SP2025 Week 02 • 2025-02-02

Seminar

Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing

Nikhil

Fuzzware

- Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing
- Authors: Tobias Scharnowski, Nils Bars, and Moritz Schloegel, Ruhr-Universität Bochum; Eric Gustafson, UC Santa Barbara; Marius Muench, Vrije Universiteit Amsterdam; Giovanni Vigna, UC Santa Barbara and VMware; Christopher Kruegel, UC Santa Barbara; Thorsten Holz and Ali Abbasi, Ruhr-Universität Bochum
- https://www.usenix.org/system/files/sec22-scharnowski.pdf

Table of Contents

- Fuzzing
- Problems with fuzzing embedded systems
- Fuzzware
- Discussion

Background: Fuzzing

- Coverage-guided random testing
- Fuzzing needs
  - Coverage instrumentation
  - Input mutator
  - Crash oracle
- Fuzzer tries to select inputs that will explore as much of the program as possible
- Intuition is that we are more likely to find bugs if we explore more of the program
- Fuzzing has been very successful at finding bugs and is now a standard technique
- Examples of fuzzers: AFL, libfuzzer

Background: Embedded Systems

- Small, specialized computer systems
- Smart home devices, industrial logic controllers, avionics systems, car engine controllers, medical devices, etc.
- Usually safety-critical

Background: Symbolic Execution

- “Concrete execution”: executing a program with concrete inputs
- Symbolic execution: makes the inputs symbolic values
- Track symbolic expressions program variables/state
- Look at all possible paths

Problem Setting

- We are given the firmware for an embedded system
- We know the CPU architecture
- We don't know enough about the peripherals to create a peripheral models for a full-system emulation
- We want to fuzz the firmware and find bugs

An Idea

- Emulating embedded systems is hard due to handling peripheral MMIO accesses
- We also don’t have a good notion of “input” to send fuzzer testcases to
- Use fuzzer input as MMIO access values!

Input Overhead

- All bits in an MMIO read value might not be important
- Amount of “information” in an MMIO value might be less than the full read size

```
void perform_op() {
 // Check requested operation
 switch (mmio->op) {
   case A: handle_A(); break;
   case B: handle_B(); break;
   case C:
     if(mmio->status == SPECIAL) {
       handle_C_special(); break;
     } else {
       handle_C_default(); break;
     }
   default: housekeeping();
 }
}
```

Figure 3: An example of a function that takes actions based on MMIO input using switch/case and if/else constructs.

Dealing with input overhead

- Use symbolic execution to figure out the “important” MMIO values (corresponding to code paths)
- Create a model of the peripheral access
- Use fuzzer input to select one of these “important” values
- Improved coverage

Fuzzware Design

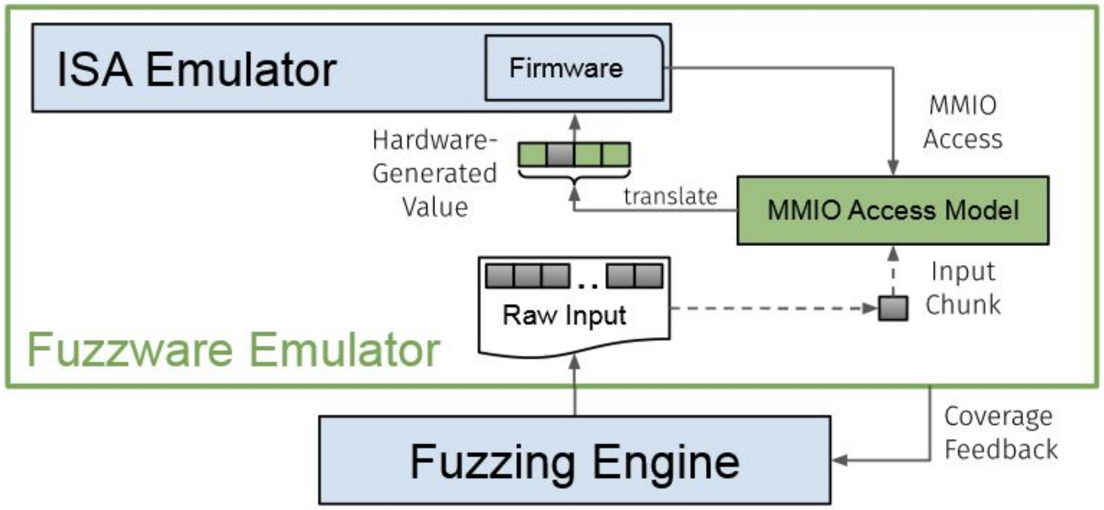

Figure 4: FUZZWARE's MMIO access handling design. The fuzzing engine generates a raw input file. Upon MMIO accesses, chunks of the input file are consumed by MMIO access models and translated into (potentially larger) hardware-generated values, which are then served to the emulated firmware. Once the raw input is exhausted, coverage feedback is provided to the fuzzing engine to guide the fuzzing process.

Discussion

- Is the “input overhead” idea applicable to fuzz software other than embedded firmware?
- If we had source code for the firmware we were fuzzing, could we do better than the approach in this paper?
- Can we use the ideas in this paper to do more than just fuzzing?
- Do you have any ideas to make this better?
- Do you see any problems with this approach?