardware level

Cryptographic constant-time software not enough

Tricky hardware timing behavior

> "ARM Cortex M3: manual says `umull` opcode takes 3 to 5 cycles, the 'short' counts (3 or 4) being taken only if both operands are numerically less than 65536... measurements show that short cycle count could occur not only in the documented case, but also when one or both of the operands is zero or a power of 2"

System software, CSRs, I/O, peripherals, and persistent storage

# Prior work: Knox [OSDI'22] / Information-Preserving Refinement

## Information-Preserving Refinement (IPR)

Implementation leaks no more than specification

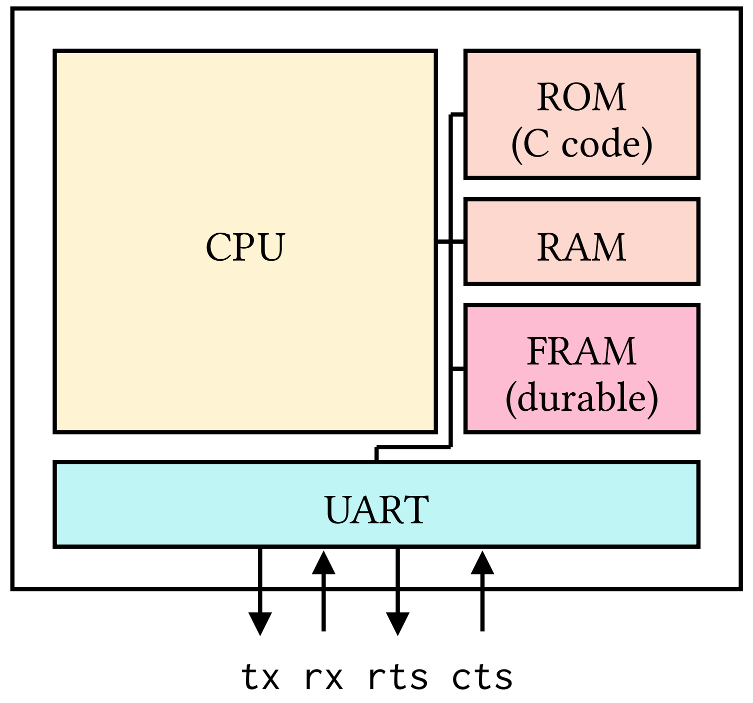
## Knox: verified HSM hardware/software

End-to-end

Monolithic verification of software + hardware

Limited scalability

anish.io/knox



~

```
# CA certificate signing HSM

var signing_key = null

def initialize(new_key):
  signing_key = new_key

def sign_certificate(cert):
  rsa_sign(signing_key, cert)
```

# Approach: K2 separation architecture

**K2 architecture**: isolate I/O, storage, and computation over secret state

Verify software correctness by leveraging prior work (HACL★)

Verify correctness down to hardware level using a new tool called **Concordance**

Verify cycle-level timing behavior using a new tool called **Chroniton**

# Approach: K2 separation architecture

**K2 architecture**: isolate I/O, storage, and computation over secret state

Verify software correctness by leveraging prior work (HACL★)

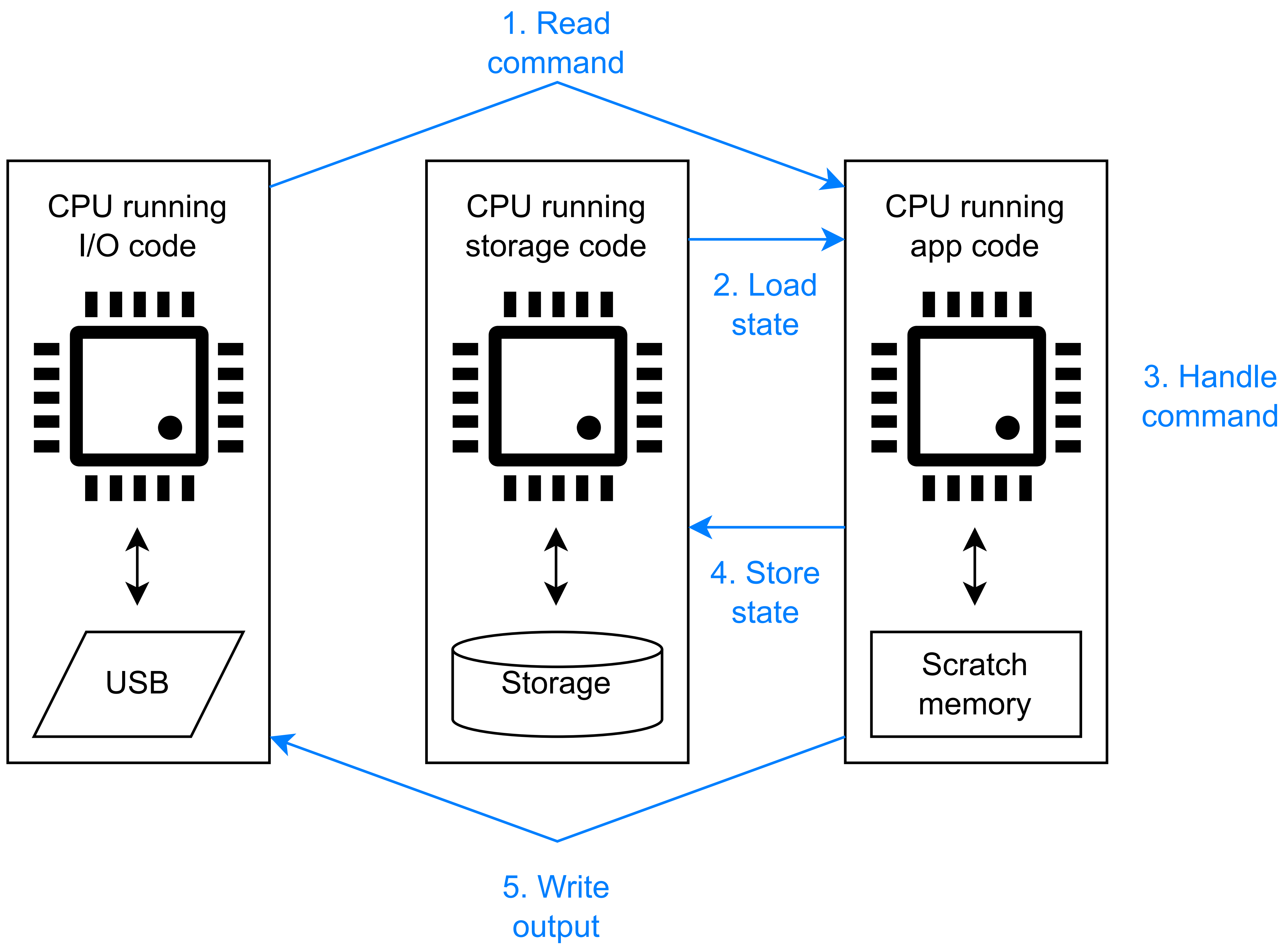
Verify correctness down to hardware level using a new tool called **Concordance**

Verify cycle-level timing behavior using a new tool called **Chroniton**

# K2 separation architecture: logical view

Separate I/O, storage, and computation over secret state: as if running on separate SoCs

Handling a single command: split into 5 phases



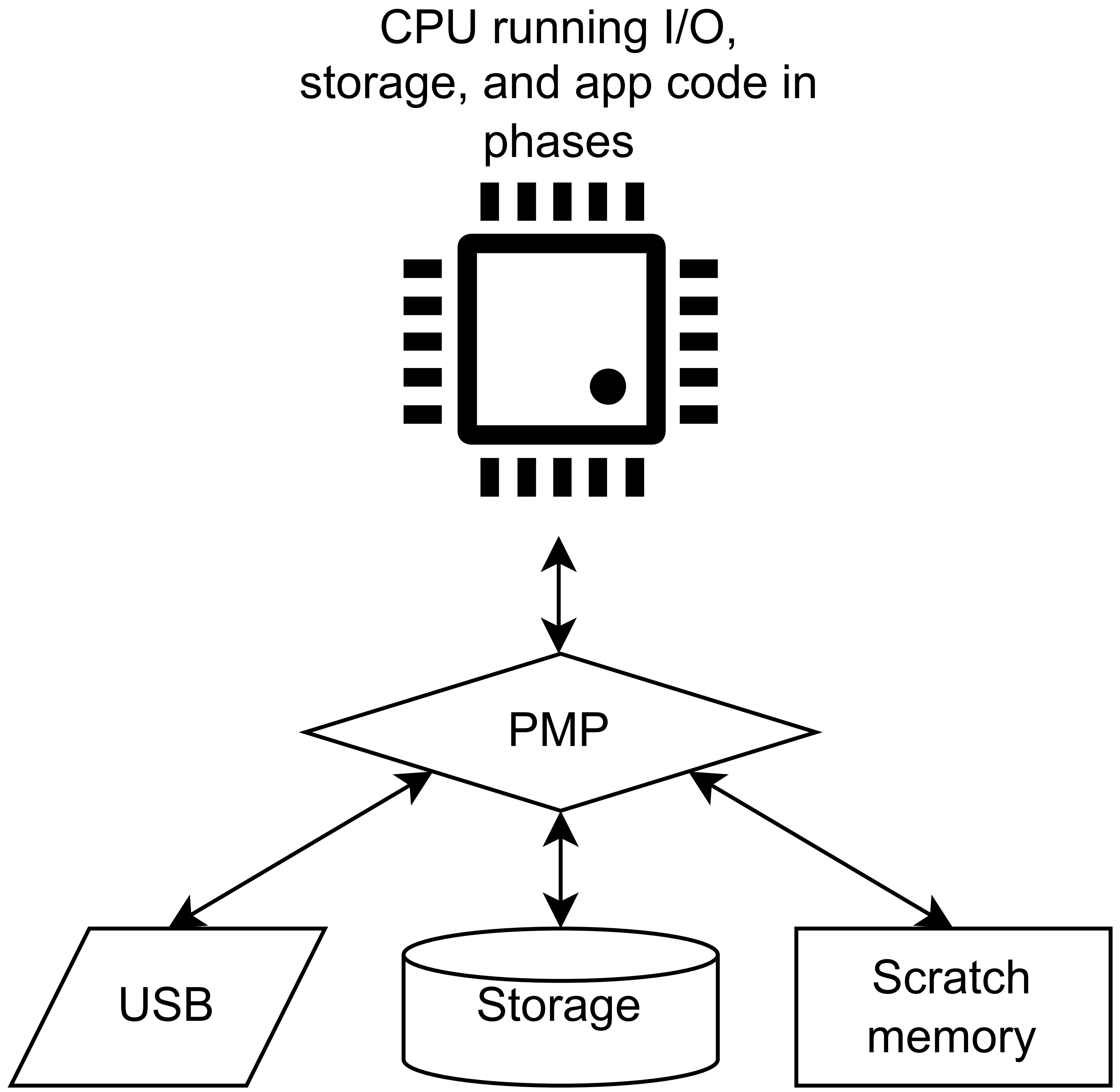
# K2 architecture: implementation

Single CPU

Tiny kernel runs phases in sequence

RISC-V PMP + state clearing for isolation

So e.g., bug in device driver can't leak secrets



# Architecture simplifies timing verification

Core application code runs start-to-finish with no interruption or intermediate observables

Reads state and command from RAM, writes new state and response to RAM: no I/O or persistence

Only timing leakage: end-to-end running time of `handle_command`

```
void handle_command(
    char *state,
    char *command,
    char *new_state,
    char *response)
{
    ...
}
```

# Verifying timing behavior at a cycle-accurate level

**Chroniton**: new tool to verify software timing behavior against hardware RTL

Proves that...

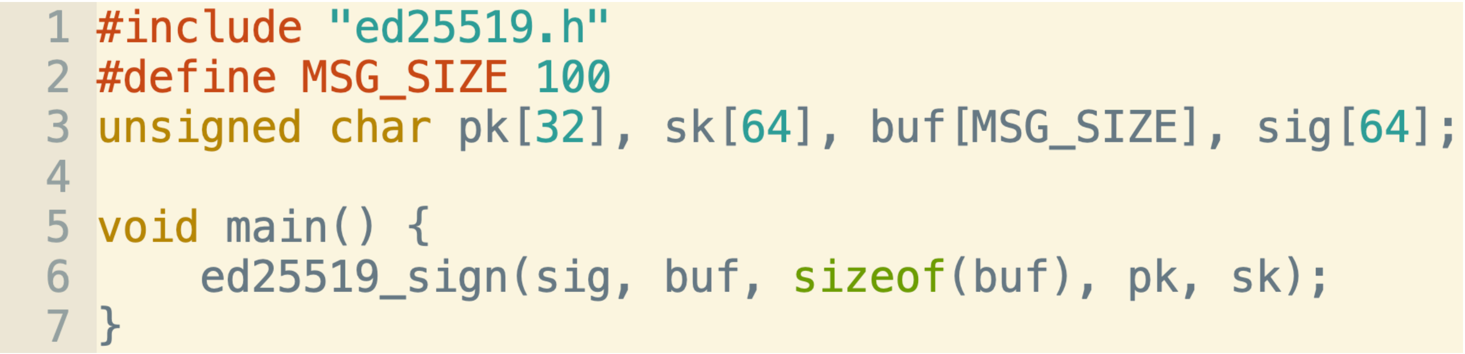
a particular hardware implementation (RTL-level)

runs a particular program (binary, memory image, e.g., `handle_command`)

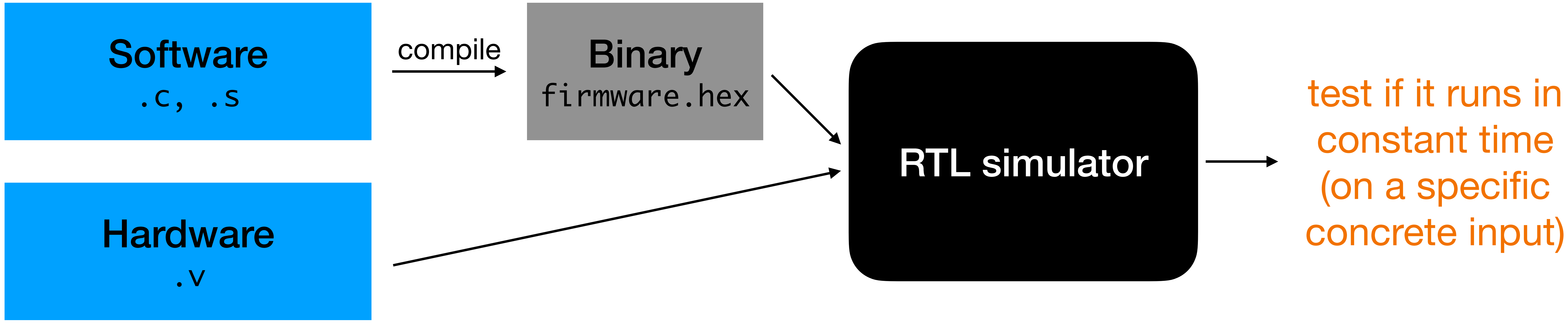
in constant time (cycles)

for all inputs

# An approximation: testing/fuzzing in RTL-level simulation

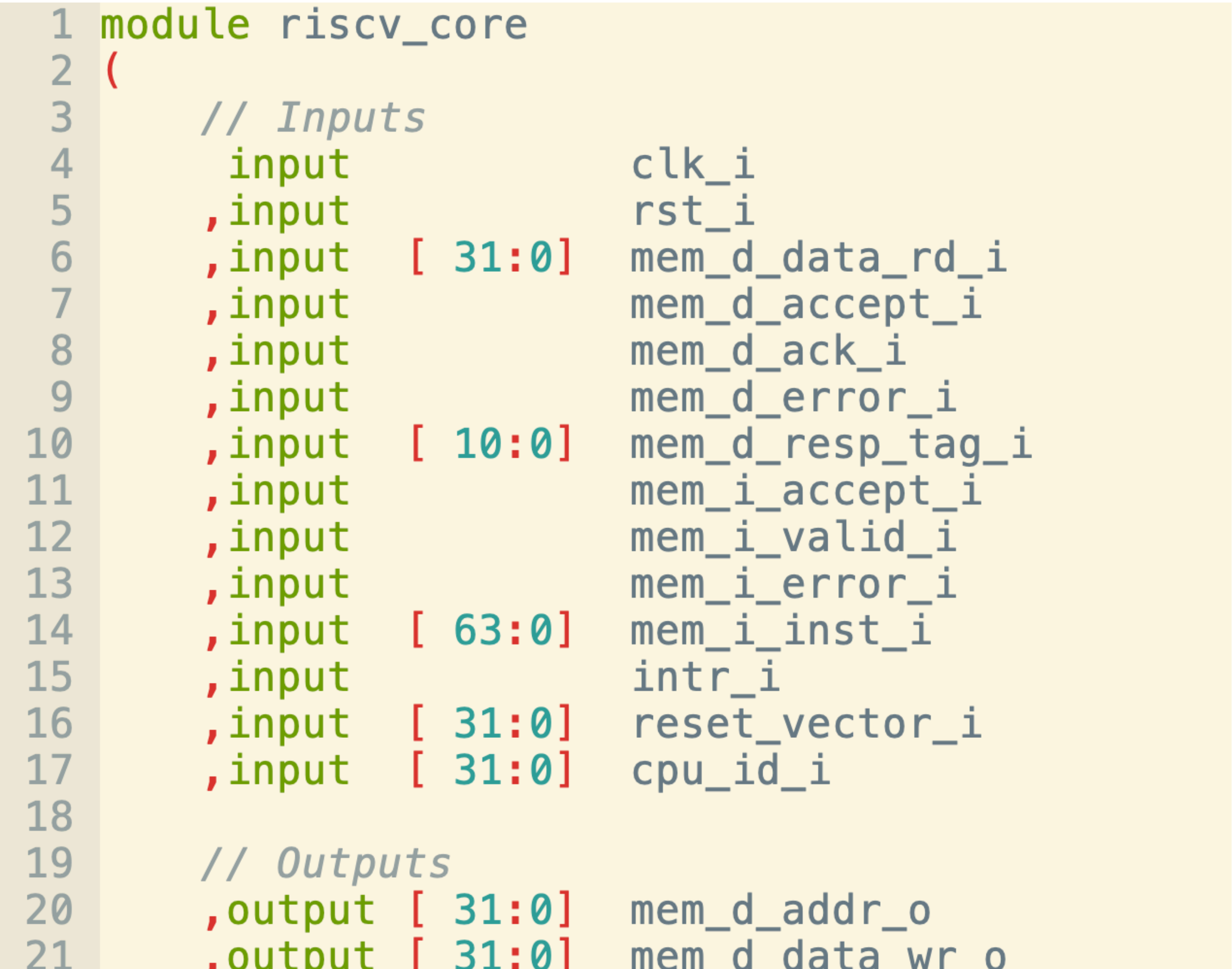


```c
#include "ed25519.h"
#define MSG_SIZE 100
unsigned char pk[32], sk[64], buf[MSG_SIZE], sig[64];

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



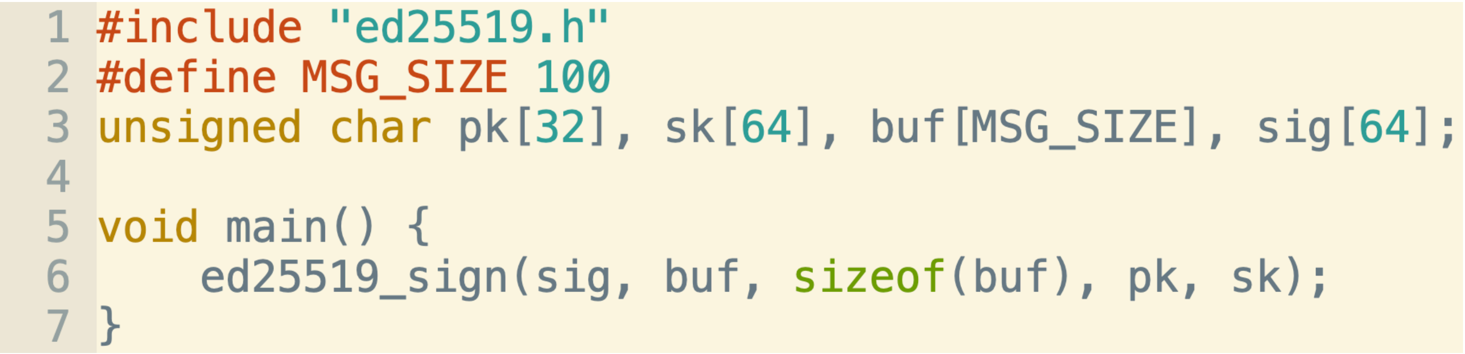
Software
.c, .s
compile
Binary
firmware.hex
Hardware
.v
RTL simulator
test if it runs in constant time (on a specific concrete input)

`$readmemh("firmware.hex", rom)`

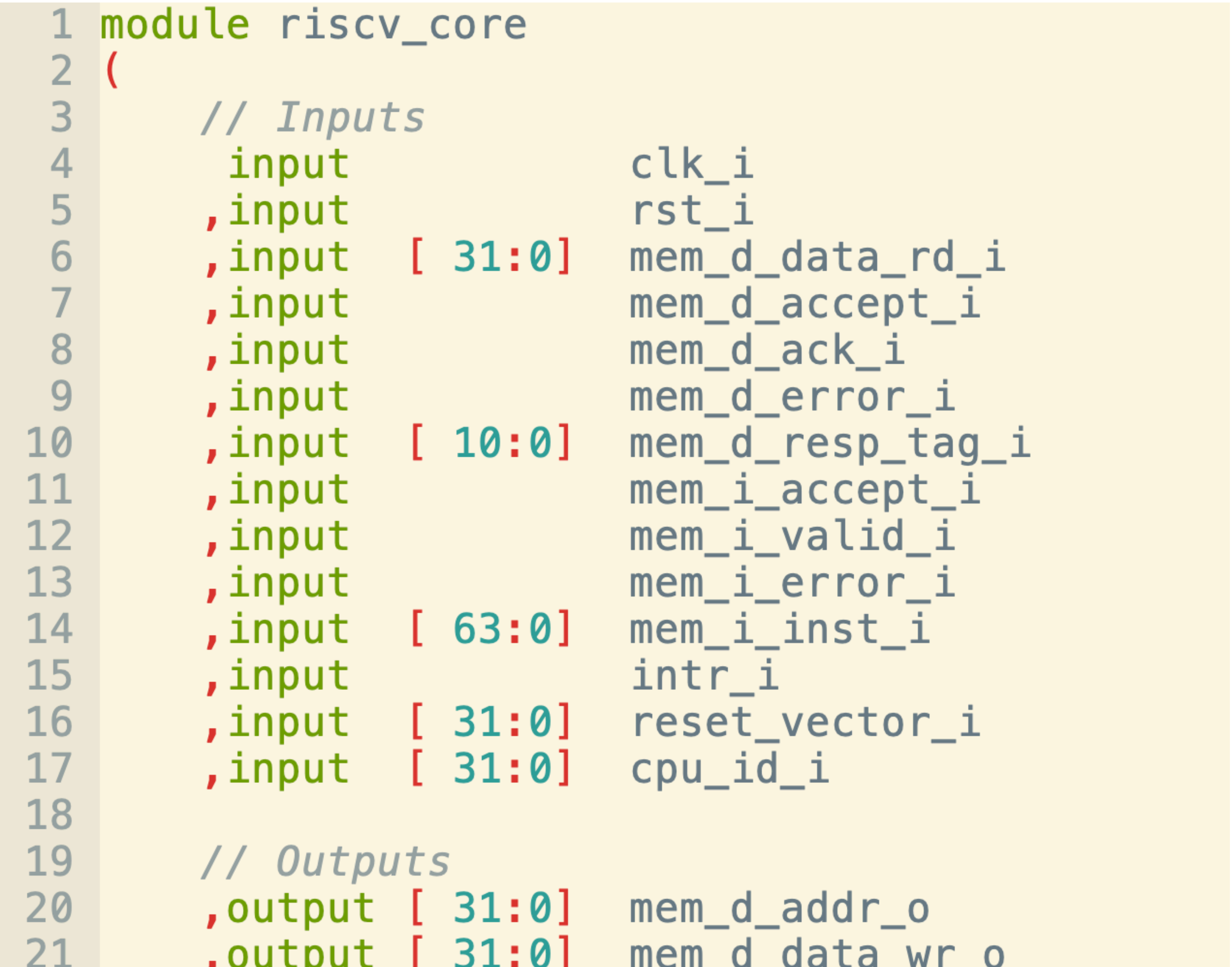


```verilog
module riscv_core
(
    // Inputs
     input              clk_i
    ,input              rst_i
    ,input  [ 31:0]     mem_d_data_rd_i
    ,input              mem_d_accept_i
    ,input              mem_d_ack_i
    ,input              mem_d_error_i
    ,input  [ 10:0]     mem_d_resp_tag_i
    ,input              mem_i_accept_i
    ,input              mem_i_valid_i
    ,input              mem_i_error_i
    ,input  [ 63:0]     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
```

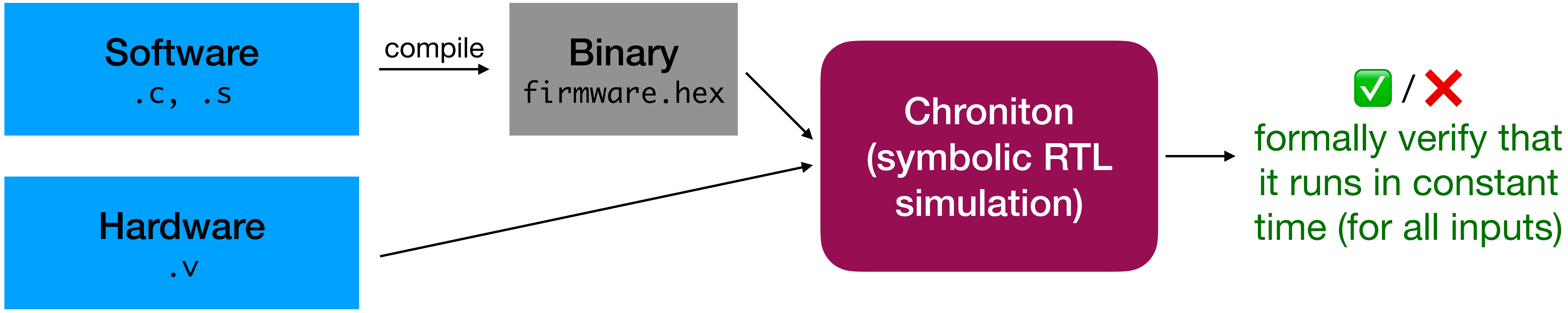
# Chroniton: verifying timing behavior using symbolic execution



```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);
}
```



```verilog
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  [ 10:0]  mem_d_resp_tag_i
    ,input           mem_i_accept_i
    ,input           mem_i_valid_i
    ,input           mem_i_error_i
    ,input  [ 63:0]  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
```



```
$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

# Verilog to Rosette compilation

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

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

endmodule
```

Verilog code

compile to state machine representation in Rosette

```
(struct state (...))

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

(define (step state)
  ...)

(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)
    ...
}
```

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

```
#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);
}
```

# 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

# Case studies: HSM following K2 architecture

CA certificate signing HSM (signature oracle)

Hardware: OpenTitan SoC

Software

- K2 kernel
- I/O code
- Storage code
- Application code, on top of HACL★ library

Implemented but not yet verified

# Related work

Hardware/software co-verification: Bedrock2 [PLDI'21], CakeML [PLDI'19]

Focused on correctness, not security

Application security verification: Ironclad Apps [OSDI'14]

Doesn't cover hardware or side channels

Verified cryptography: ct-verif [USEC'16], HACL [CCS'17], Fiat Crypto [S&P'19]

Doesn't cover hardware-level timing behavior

# Summary

K2 architecture: separate I/O, storage, and computation over secret state

Chroniton: verify timing at hardware level using whole-circuit symbolic execution

anish.io/k2

github.com/anishathalye/chroniton