Towards verifying application isolation for cryptocurrency hardware wallets

Andrew Shen

Mentored by Anish Athalye

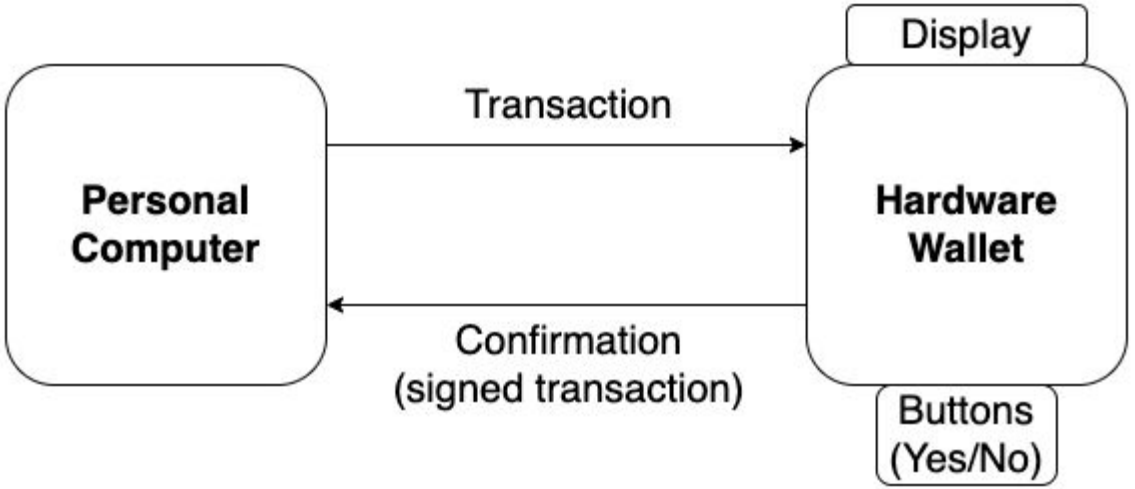
What do you do when your computer is not secure?

- Users perform cryptocurrency transactions on personal computer (PC).
- Security relies on PC being secure.
- Modern PC are full of security vulnerabilities.

Hardware wallets provide security 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) from a personal computer.

Ledger: A Common Cryptocurrency 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.
- This complexity has led to bugs and issues in security in past real-world wallets.
- Can we do better? Increase confidence that our programs cannot interfere or corrupt data in other programs or in the 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 our code and check that the code 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

Our kernel has the following features:

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

Implemented for a RISC-V processor.

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, it correctly executes as the specification states.

A simple verification example: sorting

Implementation (merge sort)

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Specification

```
forall list i j,
    i < j -> sort(list)[i] < sort(list)[j]
```

- If the implementation satisfies the specification, then we know that the implementation works for all possible values.
- This gives us confidence that our implementation function works, without having to trust that we wrote it correctly.

SAT and SMT Solvers: How do we do this proof?

How do we reason about every possible input?

- Use an SAT or SMT solver.
- SAT Solvers (SATisfiable) solve boolean satisfiability problems.
- These are identical to regular equations except the SAT solver tries to assign values to each variable to make the equation true.

Example 1: **a and not b.** If **a = True** and **b = False.** This equation is SAT.

Example 2: **a and not a**. This is equation is UNSAT.

- SMT solvers are more powerful e.g. can solve linear arithmetic.

Powerful Tools: Z3 and Rosette

Z3

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

Rosette

- Rosette is a library in the Racket language which provides us with a nice interface to “lift” or automatically port our implementation code to a format that can be understood by our SMT solver.

Rosette Example: sign function in ARM

Implementation (written ARM)

```
  cmp R0, #0
  ble 4
  mov R0, #1
  ret
4:
  cmp R0, #0
  bge 8
  mov R0, #-1
  ret
8:
  mov R0, #0
  ret
```

Specification (written in Racket)

```
(define sign (cond
    [(positive? a0) 1]
    [(negative? a0) -1]
    [else 0]))
```

Rosette Example: How do we lift our function?

Rosette Interpreter for ARM

```
(define (execute c opcode Rd Rn Op2
addr Rm Rs)
  (define pc (cpu-pc c))
  (case opcode
    [(ret)
     (set-cpu-pc! c 0)]
    [(mov)
     (set-cpu-pc! c (+ 1 pc))
     (set-cpu-reg! c Rd Op2)]
    # more interpreted instructions
))
```

- We wrote an interpreter for a small subset of ARM instructions.
- Emulates an ARM CPU.

Current Results (verification)

Goal Recap: Formulating and proving properties about our simple kernel.

Isolation Property: Loading and launching a program **a** does not depend on nor affect the contents of program **b**.

What does the implementation and specification look like?

- **Implementation**: Our prebuilt kernel.
- **Specification**: Regardless of the application, we would like our sensitive data for the kernel and each application to be unchanged.

Acknowledgements

Anish Athalye

PRIMES

My family