Reading Notes of Recent Papers

TSIS Lab. (Trusted Software and Intelligent System Lab.)

2020/01/21

Contents

1 SAVIOR: Towards Bug-Driven Hybrid Testing @S&P'20

1.1 Background/Problems

Hybrid testing combines fuzzing and concolic execution. It leverages fuzzing to test easy-to-reach code regions and uses concolic execution to explore code blocks guarded by complex branch conditions. As a result, hybrid testing is able to reach deeper into program state space than fuzz testing or concolic execution alone. Recently, hybrid testing has seen significant advancement. However, **its code coverage-centric design is inefficient in vulnerability detection**. First, it **blindly** selects seeds for concolic execution and aims to explore new code continuously. However, as statistics show, a large portion of the explored code is often bug-free. Therefore, giving equal attention to every part of the code during hybrid testing is a non-optimal strategy. It slows down the detection of real vulnerabilities by over 43%. Second, classic hybrid testing quickly moves on after reaching a chunk of code, rather than examining the hidden defects inside. It may frequently **miss subtle vulnerabilities** despite that it has already explored the vulnerable code paths.

1.2 Methods/Techniques

The authorspropose SAVIOR (Fig.1), a new hybrid testing framework pioneering a bug-driven principle.

Bug-driven prioritization: Instead of running all seeds without distinction in concolic execution, SAVIOR prioritizes those that have higher possibilities of leading to vulnerabilities. Specifically, before the testing, SAVIOR analyzes the source code and **statically labels the potentially vulnerable locations** in the target program. Moreover, SAVIOR computes the set of basic blocks reachable from each branch. During dynamic testing, SAVIOR **prioritizes the concolic execution seeds that can visit more important branches** (i.e., branches whose reachable code has more vulnerability labels).

Bug-guided verification: Aside from accelerating vulnerability detection, SAVIOR also verifies the labeled vulnerabilities along the program path traversed by the concolic executor. Specifically, SAVIOR synthesizes the faulty constraint of triggering each vulnerability on the execution path. If such constraint under the current path condition is satisfiable, SAVIOR solves the constraint to construct a test input as the proof. Otherwise, SAVIOR proves that the vulnerability is infeasible on this path, regardless of the input.

1.3 Results/Evaluation

Evaluation shows that the bug-driven approach outperforms mainstream hybrid testing systems driven by code coverage. On average, SAVIOR detects vulnerabilities 43.4% faster than DRILLER and 44.3% faster than QSYM, leading to the discovery of 88 and 76 more unique bugs, respectively. According to the **evaluation on 11 well fuzzed benchmark programs, within the first 24 hours, SAVIOR triggers 481 UBSAN violations, among which 243 are real bugs.**

1.4 Limitations/My Comments

- Over-approximation in Vulnerability Labeling: SAVIOR leverages sound algorithms to label vulnerabilities where the over-approximation may introduce many false-positive labels. This imprecision can consequently weaken the performance of SAVIOR's prioritization. One solution is to include more precise static analysis for finer-grained label pruning.
- Prediction in Vulnerability Detection: Once reaching a potentially vulnerable location in concolic execution, SAVIOR extracts the guarding predicates of the vulnerability label. However, these predicates may contradict the current path condition. In case of such contradiction, SAVIOR terminates the exploration of the labeling site immediately. Moreover, we can predict whether an execution path can trigger a vulnerability or not by studying the runtime information of previous executions, more importantly, before that execution arrives the vulnerability site. To achieve this goal, we need to backwardly summarize path constraints from the labeled site to its predecessors in the explored paths, by using the weakest precondition.
- Hybrid testing in SAVIOR is same with hybrid fuzzing in Driller and Berry. Both tools run fuzzing for code exploration and invoke concolic execution only on hard-to-solve branches, which takes advantage of both fuzzer's efficiency and concolic executor's constraint solving.

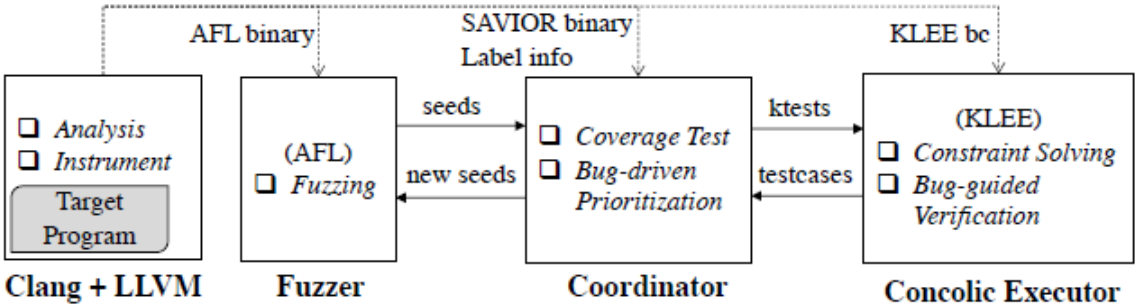

Figure 1: SAVIOR's arch.

2 Fuzzing File Systems via Two-Dimensional Input Space Exploration @S&P'19

2.1 Background/Problems

File systems are too big and too complex to be bug free. Nevertheless, to find bugs in file systems, regular stress-testing tools and formal checkers are limited due to the ever-increasing complexity of both file systems and OSes. Thus, fuzzing becomes a preferable choice, as it does not need much knowledge about a target. However, three prominent issues of existing file systems fuzzers exist: (1) fuzzing a large blob image is inefficient; (2) fuzzers do not exploit the dependence between a file system image and file operations; (3) fuzzers use aging OSes and file systems, which results in irreproducible bugs.

2.2 Methods/Techniques

The authors present JANUS (Fig.3, 120K C++/C LoC), the first feedback-driven fuzzer that explores the two-dimensional input space of a file system, i.e., mutating metadata on a large image, while emitting image-directed file operations. In addition, JANUS relies on a library OS rather than on traditional VMs for fuzzing, which enables JANUS to load a fresh copy of the OS, thereby leading to better reproducibility of bugs.

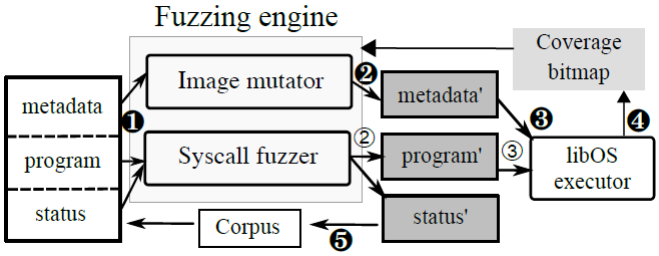

Figure 2: JANUS' arch.

2.3 Results/Evaluation

The authors evaluate JANUS on 8 file systems and found 90 bugs in the upstream Linux kernel, 62 of which have been acknowledged. 43 bugs have been fixed with 32 CVEs assigned. In addition, JANUS achieves higher code coverage on all the file systems after fuzzing 12 hours, when compared with the state-of-the-art fuzzer Syzkaller. JANUS visits 4.19X and 2.01X more code paths in Btrfs and ext4, respectively. Moreover, JANUS is able to reproduce 88–100% of the crashes, while Syzkaller fails on all of them.

2.4 Limitations/My Comments

- JANUS cannot fuzz the DAX mode of a file system without modification on LKL.
- To achieve a minimal PoC, JANUS uses a brute force approach to revert every mutated byte and also tries to remove every invoked file operation to check whether the kernel still crashes at the expected location, which is sub-optimal.
- JANUS does not support file systems (e.g., NTFS, GVfs, SSHF, etc.) that rely on FUSE (Filesystem in Userspace).
- By combining Janus and kAFL, we can fuzz file systems on other OSes.
- Other crash-consistency checkers [6,73] and semantic correctness checkers [36, 58] can rely on or integrate with Janus which aims to find general security bugs in file systems.
- To find security bugs in OSes, a number of general kernel fuzzing frameworks [20, 43, 46, 61] and OS-specific kernel fuzzers [22, 25, 44, 45, 47] have been proposed. Unlike JANUS, all these fuzzers generate random system calls based upon predefined grammar rules, which is ineffective in the context of file system fuzzing. Several recent OS fuzzers such as IMF [22] and MoonShine [49] focusing on seed distillation are orthogonal to this work. Nevertheless, JANUS can start with seed programs of high quality by utilizing their approaches.

3 REDQUEEN: Fuzzing with Input-to-State Correspondence @NDSS'19

3.1 Background/Problems

Two common problems of fuzzing are magic numbers and (nested) checksums (see Listing 1). Computationally expensive methods such as taint tracking and symbolic execution are typically used to overcome such roadblocks. Unfortunately, such methods often require access to source code, a rather precise description of the environment (e.g., behavior of library calls or the underlying OS), or the exact semantics of the platform's instruction set, and thus such methods are the polar opposite of the approach pioneered by AFL: to a large extend, AFL's success is based on the fact that it makes few assumptions about the program's behavior.

Listing 1: Roadblocks for feedback-driven fuzzing.

```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
      bug(2);
```

Bonus: Fuzzers from the AFL family have three important components: (i) the queue, (ii) the bitmap, and (iii) the mutators. The queue is where all inputs are stored. During the fuzzing process, an input is picked from the queue, fuzzed for a while, and, eventually, returned to the queue. After picking one input, the mutators perform a series of mutations. After each step, the mutated input is executed. The target is instrumented such that the coverage produced by the input is written into a bitmap. If the input triggered new coverage (and, therefore, a new bit is set in the bitmap), the input is appended to the queue. Otherwise, the mutated input is discarded. The mutators are organized in different stages. The first stages are called the *deterministic stages*. These stages are applied once, no matter how often the input is picked from the queue. They consist of a variety of simple mutations such as "try flipping each bit". When the deterministic stages are finished or an input is picked for the second time, the so called *havoc phase* is executed. During this phase, multiple random mutations are applied at the same time at random locations. Similarly, if the user provided a dictionary with interesting strings, they are added in random positions. Linked to the havoc stage is the *splicing stage*, in which two different inputs are combined at a random position.

3.2 Methods/Techniques

The authors introduce a lightweight, yet very effective approach to facilitate and optimize state-of-the-art feedback fuzzing that easily scales to large binary applications and unknown environments. We observe that during the execution of a given program, parts of the input often end up directly (i.e., nearly unmodified) in the program state. This *input-to-state correspondence* can be exploited to create a robust method to overcome common fuzzing roadblocks in a highly effective and efficient manner. Their prototype implementation, called REDQUEEN, is able to solve magic bytes and (nested) checksum tests automatically for a given binary executable. Additionally, The authors show that the techniques outperform various state-of-the-art tools on a wide variety of targets across different privilege levels (kernel-space and userland) with no platform-specific code.

3.3 Results/Evaluation

REDQUEEN is the first method to find more than 100% of the bugs planted in LAVA-M across all targets. Furthermore, The authors discovered 65 new bugs and obtained 16 CVEs in multiple programs and OS kernel drivers. Finally, their evaluation demonstrates that REDQUEEN is fast, widely applicable and outperforms concurrent approaches by up to three orders of magnitude.

Available at: `https://github.com/RUB-SysSec/redqueen`

3.4 Limitations/My Comments

- REDQUEEN cannot deal with those cases in which the input does not correspond to the state, such as compression or hash maps in the input.
- It would be beneficial to use this lightweight approach as a first step where possible, and than solve the remaining challenges using complex approaches.

4 GRIMOIRE: Synthesizing Structure while Fuzzing @Security'19

4.1 Background/Problems

One common challenge for current fuzzing techniques are programs which process highly structured input languages such as interpreters, compilers, text-based network protocols or markup languages. Typically, such inputs are consumed by the program in two stages: parsing and semantic analysis. If parsing of the input fails, deeper parts of the target program—containing the actual application logic—fail to execute; hence, bugs hidden "deep" in the code cannot be reached. Even advanced feedback fuzzers— such as AFL—are typically unable to produce diverse sets of syntactically valid inputs.

Previous approaches to address this problem are typically based on manually provided grammars or seed corpora. On the downside, such methods require human experts to (often manually) specify the grammar or suitable seed corpora, which becomes next to impossible for applications with undocumented or proprietary input specifications. An orthogonal line of work tries to utilize advanced program analysis techniques to automatically infer grammars. Typically performed as a pre-processing step, such methods are used for generating a grammar that guides the fuzzing process. However, since this grammar is treated as immutable, no additional learning takes place during the actual fuzzing run.

4.2 Methods/Techniques

The authors present the design and implementation of GRIMOIRE, a fully automated coverage-guided fuzzer which works without any form of human interaction or pre-configuration; yet, it is still able to efficiently test programs that expect highly structured inputs. Thet achieve this by performing large-scale mutations in the program input space using grammar-like combinations to synthesize new highly structured inputs without any pre-processing step.

Their approach is based on two key observations: First, they can use code coverage feedback to automatically infer structural properties of the input language. Second, the precise and "correct" grammars generated by previous approaches are actually unnecessary in practice: since fuzzers have the virtue of high test case throughput, they can deal with a significant amount of noise and imprecision. In fact, in some programs (such as Boolector) with a rather diverse set of input languages, the additional noise even benefits the fuzz testing. In a similar vein, there are often program paths which can only be accessed by inputs outside of the formal specifications, e. g., due to incomplete or imprecise implementations or error handling code.

4.3 Results/Evaluation

First, The authors evaluate GRIMOIRE against other fuzzers on four scripting language interpreters (mruby, PHP, Lua and JavaScriptCore), a compiler (TCC), an assembler (NASM), a database (SQLite), a parser (libxml) and an SMT solver (Boolector). GRIMOIRE outperforms all existing coverage-guided fuzzers; in the case of Boolector, GRIMOIRE finds up to 87% more coverage than the baseline (REDQUEEN).

Second, they evaluate GRIMOIRE against state-of-the-art grammar-based fuzzers. they observe that in situations where an input specification is available, it is advisable to use GRIMOIRE in addition to a grammar fuzzer to further increase the test coverage found by grammar fuzzers.

Third, they evaluate GRIMOIRE against current state-of-the-art approaches that use automatically inferred grammars for fuzzing and found that they can significantly outperform such approaches. Overall, GRIMOIRE found 19 distinct memory corruption bugs that they manually verified. they responsibly disclosed all of them to the vendors and obtained 11 CVEs. During their evaluation, the next best fuzzer only found 5 of these bugs. In fact, GRIMOIRE found more bugs than all five other fuzzers combined.

Available at: `https://github.com/RUB-SysSec/grimoire`

Table 1: Requirements and limitations of different fuzzers and inference tools when used for fuzzing structured input languages. If a shortcoming applies to a tool, it is denoted with ✗, otherwise with ✓.

	PEACH	AFL	REDQUEEN	QSYM	ANGORA	NAUTILUS	AFLSMART	GLADE	AUTOGRAM	PYGMALION	GRIMOIRE
human assistance	✗	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓
source code	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓
environment model	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓
good corpus	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓
format specifications	✗	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓
certain parsers	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
small-scale mutations	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓

Figure 3: Advantage of Grimoire.

4.4 Limitations/My Comments

The approach has significant difficulties with more syntactically complex constructs, such as matching the ID of opening and closing tags in XML or identifying variable constructs in scripting languages.

The generalization approach might be too coarse in many places. Obtaining more precise rules would help uncovering deeper parts of the target application in cases where multiple valid statements have to be produced.

5 Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing @CCS'19

5.1 Background/Problems

Hybrid fuzzing is promising in light of the recent performance improvements in concolic engines. However, there is room for further improvement: symbolic emulation is still slow, unnecessary constraints dominate solving time, resources are overly allocated, and hard-to-trigger bugs are missed.

5.2 Methods/Techniques

The authors present a new hybrid fuzzer named Intriguer (Fig.4). The key idea of Intriguer is field-level constraint solving, which optimizes symbolic execution with field-level knowledge. Intriguer performs instruction-level taint analysis and records execution traces without data transfer instructions like mov. Intriguer then reduces the execution traces for tainted instructions that accessed a wide range of input bytes, and infers input fields to build field transition trees. With these optimizations, Intriguer can efficiently perform symbolic emulation for more relevant instructions and invoke a solver for complicated constraints only.

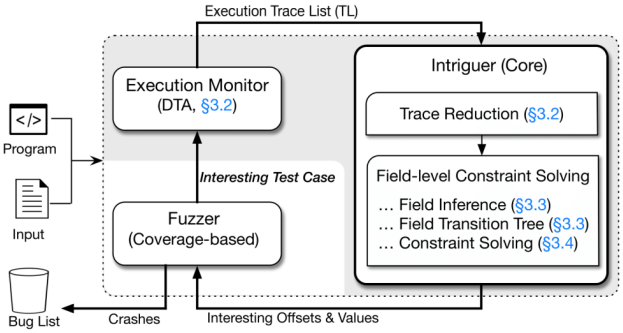

Figure 4: Intriguer's arch.

5.3 Results/Evaluation

Evaluation results indicate that Intriguer outperforms the state-of-the-art fuzzers: Intriguer found all the bugs in the LAVA-M(5h) benchmark dataset for ground truth performance, and also discovered 43 new security bugs in seven real-world programs. They reported the bugs and received 23 new CVEs.

5.4 Limitations/My Comments

Intriguer performed field inference by inspecting off sets recorded in the execution traces and did not consider the association between those fields. To perform field-level constraint solving more efficiently, Intriguer should identify fields of the same type through the FL and grouping them together. In addition to identifying repeated fields, we may infer structure from those fields and then consider mutation strategies specific to repeated fields based on it.

Intriguer reduces the execution traces to emulate only the small portion of the instructions that are repeatedly used to access a wide range of input bytes. One may have two concerns regarding trace reduction. First, excessively reduced traces may affect a constraint solving for important branches. By considering the context-sensitivity of the run-time process, Intriguer will not perform trace reduction when a program is in a different context (e.g., different call stack). Second, the instructions can be repeatedly used to access only a narrow range of input bytes. Note that an execution bottleneck can also occur if the instructions are used to repetitively access the input with specific offsets. We can address this problem by reducing the execution traces for the instructions that use the same offset by considering the program's context.

Intriguer currently supports most of the x86 instruction set and a part of x86_64 instruction set. Although it is challenging to support all of x86_64 instructions, we can implement frequently used instructions to support actual program execution.

The current version of Intriguer does not consider the control-flow dependency, e.g., occurring from indirect and conditional jump.

6 FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation @Sec'19

6.1 Background/Problems

Two fundamental problems in *IoT fuzzing* due to its strong dependency on the actual hardware configuration: 1) Compatibility issues by enabling fuzzing for POSIX-compatible firmware that can be emulated in a system emulator. 2) Performance bottleneck caused by system-mode emulation.

6.2 Methods/Techniques

A novel technique called augmented process emulation. By combining system- mode emulation (high generality and low efficiency) and user-mode emulation (low generality and high efficiency) in a novel way, augmented process emulation provides high compatibility as system-mode emulation and high throughput as user-mode emulation. The program to be fuzzed is mainly run in user-mode emulation to achieve high efficiency, and switches to full system emulation only when necessary to ensure correct program execution.

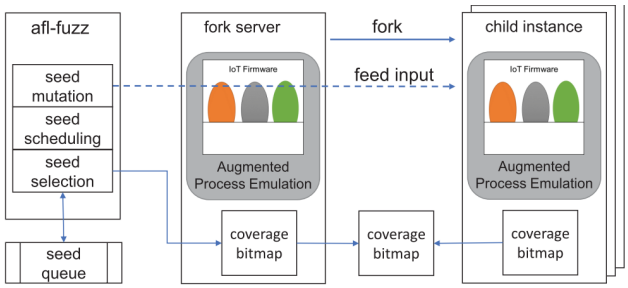

Figure 5: firm-afl's arch.

6.3 Results/Evaluation

Evaluation results show that (1) FIRM-AFL is fully functional and capable of finding real-world vulnerabilities in IoT programs; (2) the throughput of FIRM-AFL is on average 8.2 times higher than system-mode emulation based fuzzing; and (3) FIRM-AFL can find 1-day vulnerabili- ties much faster than system-mode emulation based fuzzing, and find 0-day vulnerabilities.

available at `https://github.com/zyw-200/FirmAFL`.

6.4 Limitations/My Comments

FIRM-AFL only supports the following CPU ar- chitectures: mipsel, mipseb and armel. It can also only fuzz a program in a firmware image that can be properly emulated by Firmadyne and runs a POSIX-compatible OS (e.g., Linux).