Towards verifying application isolation for cryptocurrency hardware wallets

Andrew Shen

Mentored by Anish Athalye

Hardware wallets provide useful properties

What are hardware wallets?

- Small devices that can make transactions e.g. using cryptocurrency or banks.
- Hardware wallets have real world usages and can be used to make securely transactions with cryptocurrency.
- They can reduce the size of the Trusted Code Base (TCB) from the PC.

Ledger: A Common Cryptocurrency Hardware Wallet

Isolation bugs in current hardware wallets

- Even for hardware wallets, the code base is still complex.
- Each wallet should be able to run numerous cryptocurrency programs.
- Each of these programs should be separated.
- 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?

- Reduce the size of the trusted code base.
- We have our implementation, so we can write a specification: what the program **should** do.
- Check against the specification, does our program do what we expect all the time?
- We now have simple model to check the expected output of our code.
- This is known as verification.

Goal: Apply verification to prove security properties

We would like to use the ideas in verification to show important properties about the way a kernel functions.

Simple Kernel Design

Our kernel should have the following features:

- Small code base
- Loads and launches programs from flash memory.
- Reset the entire kernel and run again.

A deeper look into verification

What is verification?

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

Specification - our representation of how the code **should** function. It is **trusted**.

- If the “implementation satisfies the specification”, this means that for any input to the function code, it correctly executes as the specification states.

A simple verification example

- Implementation: A sorting function that takes a list of 5 numbers as input, and output a list of the same 5 numbers, but in ascending order.
- Specification: Another function that checks if the output list is in ascending order.
- 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 reason about every single 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 just generalized versions of SAT solvers.

Powerful Tools: Z3 and Rosette

Z3

- Z3 is an SMT solver that we will use to prove our properties.
- It provides us with high performance and numerous features.

Rosette

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

Current Results (kernel)

- Implemented a bare-bones “kernel”
  - Built on both ARM and RISC-V processors.

Our kernel has the following features:

- Boot up processor
- Install applications
- Launch applications

Current Results (verification)

- Formulating and proving properties about our simple kernel.

Example: Loading and launching a program does not affect the sensitive contents of the kernel or other programs.

Acknowledgements

Anish Athalye

PRIMES

My family