April 08, 2019

Fuzz testing of smartphones and IoT devices

Keun Soo Yim

Ji Won Shin

Fuzz testing of smartphones and IoT devices

ABSTRACT

Fuzz testing is an effective technique for finding software vulnerabilities. Fuzzing works by feeding quasi-random, auto-generated input sequences to a target program and searching for failures. When used to test physical devices, fuzzing is found to occasionally brick the devices, leading to significant testing expenses. Also, while existing kernel fuzzing is effective in finding kernel-interface vulnerabilities, it is not as efficient in finding deeply-hidden vulnerabilities.

This disclosure presents an architecture for continuously running fuzz tests at scale on physical devices, including on kernel and hardware abstraction layer (HAL) modules. Multiple fuzzers run parallel tests and collaborate in a decentralized manner. Fuzzers share control flow paths and corresponding code coverages as they are discovered. Fuzzers share syscall sequences that brick devices as they are discovered, and arrive at an efficient set of sequences that maximize test coverage.

KEYWORDS

Fuzzing; kernel fuzzing; hardware abstraction layer; HAL; device driver fuzzing; greybox testing; decentralized scheduling; distributive testing; collaborative testing; fuzz testing

BACKGROUND

Fuzz testing (or simply, fuzzing) is a simple and practical technique for finding software vulnerabilities. Fuzzing works by feeding quasi-random, auto-generated inputs into a target program searching for failures. With the proliferation of consumer devices running operating systems or kernels thereof, e.g., smartphones, TVs, smartwatches and other wearable devices, automotive electronics, IoT devices, etc., fuzzing has emerged as an efficient testing procedure that scales to large production volumes.

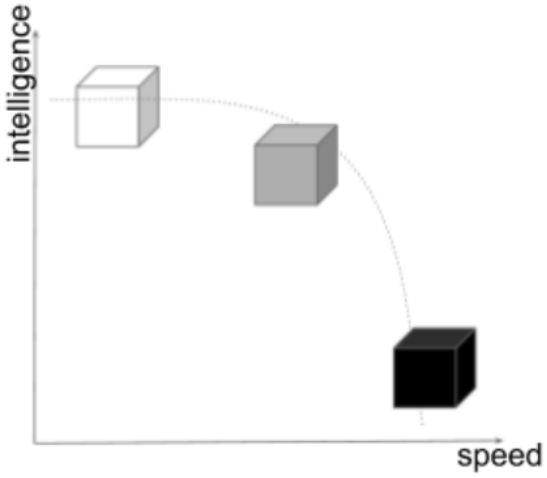

Fig. 1: The degree of randomness in fuzzing

As shown in Fig. 1, the degree of randomness in fuzzing can be between whitebox testing (systematic data and program analysis) and blackbox testing (searching for vulnerabilities in a manner oblivious to program/data structure). Greybox fuzzing, which stands between blackbox testing and whitebox testing, combines the speed of blackbox with the guidance of whitebox.

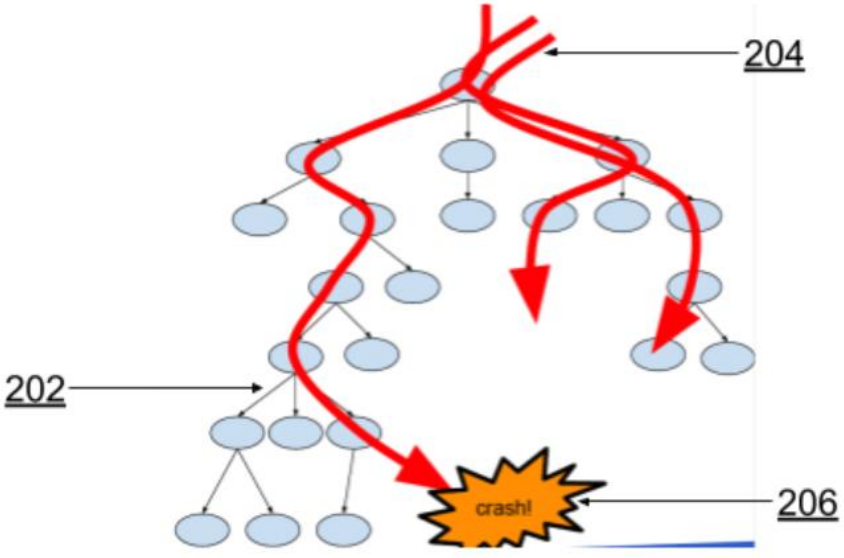

Fig. 2: Path selection in fuzzing

Fig. 2 illustrates path selection in a typical fuzzing run. One or more fuzzers select random paths (204) through a test graph (202). One path results in the finding of a vulnerability (206).

When used to test physical devices, fuzzing is found to occasionally brick the devices, leading to significant testing expenses. Use of multiple fuzzers can sometimes brick several devices over the same failure. Also, while kernel fuzzing is effective in finding kernel-interface vulnerabilities, it is not as efficient in finding deeply-hidden vulnerabilities. This is because a sequence of randomly generated syscalls does not necessarily replicate real world user behavior, e.g., taking a camera image, interacting with multiple device drivers, etc.

DESCRIPTION

This disclosure describes architectures for continuously running greybox fuzz tests at scale on physical devices, including kernel and hardware abstraction layers (HAL), e.g., user-space device drivers.

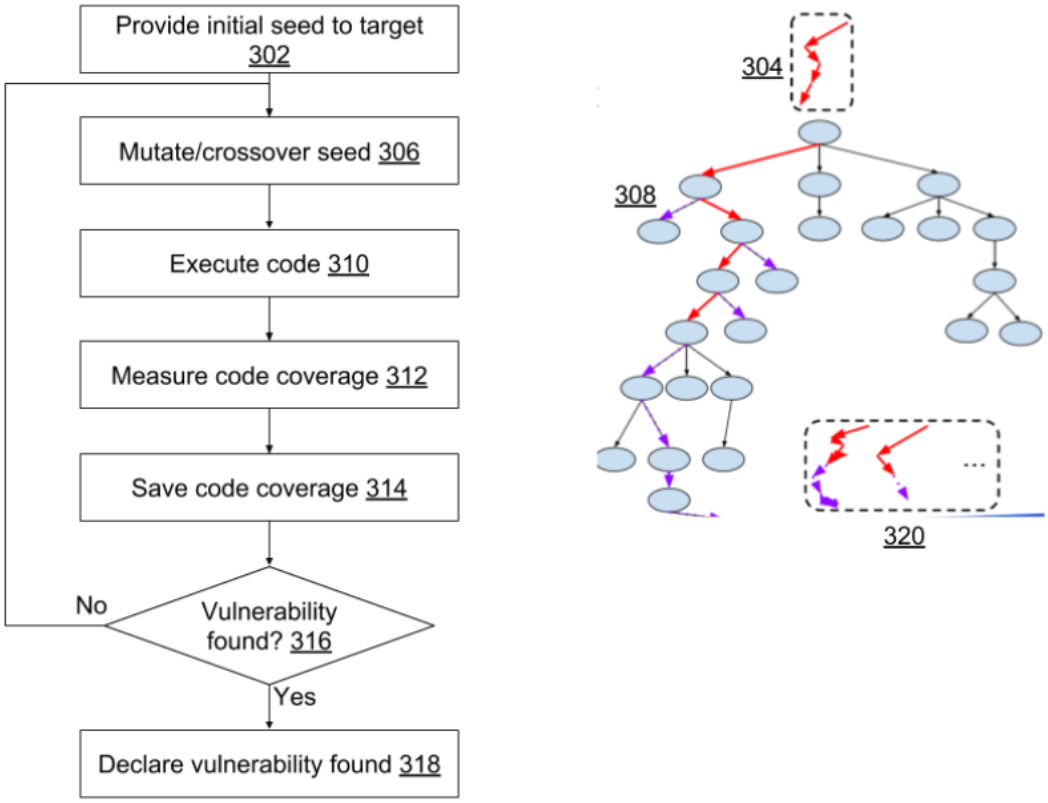

Fig. 3: Greybox fuzzing guided by code coverage

Fig. 3 illustrates greybox fuzzing guided by code coverage, per techniques of this disclosure. Fig. 3(a) illustrates the test flow in the style of a chart, while Fig. 3(b) illustrates the test flow in the style of a graph. The fuzzer provides a random initial seed, e.g., input sequence, to the target program (302). Graphically, an initial seed is represented by a sequence of red arrows (304). The fuzzer mutates or crosses over the seed (306) in a manner similar to genetic algorithms. The mutation is represented by purple branches (308) in the test graph, which deviate away from the initial seed.

The code is executed (310). Program instrumentation measures code coverage (312). The corpus, e.g., an input data set that leads to a specific control-flow path discovered by the fuzzing run, is saved (314), if found interesting. If no error or vulnerability, e.g., crash, security leak,

memory leak, etc., is found (316), then the fuzzer loops back to 302, and tests the target with a new mutation. If a vulnerability is found then it is declared (318). Examples of accumulated corpuses, e.g., sets of input data with specific control-flow paths, are shown (320). With the passage of time, the mutations result in good input sequences, e.g., sequences that provide good coverage.

Kernel fuzzing

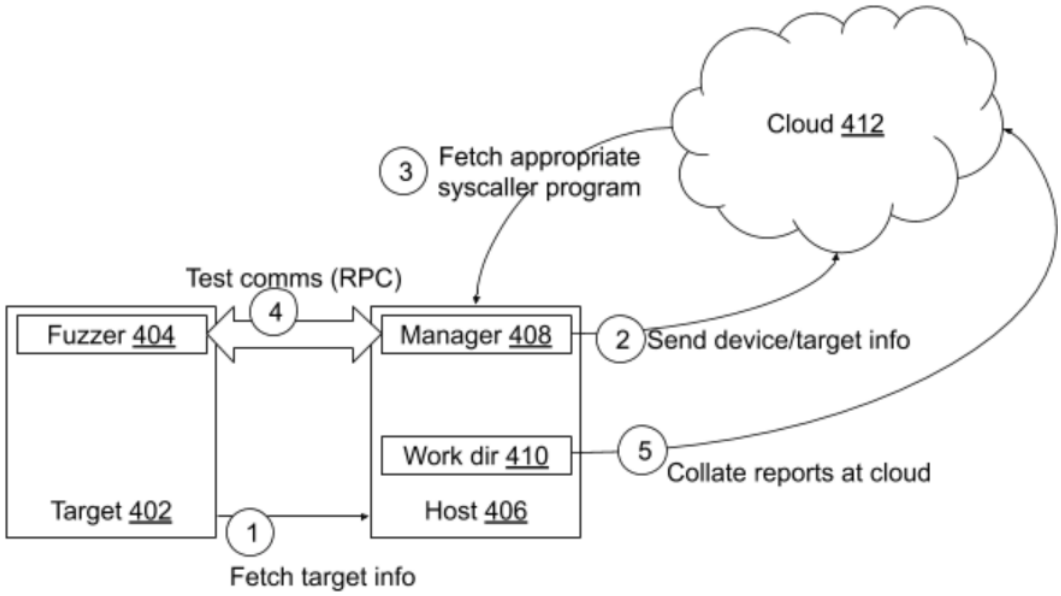

Fig. 4: Architecture for kernel fuzzing

Fig. 4 illustrates an architecture for kernel fuzzing, per the techniques of this disclosure. A kernel-under-test resides on a target (402), e.g., a physical device, on which also resides a fuzzer (404). Testing is managed by a manager (408) that resides on a host (406). The host also includes other components, e.g., a working directory (410). The cloud (412) serves as a repository for system caller programs, reports, etc. Fuzzing is executed as follows.

1. The host fetches target information from the target.
2. The host sends device/target information to the cloud.

3. The host fetches from the cloud a system caller program appropriate to the target.
4. The manager (on the host) and the fuzzer (on the target) communicate, e.g., via RPC, to establish and run the test.
5. Results are sent from the working directory to be collated at the cloud.

The physical-device based fuzzing architecture described herein enables discovery of bugs that are not easily discoverable under virtual environments. However, as mentioned before, the use of physical devices for fuzzing can occasionally result in the device being bricked. When multiple parallel fuzzers are used, such testing can sometimes brick several devices over the same failure.

Per the techniques of this disclosure, the multiple parallel fuzzers register syscall sequences that resulted in bricked devices such that those sequences are avoided in future runs. In this manner, coordination between the fuzzers minimizes damage to devices under test. The fuzzers also coordinate to optimize coverage. Coordination between fuzzers is distributed, e.g., there is no central or cloud-based coordinating agency. Distributed fuzzer coordination, as described herein, results in a simpler architecture and a robust test environment.

When fuzzing kernels, it is worthwhile to note that a kernel may not behave in a fully deterministic manner. For example, a sequence of syscalls may sometimes result in a bricked device, and at other times it may not. Therefore, kernel fuzzing, per the techniques herein, is carried out by generating several testing threads, assigning a set of syscalls to each, and running for a long enough time to eliminate the non-determinism.

User-space device driver (HAL) fuzzing

User-space device driver fuzzing is performed by automatically generating fuzzer logic from the specification of a component using compiler-based techniques. The fuzzer generates

structured random input comprising a specific sequence of high level function calls within a practical control flow. The function calls are independent of the processor architecture and device driver implementations. The fuzzing tool automatically generates fuzz drivers for the given interface definition language specifications of a HAL module.

The fuzz drivers target given devices and HAL modules by using remote procedure calls (RPC) and coverage-guided greybox fuzzing techniques. The fuzzing architecture collects the corpus, e.g., an input data set that leads to specific control-flow paths discovered by previous fuzzing runs, and similar to genetic algorithms, intelligently selects input seeds, so as to increase the chance of finding critical vulnerabilities. The fuzzing architecture is decentralized, e.g., the fuzzers coordinate without a central agency to sift through the corpus to discover relevant new seeds. Additionally, various heuristics are used for input seed selection.

Distributive collaboration between the fuzzers increases testing efficiency, since the fuzzers are able to optimize coverage, e.g., by avoiding overlap and by sharing knowledge of the corpus.

CONCLUSION

This disclosure presents an architecture for continuously running fuzz tests at scale on physical devices, including on kernel and hardware abstraction layers. Multiple fuzzers run parallel tests and collaborate in a decentralized manner. Fuzzers share control flow paths and corresponding code coverages as they are discovered. Fuzzers share syscall sequences that brick devices as they are discovered, and arrive at an efficient set of sequences that maximize test coverage. The physical-device based fuzzing architecture described herein enables discovery of bugs that are not easily discoverable under virtual environments.

REFERENCES

[1] "HIDL HAL fuzzing," http://newsvideo.su/tech/video/194727 accessed Nov 7, 2018.

[2] Jake Corina and Christopher Salls, "Fuzzing kernel drives with interface awareness" https://www.blackhat.com/docs/eu-17/materials/eu-17-Corina-Difuzzing-Android-Kernel-Drivers.pdf accessed Nov 7, 2018.