Coverage Guided Fuzzing of Remote Embedded Devices

David Lazar [1)] **,Chiharu Ota** [2)]

1) Argus Cyber Security (HQ)
Floor 36,. 94 Yigal Alon Street, Tel-Aviv, Israel (Email: david.lazar@argus-sec.com)
2) Argus Cyber Security (HQ)
Floor 36,. 94 Yigal Alon Street, Tel-Aviv, Israel (Email: chiharu.ota@argus-sec.com)

KEY WORDS:**fuzzing,coverage,coverage-guided,cyber security,security, embedded systems,** information system, system/communication system, Information, communication, and control (E2)

Fuzzing or fuzz testing is an automated software testing technique that involves random data as input to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. Fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software; it is responsible for the vast majority of remote code execution and privilege escalation bugs found to date in security-critical software. Unfortunately, fuzzing can be relatively shallow; blind, random mutations make it very unlikely to reach certain code paths in the tested code, leaving some vulnerabilities firmly outside the reach of this technique.

In order to make fuzzing more efficient, most used fuzzers nowadays follow a technique called coverage-guided fuzzing, which collects coverage data (i.e. the edges or paths between code blocks that the program visited during execution) and utilizes to mutate inputs based on this coverage data. The goal is to maximize the coverage of the tested program, which can lead to very interesting test cases that shallow fuzzers won't be able to reach.

In order to collect coverage, a small code needed to be added to each code basic block that will report to the fuzzer which basic blocks have been reached. This is called instrumentation.

When dealing with source code fuzzing, adding instrumentation is quite easy – the compiler adds the instrumentation in compilation time. However, when dealing with binary fuzzing, and specifically binaries that originated from embedded systems – one needs to be more creative to add this instrumentation to the existing binary. The most used way is to run the binary under the context of a CPU emulator (because most of the time the CPU architecture of embedded devices is different from the one of PCs). The CPU emulator can inject the instrumentation as it translates every instruction and can add its own instructions to support the instrumentation.

When running these binaries under the context of an emulator, there is a need to simulate all the hardware peripherals that exist in the real system environment but don't exist in the emulated environment.

This task is quite complicated and can take a long time, so setting up the fuzzing environment might not be worth it. Moreover, an appropriate emulator must be used for the CPU architecture of the embedded device, and if one doesn't exist, it is very time consuming and difficult to develop one.

We suggest a concept architecture of an automated way to collect code coverage of a live remote target embedded system as a part of a coverage-guided fuzzing process that aims to find software vulnerabilities in an authentic environment of the embedded system (i.e. while the software uses and communicates in real time with its peripheral hardware).

Using this method, vehicle manufacturers can test all of their software components embedded into their electronic control units (ECUs), when searching for software vulnerabilities, even if the source code for these is not available (for example, third party libraries).

Our solution solves the problems mentioned above when it comes to fuzzing embedded software by using a debugger on a live system to inject the instrumentation instead of using a standalone copy of the embedded software and running it under the context of an emulator with simulated peripherals. With this solution, it is possible to fuzz the embedded software in its authentic environment while all the hardware peripherals are up and running without the need to simulate them.

The solution is also CPU architecture agnostic that only demands a running debugger on the system (using JTAG for example), which is different from emulator based solution that needs tailored-made support for every CPU architecture.

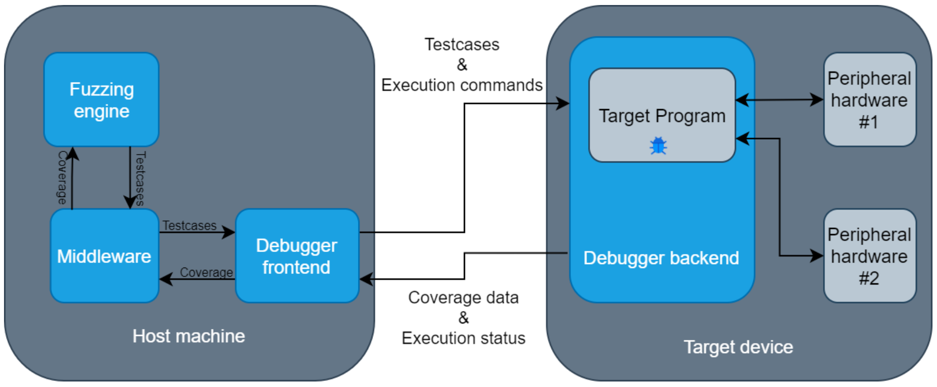

Fig. 1: Concept architecture