The K2 Architecture for Trustworthy Hardware Security Modules

Anish Athalye¹, M. Frans Kaashoek¹, Nickolai Zeldovich¹, Joseph Tassarotti²
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

Hundreds of millions of deployed HSMs
HSMs suffer from bugs

Hardware

Software

Timing side channels

STM32F303xB/C

Description of device error

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+16Kbytes, the read operation will return the content at address A instead of the content of address B.

CVE-ID

CVE-2019-18671

Learn more at National Vulnerability Database (NVD)

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 firmware buffer overflow via crafted messages. The overflow in ethereum_extractT can circumvent stack protections and lead to code execution. The vulnerability over WebUSB.

Security Advisory 2015-04-14

CVE-2018-6875

Learn more at National Vulnerability Database (NVD)

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.
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

```python
# 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`.

```c
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);
}
```

![Diagram](image)

**Software**
- .c, .s

**Hardware**
- .v

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

**Compile**
- firmware.hex

**RTL simulator**
- test if it runs in constant time (on a specific concrete input)
Chroniton: verifying timing behavior using symbolic execution

Software
.C, .S

Hardware
.V

compile
Binary
firmware.hex

Chroniton (symbolic RTL simulation)

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

(struct state (...))
(define (new-symbolic-state)  
    ...)  
(define (step state)  
    ...)  
(define (with-input state input)  
    ...)  
(define (get-output state)  
    ...

Verilog code

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

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);
}
```
## 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
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