Towards verifying application isolation for cryptocurrency hardware wallets

Andrew Shen

Personal computers (PCs) are inadequate for security sensitive operations

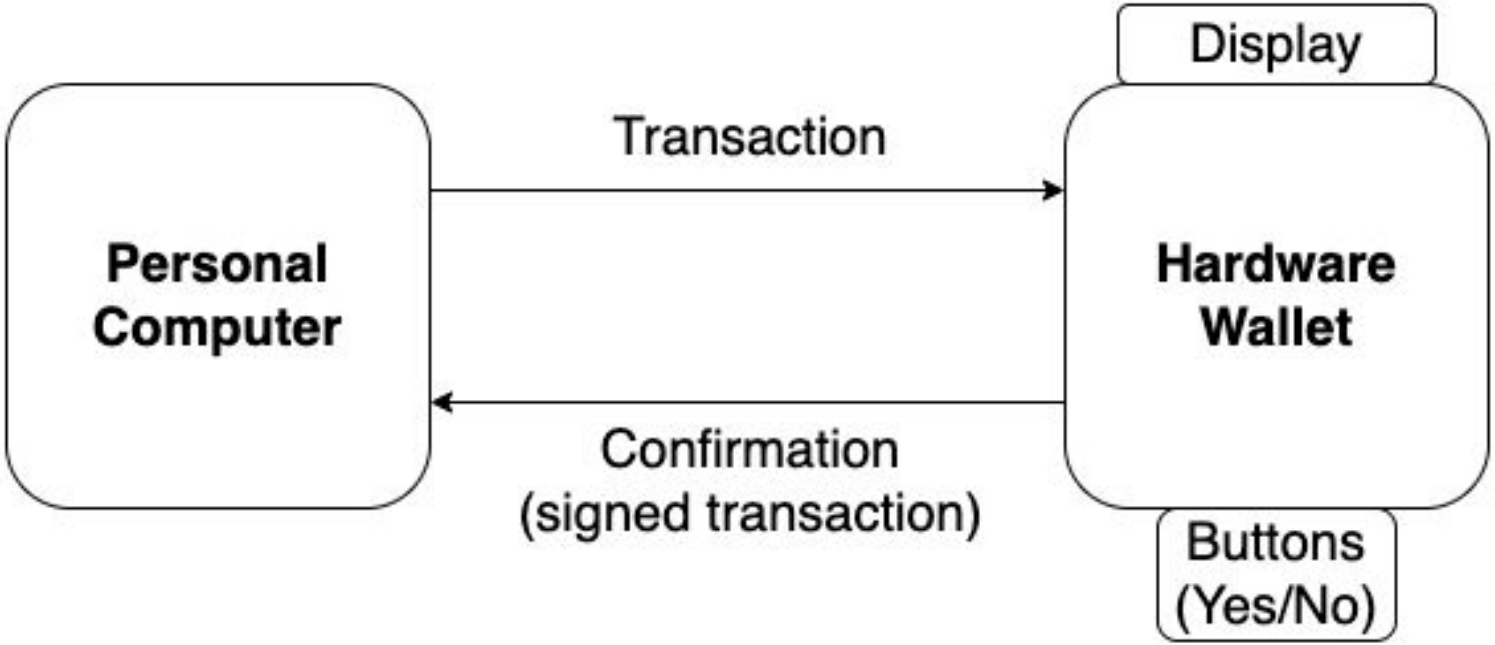
- Users use their PCs for many security sensitive operations such as cryptocurrency transactions and bank transactions.
- Security relies on PCs being secure.
- Modern PCs are full of security vulnerabilities.
  - Too complicated.
  - Lots of software, lots of room to go wrong.

Hardware wallets provide security even when PC is compromised

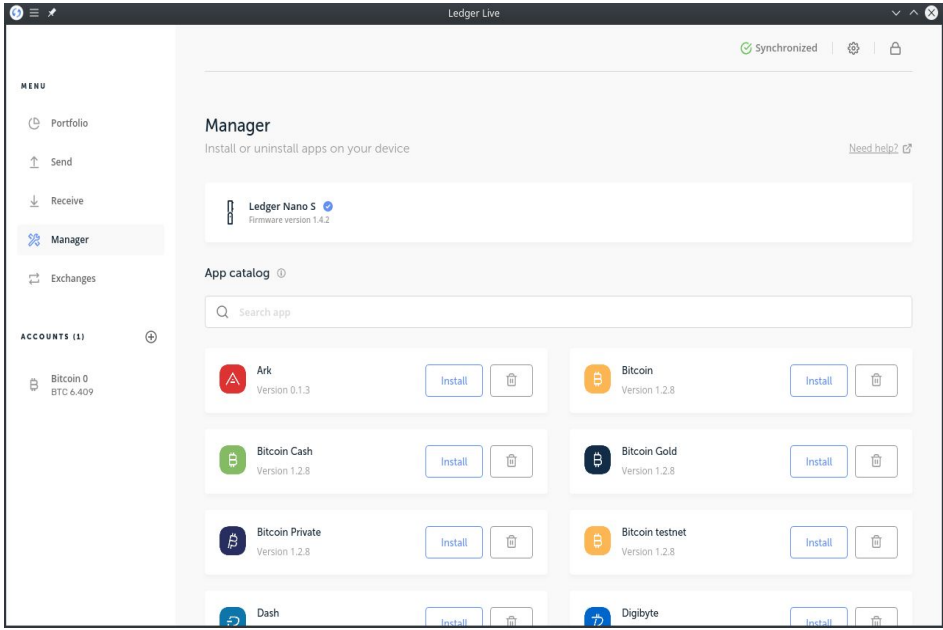
- Separate the confirmation and the transaction.
- The hardware wallet connects to the computer through USB and provides a display and buttons to verify the transaction.
- They can reduce the size of the Trusted Computing Base (TCB) of the personal computer.

Ledger: a common cryptocurrency hardware wallet

Hardware Wallet Diagram



Hardware wallet



Hardware wallets have isolation bugs

- Each wallet should be able to run numerous cryptocurrency applications (ex. Bitcoin, Ethereum, etc).
- The wallet operating system code base is still complex.
- Each of these applications should be isolated.
- Past wallets have had bugs and issues in security in past real-world wallets.
- **Can we do better?** Increase confidence that programs cannot interfere with or corrupt data in other programs or in our kernel?

How do we increase our confidence in our code?

- Add test cases, we can formulate examples to check the expected outcome against the actual outcome.
- Test cases can’t encompass all edge cases.

Wouldn’t it be nice if we could “test against all possible inputs?”

- We describe the expected outcome of the kernel and check that the kernel always matches our expectation, regardless of the input.
- This is known as verification.

Goal: Apply verification to prove security properties of a hardware wallet kernel.

Simple kernel design

- Kernel for an embedded device.

Our kernel has the following features:

- Small code base.
- Install applications.
- Loads and launches application.

A deeper look into verification

Implementation - our running code that is **untrusted.**

Specification - our description of how the code **should** behave. It is **trusted**.

- If the “implementation satisfies the specification”, this means that “for any input, our code correctly executes as the specification states.”
- In this project, the specification is we have set up the kernel in a such a way that code running in the user space cannot corrupt kernel memory.

Related work

- Hyperkernel (SOSP ‘17) and Serval (SOSP ‘19) outlined ways of using push-button verification to prove correct kernels’ system calls.
  - Does not reason about user mode (i.e. applications running on the kernel).
  - Does not reason about configuration of memory protection.
  - Does not reason about CPU privilege levels.

Structure of our proof

- Show that the kernel sets up the machine in a reasonable way and enters user mode (memory protection, CPU privilege levels).
- Show that from that starting state, execution satisfies invariants such that the kernel memory cannot be overwritten.

Proof

OK(s) - our predefined “reasonable” state.

Base Case:

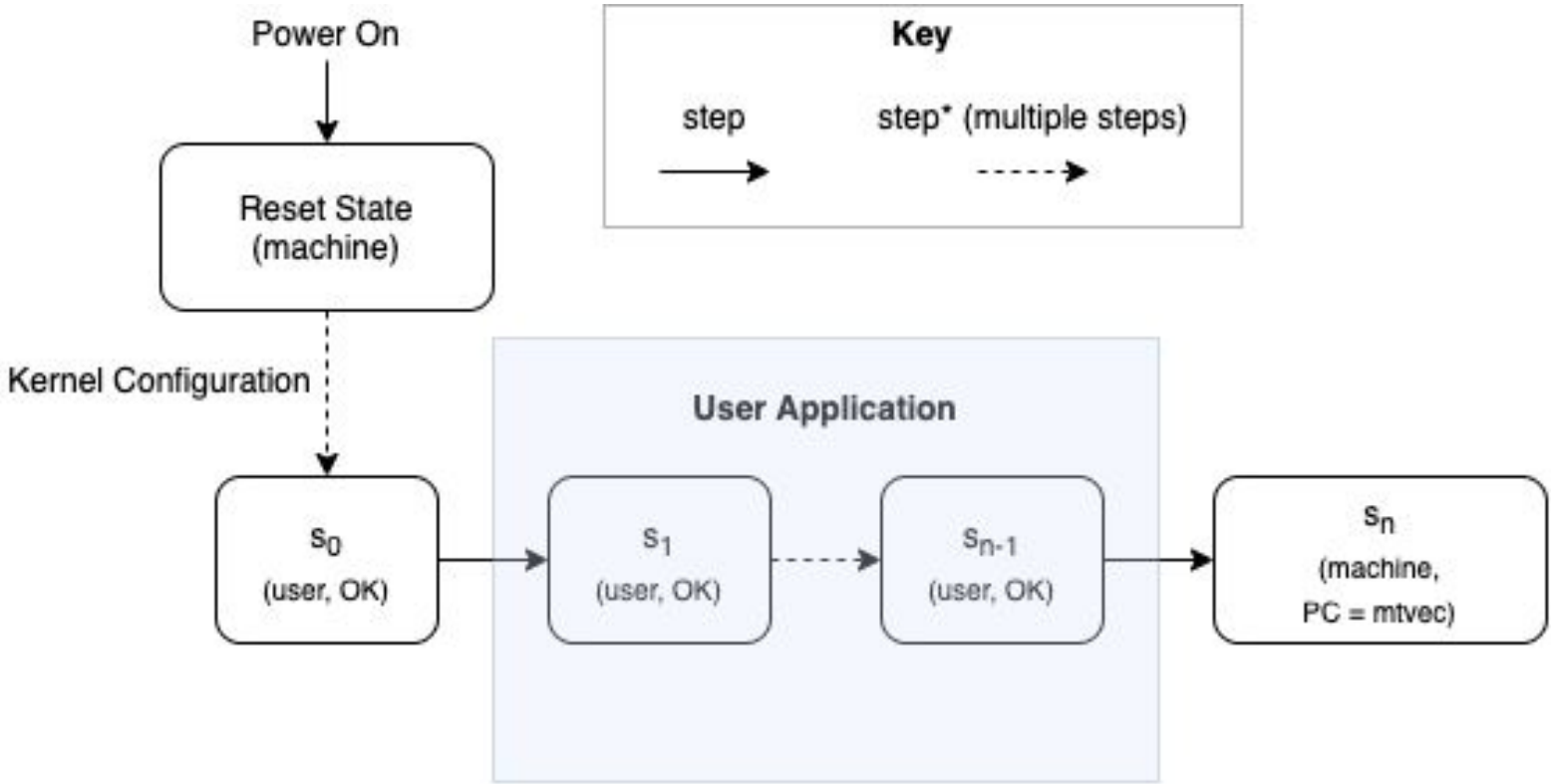
$OK(s_0)$, where s_0 is the state of the machine immediately after kernel finishes booting.

Induction Case:

$$\forall\, s, s' : OK(s) \Longrightarrow s' = step(s) \Longrightarrow is\_user(s') \wedge OK(s) \Longrightarrow$$

$$(is\_user(s') \wedge OK(s')) \vee (is\_m(s') \wedge pc(s') = mtvec(s))$$

Proof (cont.)


Power On
Reset State
(machine)
Kernel Configuration
Key
step
step* (multiple steps)
User Application
s_0
(user, OK)
s_1
(user, OK)
s_{n-1}
(user, OK)
s_n
(machine,
PC = mtvec)

Implementation of proof

- Build a symbolic RISC-V machine emulator using Rosette.
- Apply this to our sample kernel.
- Set up a machine to be in a generic OK state.
- Run a symbolic instruction to generate a resulting state.
- Compare kernel memory in both states to generate our proof formula.
- Solve formula.

Powerful tools: Rosette and Z3

Rosette

- Nice interface to “lift” or automatically port our implementation code so that it can work with symbolic values.
- Prove properties about the behaviour of our kernel.

Z3

- Z3 is an SMT solver that we used to prove our properties.

Example: Rosette

Code:

```
#lang rosette/safe

(define (add x y)
     (+ x y))

(define x 3)
(define y 5)
(printf "Concrete Result: ~a~n" (add x y))

(define-symbolic sym-x sym-y integer?)
(printf "Symbolic Result: ~a~n" (add sym-x sym-y))

(define model-add (verify
     (assert (not (equal? (add sym-x sym-y) 10)))))
(printf "Result of model-add: ~a~n" model-add)
```

Output:

```
Concrete Result: 8
Symbolic Result: (+ sym-x sym-y)
Result of model-add:
(model
 [sym-x 10]
 [sym-y 0])
```

Example: step

```
(define (step m)
    (define next_instr (get-next-instr m)) ; fetch actual instruction
    (define decoded_instr (decode m next_instr))
    (execute decoded_instr m))
```

Example: decode

```
(define (decode m b_instr)
     (define instr null)
     (define opcode (extract 6 0
b_instr))
     (define fmt (get-fmt m opcode))
     (cond
          [(eq? fmt 'R)
               (decode-R m b_instr)]
          [(eq? fmt 'I)
               (decode-I m b_instr)]
          [...]
          [else
               (illegal-instr m)]))

(provide decode)
```

```
(define (decode-R m b_instr)
     (define op null)
     (define rd (extract 11 7 b_instr))
     (define funct3 (extract 14 12 b_instr))
     [...]
     (define valid null)
     (cond
          [(and (bveq funct3 (bv #b000 3))
               (bveq funct7 (bv #b0000000 7)))
               (list 'add rd rs1 rs2)]
          [(and (bveq funct3 (bv #b000 3))
               (bveq funct7 (bv #b0100000 7)))
               (list 'sub rd rs1 rs2)]
          [...]
          [else
               (illegal-instr m)]))
```

Example: execute

```
(define (execute instr m)
     (define opcode (list-ref instr 0))
     (define pc (get-pc m))
     (cond
          [(eq? opcode 'lb)
               (define rd (list-ref-nat instr 1))
               (define v_rs1 (gprs-get-x m (list-ref-nat instr 2)))
               [...]
               (define val (sign-extend (machine-ram-read m adj_addr 1) (bitvector 64)))
               (gprs-set-x! m rd val)
               (set-pc! m (bvadd pc (bv 4 64)))
               instr]
          [(eq? opcode 'sb)
               (define v_rs1 (gprs-get-x m (list-ref-nat instr 1)))
               [...]
               (define success (machine-ram-write! m adj_addr v_rs2 8))
               instr])]))
```

Rosette recap

- Encapsulate the state of a CPU into a formula.
- Reason about the effect of running any instruction on the CPU.
- If before and after running any instruction the kernel memory is unchanged, then this indicates that no instruction in user mode can write on kernel memory.

Current results

- Built fully functional kernel with memory protection.
- Built a symbolic emulator.
- Proved that the kernel configures the machine such that user applications are properly isolated.
- Implemented multiple memory representations to improve verification time.
- https://github.com/AndrewTShen/riscv-symbolic-emulator

Future Experimentations

- Continue to expand our symbolic machine emulator.

Acknowledgements

- PRIMES
- Anish Athalye
- My family