Forming Faster Firmware Fuzzers

Lukas Seidel, Qwiet AI and TU Berlin
Dominik Maier, TU Berlin
Marius Muench, VU Amsterdam and University of Birmingham

The 32nd USENIX Security Symposium
Our Goal: Re-Think Firmware Emulation for Fuzzing
Firmware Fuzzing

CPU

Firmware

MMIO

External Peripherals e.g., WiFi Antenna
Firmware Fuzzing

Emulator

Firmware (lifted by emulator)

MMIO

External Peripherals
  e.g., WiFi Antenna
Firmware Fuzzing

Firmware (lifted by emulator)

MMIO

Emulator

Fuzzing Engine

External Peripherals e.g., WiFi Antenna
Observations

1) Full Binary lifting / rewriting (even if heavily cached) is expensive. QEMU’s advantage is executing diverse architectures but most embedded work focuses on ARM.

2) QEMU was developed for more complex systems, deploying a SoftMMU which dispatches all memory accesses and introduces significant overhead.

For more roadblocks that we addressed, please refer to our paper.
Near-Native Rehosting

Core Idea:

a) A lot of embedded firmware runs on ARMv7-M chips
b) Certain ARMv8-A cores provide compatibility with AArch32 and Thumb instruction set variants

⇒ Execute binaries for small embedded devices on their “bigger brothers”!

By this, we

- Heavily reduce the amount of code which needs lifting / rewriting
- Outperform rehosting approaches built on top of general-purpose emulators
Reduced Memory Access Overhead

- Mirror memory layout of the embedded device in userspace
  
  $\Rightarrow$ rewritten instructions do not need extra logic to dispatch memory accesses

- Use your usual MMU to detect memory violations

  $\Rightarrow$ no need for overhead-inducing SoftMMU
The Framework

SAFIREFUZZ

Userspace

LibAFL

HAL Layer Abstractions

Basic Blocks

Rehosted Binary

Dynamic Rewriting
& Instrumentation

Basic Blocks

Firmware Binary (Target)
High-Level Emulation

- Search for functions accessing MMIO peripherals (HAL)
- Emulate their behavior in a high-level language (handler)
- Insert hooks to your handler while rewriting

⇒ Eliminate problematic MMIO accesses
Basic Block Rewriting

Original Basic Block

```
0x10000:  movs  r0, #0
0x10002:  movs  r1, #0
0x10004:
    ldr  r3, [pc, #0x30]
0x10006:  cmp  r3, #1
0x10008:  beq  #0x20e
```

Rewritten Basic Block

```
movs  r0, #0
movs  r1, #0
movt  r3, #0x1
movw  r3, #0x34
ldr  r3, [r3]
cmp  r3, #1
```

Rewritten Basic Block after first Execution

```
mov  r0, #SUCC_0_ADDR
blx  rewrite_bb
mov  r0, #SUCC_1_ADDR
blx  rewrite_bb
blx  resolve_branch
pop  {r0-r12, lr}
nop
```
Evaluation

- 12 targets previously fuzzed by other firmware fuzzing work, e.g.,
  - STM32-based PLC firmware
  - HTTP Server for Atmel SAM R21 microcontrollers
  - Contiki OS-based WiFi Receiver/Transmitter
  - A fuzzing benchmark firmware with artificial vulnerabilities (What You Corrupt Is Not What You Crash)
Evaluation

- 12 targets previously fuzzed by other firmware fuzzing work, e.g.,
  - STM32-based PLC firmware
  - HTTP Server for Atmel SAM R21 microcontrollers
  - Contiki OS-based WiFi Receiver/Transmitter
  - A fuzzing benchmark firmware with artificial vulnerabilities (*What You Corrupt Is Not What You Crash*)

- 4 baseline configurations
  - HALucinator (state-of-the-art HLE-based)
  - HALucinator-LibAFL
  - FuzzWare (state-of-the-art symbolic execution-based)
  - FuzzWare-NoHAL
Basic Block Coverage
Performance

690x faster than HALucinator

145x faster than FuzzWare
New Targets

- 2 previously unfuzzed targets
  - Sine: open-source firmware for electric motor inverters
  - STMicroelectronics firmware example for image processing (libjpeg)

- 3 new Bugs
  - Sine:
    - Arbitrary write by corrupted config value (probably not exploitable)
  - Libjpeg:
    - Segfault after accessing uninitialized struct
    - Out-of-bounds write
Conclusion

⇒ Near-native execution, minimal rewriting
⇒ Rehosting of embedded firmware in Linux userspace
⇒ Vastly increased execution speeds
⇒ Less time to achieve (more) coverage