Reading Notes

Trusted Software and Intelligent System Lab., BUPT

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 authors propose 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.

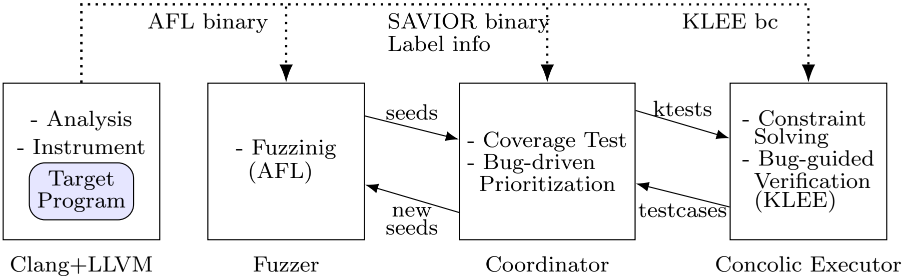


Figure 1: SAVIOR's arch.

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/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.

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.2, 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.

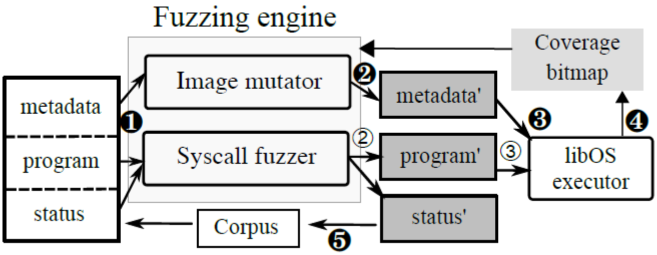


Figure 2: JANUS' arch.

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/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.

```
/* 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);
```

Listing 1: Roadblocks for feedback-driven fuzzing.

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. They 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/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.

Core design of AFL

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.

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 (Fig.3), 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.

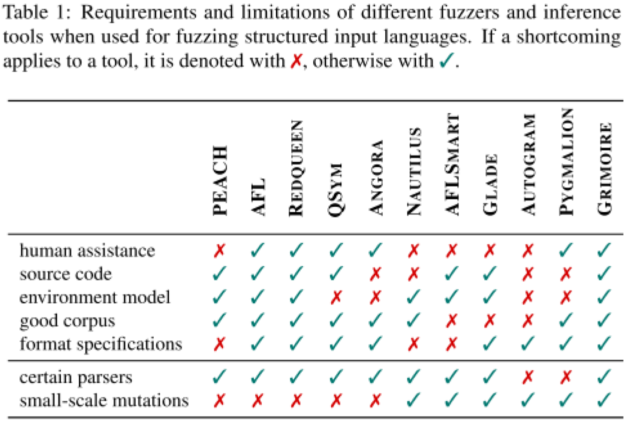


Table 1: Requirements and limitations of different fuzzers and inference tools when used for fuzzing structured input languages. If a shortcoming applies to a tool, it is denoted with ✗, otherwise with ✓.

	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.

4.3 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

4.4 Limitations/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.

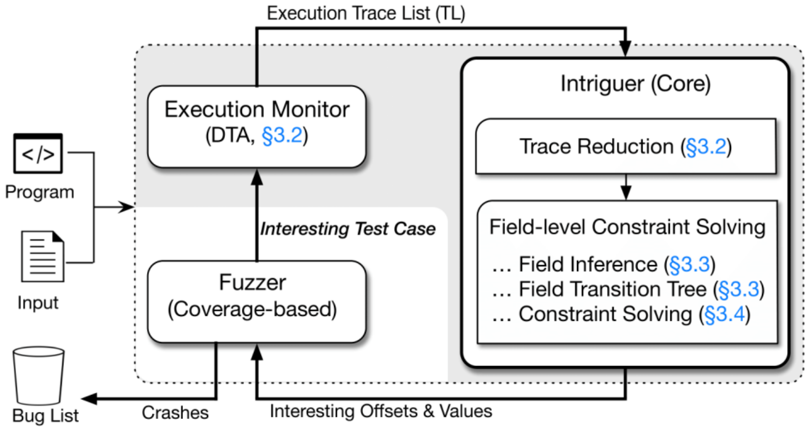

Execution Trace List (TL)
Execution Monitor (DTA, §3.2)
Intriguer (Core)
Trace Reduction (§3.2)
Field-level Constraint Solving
... Field Inference (§3.3)
... Field Transition Tree (§3.3)
... Constraint Solving (§3.4)
</>
Program
Input
Interesting Test Case
Fuzzer (Coverage-based)
Bug List
Crashes
Interesting Offsets & Values

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/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.

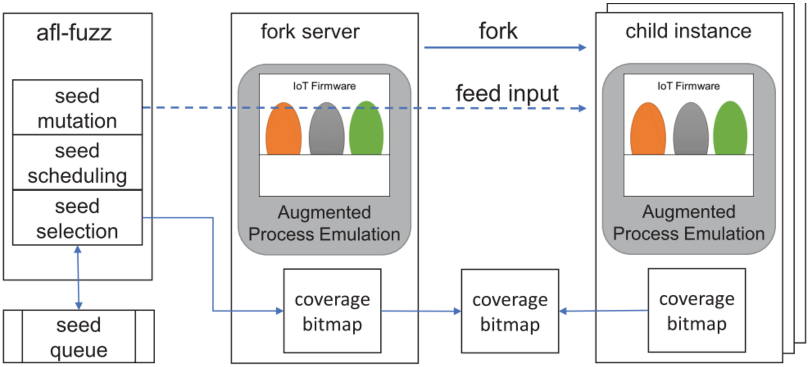


Figure 5: firm-afl's arch.

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 vulnerabilities much faster than system-mode emulation based fuzzing, and find 0-day vulnerabilities.

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

6.4 Limitations/Comments

FIRM-AFL only supports the following CPU architectures: 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).

7 RVFUZZER: Finding Input Validation Bugs in Robotic Vehicles Through Control-Guided Testing @Security'19

7.1 Background/Problems

Attack surface of robotic vehicles (RVs) spans multiple aspects, such as (1) physical vulnerabilities of its sensors that enable external sensor spoofing attacks [72,77,80]; (2) traditional "syntactic" bugs in its control program (e.g., memory corruption bugs) that enable remote or trojaned exploits [75]; and (3) control-semantic bugs in its control program that enable attacks via remote control commands. For attacks exploiting (1) and (2), there have been research efforts in defending against them [30,38,40,50,52,70,76]; whereas those exploiting (3) have not received sufficient attention.

In this paper, the authors address a new type of vulnerability in RV control programs, called input validation bugs, which involve missing or incorrect validation checks on control parameter inputs. Such bugs can be exploited to cause physical disruptions to RVs which may result in mission failures and vehicle damages or crashes. Furthermore, attacks exploiting such bugs have a very small footprint: just one innocent-looking ground control command, requiring no code injection, control flow hijacking or sensor spoofing.

7.2 Methods/Techniques

Testing RV control programs to find input validation bugs is challenging due to many different RV models (e.g., quadcopters and ground rovers) with a large number of hardware, software and control configuration options. Moreover, traditional fuzzing techniques are not directly applicable to RV control programs because: (1) With hundreds of configurable parameters, the control program has an extremely large input space to explore and (2) there is no uniform and obvious condition to automatically decide that a control program is malfunctioning. Many input validation bugs do not exhibit system-level symptoms until certain control and physical conditions are met at run-time.

Our solution to finding input validation bugs – without control program source code – is motivated by the following ideas: (1) The impacts of attacks exploiting input validation bugs can be manifested by the victim vehicle's control state; and (2) such state can be efficiently reproduced by combining the RV control program and a high-fidelity RV simulation framework, which is readily available [7,8].

Based on these ideas, they develop RVFUZZER (Fig.6), a vetting system for finding input validation bugs in RV control programs through control-guided input mutation. The key insight behind RVFUZZER is that the RV control model, which is the generic theoretical model for a broad range of RVs, provides helpful semantic guidance to improve bug-discovery accuracy and efficiency. Specifically, RVFUZZER involves a control instability detector that detects control program misbehavior, by observing (simulated) physical operations of the RV based on the control model. In addition, RVFUZZER steers the input generation for finding input validation bugs more efficiently, by leveraging results from the control instability detector as feedback.

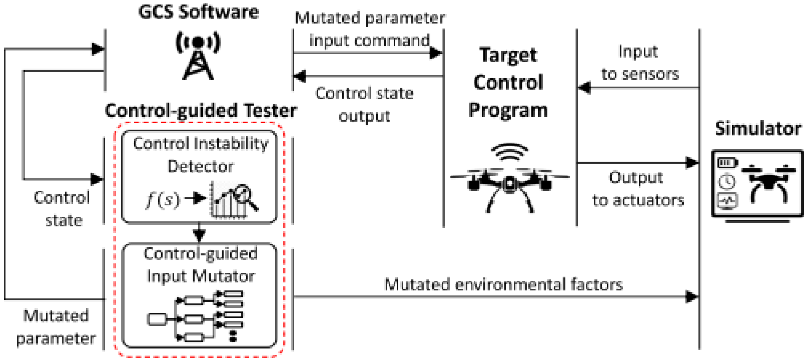


Figure 6: rvfuzzer's arch.

7.3 Results/Evaluation

In their evaluation of RVFUZZER on two popular RV control programs (ArduPilot [15] and PX4 [24]), a total of 89 input validation bugs are found, with 87 of them being zero-day bugs.

7.4 Limitations/Comments

The control parameters may have dependencies on one another. In this work, they consider the subject control program binary as a black box and take a pragmatic approach by only revealing part of such inter-dependencies. A more generic approach to control parameter dependency derivation – possibly based on source code and a formal control model – is left as future work.

There has been no standard safety testing framework created for RVs. We believe that RVFUZZER's post-production, black-box-based vetting will serve as a useful complement to standardized safety testing during RV design and production.

8 ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery @S&P'19

8.1 Background/Problems

Existing mutation based fuzzers tend to randomly mutate the input of a program without understanding its underlying syntax and semantics. This information (fields and their semantics) may not be documented and available to the fuzzer, and can be hard to recover without going through an in-depth heavyweight analysis procedure.

8.2 Methods/Techniques

In this paper, the authors propose a novel on-the-fly probing technique (called ProFuzzer, Fig.7) that automatically recovers and understands input fields of critical importance to vulnerability discovery during a fuzzing process and intelligently adapts the mutation strategy to enhance the chance of hitting zero-day targets. Since such probing is transparently piggybacked to the regular fuzzing, no prior knowledge of the input specification is needed. During fuzzing, individual bytes are first mutated and their fuzzing results are automatically analyzed to link those related together and identify the type for the field connecting them; these bytes are further mutated together following type-specific strategies, which substantially prunes the search space. They define the probe types generally across all applications, thereby making their technique application agnostic.

ProFuzzer is inspired by two important observations. First, a comprehensive, application-specific and semantically rich input specification is not necessary for fuzzing. Second, inputs can be understood and their fields and data types can be discovered by directly observing the fuzzing process, particularly the ways the input content is mutated and the program's execution path variations in response to the mutations.

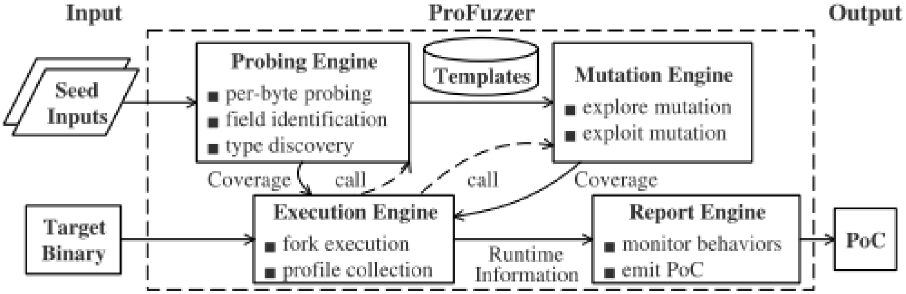


Figure 7: profuzzer's arch.

8.3 Results/Evaluation

Eexperiments on standard benchmarks and real-world applications show that ProFuzzer substantially outperforms AFL and its optimized version AFLFast, as well as other state-of-art fuzzers including VUzzer, Driller and QSYM. Within two months, it exposed 42 zero-days in 10 intensively tested programs, generating 30 CVEs.

8.4 Limitations/Comments

Currently, the exploitation mutation procedure requires domain knowledge and manual efforts. A future work is the automatic learning of exploitation rules from a larger pool of PoC.

Review the assumptions that the target application has the following properties. First, its execution is deterministic: that is, given the same input, multiple executions of the application all follow the same execution path and yield the same result. Second, initial valid seed inputs of reasonable size are available. Third, if the validation on certain bytes fails, the execution will quickly terminate, which means that an exceptional execution has a shorter execution path than a normal one.

9 RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization @S&P'20

9.1 Background/Problems

The current state of the art for applying fuzzing or sanitization to binaries is dynamic binary translation, which has prohibitive performance overhead. The alternate technique, static binary rewriting, cannot fully recover symbolization information and hence has difficulty modifying binaries to track code coverage for fuzzing or to add security checks for sanitizers.

9.2 Methods/Techniques

In this paper, the authors show that static binary rewriting, leveraging reassembleable assembly, can produce sound and efficient code for an important class of binaries: *64-bit position-independent code (PIC)*. Notably, such binaries include third party shared libraries, the analysis of which is the most pressing use-case for such a rewriter. The rewriting technique, called RetroWrite (Fig.8), leverages relocation information which is required for position independent code, and produces assembly files that can be reassembled into binaries.

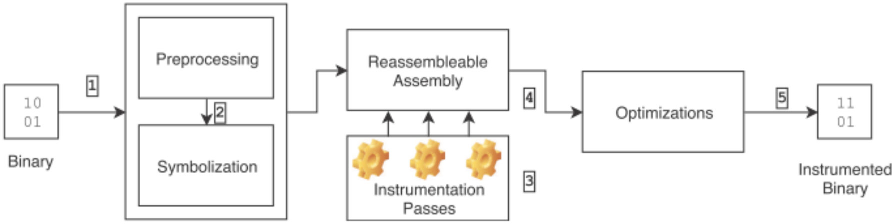


Figure 8: retrowrite's arch.

9.3 Results/Evaluation

Binaries rewritten for coverage-guided fuzzing using RetroWrite are identical in performance to compiler-instrumented binaries and outperform the default QEMU-based instrumentation by 4.5x while triggering more bugs. Their implementation of binary-only Address Sanitizer is 3x faster than Valgrind's memcheck, the state-of-the-art binary-only memory checker, and detects 80% more bugs in the evaluation. Available at: https://github.com/HexHive/retrowrite.

9.4 Limitations/Comments

a) Support for C++ Binaries: The current implementation of RetroWrite cannot rewrite C++ binaries safely due to missing symbolization for C++ exception handlers.

b) Closing the Performance Gap: Another opportunity to reduce overhead is to remove unnecessary checks when a memory access is known to be safe, e.g., accessing variables on stack through constant offsets from the stack top.

c) Limitations of ASan-retrowrite: The limitations of ASan-retrowrite on stack and global sections are fundamental to static binary rewriting. To improve precision on stack and data sections, we may need to trade-off soundness or scalability. One attractive option is to use local symbolic execution to track base-pointers and disambiguate references.

d) Obfuscation: To protect intellectual property, some vendors ship obfuscated binaries. Retrowrite does not address obfuscation. Binary unpacking is usually specific to the obfuscation scheme used; and an obfuscated binary may be rewritten by RetroWrite, after it is pre-processed by a de-obfuscation step [57], [58].

10 SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel @CCS'19

10.1 Background/Problems

To determine the exploitability for a kernel vulnerability, a security analyst usually has to manipulate slab and thus demonstrate the capability of obtaining the control over a program counter or performing privilege escalation. However, this is a lengthy process because (1) an analyst typically has no clue about what objects and system calls are useful for kernel exploitation and (2) he lacks the knowledge of manipulating a slab and obtaining the desired layout. In the past, researchers have proposed various techniques to facilitate exploit development. Unfortunately, none of them can be easily applied to address these challenges due to the complexity of the Linux kernel and the dynamics and non-deterministic of slab variations.

10.2 Methods/Techniques

To tackle the challenges, the authors first use static and dynamic analysis techniques to explore the kernel objects, and the corresponding system calls useful for exploitation. Second, they model commonly-adopted exploitation methods and develop a technical approach to facilitate the slab layout adjustment. By extending LLVM as well as Syzkaller, they implement the techniques and name their combination after SLAKE.

More specifically, they first perform reachability analysis over the call graph and preserve only those system calls that could potentially reach to the sites of interest. For each site of interest, they then perform fuzz testing using the results of reachability analysis, exploring the actual path towards that site.

10.3 Results/Evaluation

They evaluate SLAKE by using 27 real-world kernel vulnerabilities, demonstrating that it could not only diversify the ways to perform kernel exploitation but also sometimes escalate the exploitability of kernel vulnerabilities. Available at: https://github.com/chenyueqi/SLAKE

10.4 Limitations/Comments

Some special situations are ignored in object identification. For example, userland data can be copied first to the kernel stack through a system call and then migrated to the slab through a kernel function (e.g., memcpy()). To identify spray objects in this special situation, an accurate inter-procedural data flow analysis is inevitable.

Other exploitation methods. In addition to the four exploitation methods discussed in Section 2, security researchers have developed other approaches for exploiting some special cases (e.g., [16, 21]) as well as heap-based use-before-initialization vulnerability.

Vulnerability capability. The authors manually extract vulnerability capabilities from a PoC program under the guidance of debugging tools. Technically speaking, this process could be potentially automated by using dynamic analysis methods such as symbolic tracing. vulnerability capability exploration is a non-trivial problem, which might need the integration of various advanced techniques in program analysis.

Other allocators. SLAKE introduces a systematic approach for slab layout manipulation. To extend this approach to other kernel allocators (e.g., SLOB allocator [25], buddy system [5]) or those used in userland (e.g., ptmalloc[12]), a modification is required.

The approach can be used for other open-source OSes (e.g., FreeBSD and Android).

11 Block Oriented Programming: Automating Data-Only Attacks @CCS'18

11.1 Background/Problems

With the widespread deployment of Control-Flow Integrity (CFI), control-flow hijacking attacks, and consequently code reuse attacks, are significantly more difficult. CFI limits control flow to well-known locations, severely restricting arbitrary code execution. Assessing the remaining attack surface of an application under advanced control-flow hijack defenses such as CFI and shadow stacks remains an open problem.

11.2 Methods/Techniques

The authors introduce BOPC (Block Oriented Programming Compiler, Fig.9), a mechanism to automatically assess whether an attacker can execute arbitrary code on a binary hardened with CFI/shadow stack defenses. BOPC computes exploits for a target program from payload specifications written in a Turing-complete, high-level language called SPloit Language (SPL) that abstracts away architecture and program-specific details. SPL payloads are compiled into a program trace that executes the desired behavior on top of the target binary. The input for BOPC is an SPL payload, a starting point (e.g., from a fuzzer crash) and an arbitrary memory write primitive that allows application state corruption. To map SPL payloads to a program trace, BOPC introduces Block Oriented Programming (BOP), a new code reuse technique that utilizes entire basic blocks as gadgets along valid execution paths in the program, i.e., without violating CFI or shadow stack policies. They find that the problem of mapping payloads to program traces is NP-hard, so BOPC first reduces the search space by pruning infeasible paths and then uses heuristics to guide the search to probable paths. BOPC encodes the BOP payload as a set of memory writes.

The core component of BOPC is the mapping process through a novel code reuse technique called Block Oriented Programming (BOP). First, BOPC translates the SPL payload into constraints for individual statements and, for each statement, searches for basic blocks in the target binary that satisfy these constraints (called *candidate blocks*). At this point, SPL abstracts register assignments from the underlying architecture. Second, BOPC infers a resource (register and state) mapping for each SPL statement, iterating through the set of candidate blocks and turning them into *functional blocks*. Functional blocks can be used to execute a concrete instantiation of the given SPL statement. Third, BOPC constructs a trace that connects each functional block through *dispatcher blocks*. Since the mapping process is NP-hard, to find a solution in reasonable time BOPC first prunes the set of functional blocks per statement to constrain the search space and then uses a ranking based on the proximity of individual functional blocks as a heuristic when searching for dispatcher gadgets.

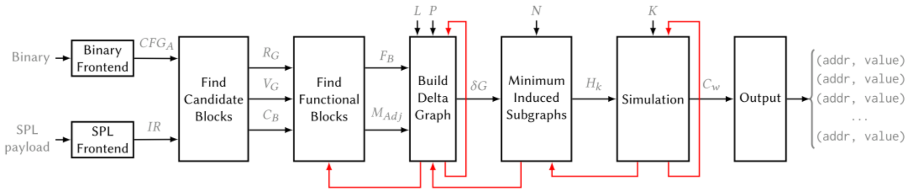


Figure 9: bopc's arch. The red arrows indicate the iterative process upon failure. CFGA: CFG with basic block abstractions added, IR:Compiled SPL payload RG:Register mapping graph, VG:All variable mapping graphs, CB: Set of candidate blocks, FB: Set of functional blocks, MAdj:Adjacency matrix of SPL payload, δG:Delta graph, Hk:Induced subgraph, Cw:Constraint set. L:Maximum length of continuous dispatcher blocks, P:Upper bound on payload "shuffles", N:Upper bound on minimum induced subgraphs, K:Upper bound on shortest paths for dispathers.

11.3 Results/Evaluation

They execute 13 SPL payloads applied to 10 popular applications. BOPC successfully finds payloads and complex execution traces – which would likely not have been found through manual analysis – while following the target's Control-Flow Graph under an ideal CFI policy in 81% of the cases. Available at https://github.com/HexHive/BOPC.

11.4 Limitations/Comments

BOPC is limited by the granularity of basic blocks. That is, a combination of basic blocks could potentially lead to the execution of a desired SPL statement, while individual blocks might not. Take for instance an instruction that sets a virtual register to 1. Assume that a basic block initializes rcx to 0, while the following block increments it by 1; a pattern commonly encountered in loops. Al- though there is no functional block that directly sets rcx to 1, the combination of the previous two has the desired effect. BOPC can be expanded to address this issue if the basic blocks are coalesced into larger blocks that result in a new CFG.

BOPC sets several upper bounds defined by user inputs. These configurable bounds include the upper limit of (i) SPL payload permutations (P), (ii) length of continuous blocks (L), (iii) of minimum induced subgraphs extracted from the delta graph (N), and (iv) dispatcher paths between a pair of functional blocks (K). These upper bounds along with the timeout for symbolic execution, reduce the search space, but prune some potentially valid solutions. The evaluation of higher limits may result in alternate or more solutions being found by BOPC.

12 Automatic Heap Layout Manipulation for Exploitation @Security'18

12.1 Background/Problems

As of 2018, the most common approach to solving heap layout manipulation (HLM) problems is manual work by experts. An analyst examines the allocator's implementation to gain an understanding of its internals; then, at runtime, they inspect the state of its various data structures to determine what interactions are necessary in order to manipulate the heap into the required layout. The process is complicated by the fact that – when constructing an exploit – one cannot directly interact with the allocator, but instead must use the API exposed by the target program.

There are four variants of the HLM problem, depending on whether the allocator is deterministic or non-deterministic and whether the starting state is known or unknown. In this paper the authors consider a known starting state and a deterministic allocator, and assume there are no other actors interacting with the heap.

Their objective in this variant of the HLM problem is as follows: *Given the API for a target program and a means by which to allocate a source and destination buffer, find a sequence of API calls that position the destination and source at a specific offset from each other.*

12.2 Methods/Techniques

The authors present the first automatic approach to the problem, based on pseudo-random black-box search. Our approach searches for the inputs required to place the source of a heap-based buffer overflow or underflow next to heap-allocated objects that an exploit developer, or automatic exploit generation system, wishes to read or corrupt. They present a framework for benchmarking heap layout manipulation algorithms, and use it to evaluate our approach on several real-world allocators, showing that pseudo-random black box search can be highly effective. They then present SHRIKE (Fig.10), a novel system that can perform automatic heap layout manipulation on the PHP interpreter and can be used in the construction of control-flow hijacking exploits.

Starting from PHP's regression tests, SHRIKE discovers fragments of PHP code that interact with the interpreter's heap in useful ways, such as making allocations and deallocations of particular sizes, or allocating objects containing sensitive data, such as pointers. SHRIKE then uses our search algorithm to piece together these fragments into programs, searching for one that achieves a desired heap layout. SHRIKE allows an exploit developer to focus on the higher level concepts in an exploit, and to defer the resolution of heap layout constraints to SHRIKE. Available at https://sean.heelan.io/heaplayout.

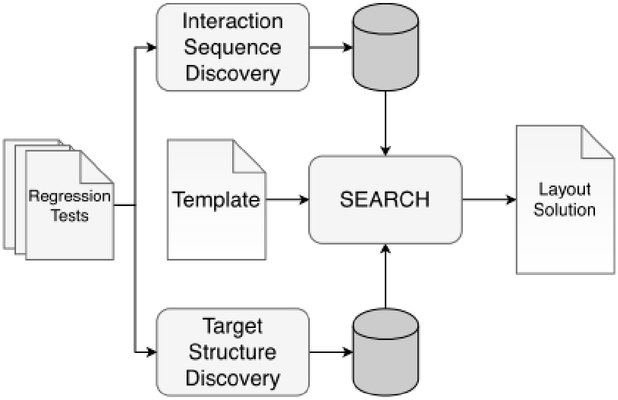


Figure 10: shrike's arch.

12.3 Results/Evaluation

They chose the tcmalloc (v2.6.1), dlmalloc (v2.8.6) and avrlibc (v2.0) allocators for experimentation. They demonstrate this by using SHRIKE in the construction of a control-flow hijacking exploit for the PHP interpreter. SHRIKE had a 70% success rate overall, and a 100% success rate in cases where there was no noise.

12.4 Limitations/Comments

Other three variants of the HLM problem are harder than the one addressed in this paper. Some allocators, e.g., Windows system allocator, jemalloc and the DIEHARD family of allocators, do utilise non-determinism to make exploitation more difficult.

Read other references, e.g., [7, 15, 26, 32, 33], for more future works.

13 MarkUs: Drop-in use-after-free prevention for low-level languages @S&P'20

13.1 Background/Problems

A variety of techniques have been proposed to mitigate use-after-free vulnerabilities in C and C++. For example, all pointer locations can be logged and then nullified when their data is freed [1], [4], [5], objects allocated with their own page-table entries [6], [7], or probabilistic reuse delays employed [8]–[10]. However, these tend to exhibit both high average- and worst-case overheads in terms of performance and memory utilisation, or have limited coverage.

13.2 Methods/Techniques

The authors present MarkUs, a memory allocator that prevents this form of attack at low overhead, sufficient for deployment in real software, even under allocation- and memory-intensive scenarios. MarkUs prevent use-after-free attacks by *quarantining data freed by the programmer and forbidding its reallocation until there are no dangling pointers targeting it.* To identify these MarkUs traverses live-objects accessible from registers and memory, marking those it encounters, to check whether quarantined data is accessible from any currently allocated location.

Unlike garbage collection, which is unsafe in C and C++, MarkUs ensures safety by *only freeing data that is both quarantined by the programmer and has no identifiable dangling pointers.* The information provided by the programmer's allocations and frees further allows to optimise the process by freeing physical addresses early for large objects, specialising analysis for small objects, and only performing marking when sufficient data is in quarantine.

13.3 Results/Evaluation

Using MarkUs, they reduce the overheads of temporal safety in low-level languages to 1.1× on average for SPEC CPU2006, with a maximum slowdown of only 2×, vastly improving upon four state-of-the-art techniques, i.e., Oscar, DangSan, pSweeper and CRCount.

13.4 Limitations/Comments

MarkUs fails to detect complex use-after-free vulnerabilities involving hidden pointers, as is a limitation with any technique that involves identifying pointers.

Tagged memory techniques [36,37] can make MarkUs more efficient, by allowing reuse of memory multiple times, based on incrementing the ID tag of each successive allocation, before address space must be quarantined to ensure old IDs have been eliminated and can be reallocated.

14 KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware @S&P'20

14.1 Background/Problems

Low-power, single-purpose embedded devices (e.g., routers and IoT devices) have become ubiquitous. While they automate and simplify many aspects of users' lives, recent large-scale attacks have shown that their sheer number poses a severe threat to the Internet infrastructure. Unfortunately, the software on these systems is hardware-dependent, and typically executes in unique, minimal environments with non-standard configurations, making security analysis particularly challenging.

Many of the existing devices implement their functionality through the use of multiple binaries. This multi-binary service implementation renders current static and dynamic analysis techniques either ineffective or inefficient, as they are unable to identify and adequately model the communication between the various executables.

14.2 Methods/Techniques

Based on the intuition that binaries communicate using a finite set of Inter-Process Communication (IPC) paradigms, the authors present KARONTE, a static analysis approach capable of analyzing embedded-device firmware by modeling and tracking multi-binary interactions. The approach propagates taint information between binaries to detect insecure interactions and identify vulnerabilities.

14.3 Results/Evaluation

First, they evaluated KARONTE on 53 firmware samples from various vendors, showing that the tool can successfully track and constrain multi-binary interactions. This led to the discovery of 46 zero-day bugs. Then, they performed a large-scale experiment on 899 different samples, showing that KARONTE scales well with firmware samples of different size and complexity. Available at: https://github.com/ucsb-seclab/karonte.

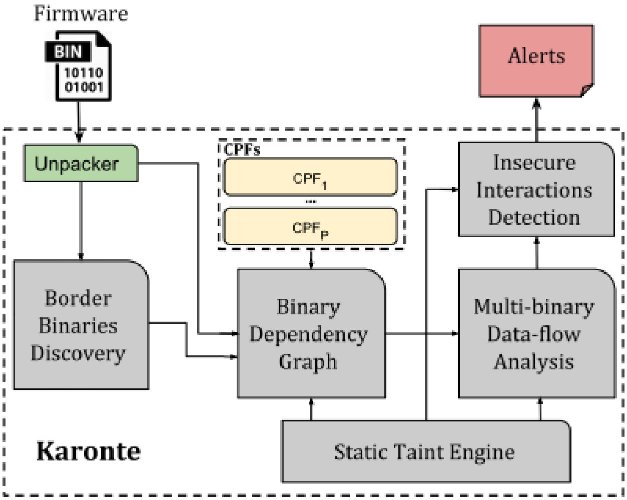


Figure 11: karonte's arch. After unpacking a firmware sample, KARONTE extracts the binaries handling user requests, identifies their data dependencies to build the Binary Dependency Graph (BDG), and uses its inter-binary taint analysis engine to find insecure data flows. CPF: Communication Paradigm Finder.

14.4 Limitations/Comments

KARONTE suffers from the path explosion problem. they limit path explosion by: (i) providing precise taint propagation policies, (ii) using timeouts, (iii) limiting loop iterations, and (iv) automatically creating function summaries.

Karonte may generate both false positives and false negatives. They are due to the fact that taint information might not be correctly propagated to unfollowed paths (e.g., due to time, call-stack depth, or loop constraints), or imprecisions of the underlying static analysis tool (i.e., angr). This might result in incomplete BDGs, and, therefore, some security vulnerabilities might be left undiscovered.

Though by default, KARONTE finds buffer overflows and denial-of-service vulnerabilities, its design allows an analyst to support different types of vulnerabilities. For instance, an analyst can extend Karonte to find use-after-free bugs by providing a new detection module, such as [16].

15 What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices @NDSS'18

15.1 Background/Problems

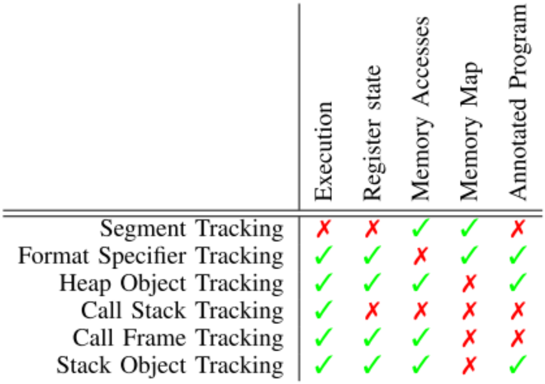
While common desktop systems have a variety of mechanisms to detect faulty states (e.g., segmentation faults, heap hardening and sanitizers) and to analyze them (e.g., command-line return values or core dumps), embedded devices often lack such mechanisms because of their limited I/O capabilities, constrained cost, and limited computing power. As a result, *silent memory corruptions* occur more frequently on embedded devices than on traditional computer systems, creating a significant challenge for conducting fuzzing sessions on embedded systems software.

15.2 Methods/Techniques

The authors analyze the challenges of fuzzing embedded devices (fault detection, performance and scalability, instrumentation), make a classification of embedded systems with respect to the difficulty of detecting memory corruptions in their software, evaluate the real world effects of different memory corruptions on those classes of embedded systems, describe the techniques that can be used for improving fuzzing on embedded devices, and present six heuristics that can be used to detect faults due to the memory corruption (tracking segment, format specifier, heap object, stack object, call stack, call frame, respectively, Fig.12), when the firmware of an embedded system can be run in either a partial or full emulation environment.

15.3 Results/Evaluation

They implemented the heuristics on top of a combination of the Avatar [54] and PANDA [13] frameworks. They use PANDA to emulate the firmware and rely on its plugin system to obtain live feedback over the execution of a partial or fully emulated firmware. Avatar allows the tester to save and replay the device state after it is initialized.



	Execution	Register state	Memory Accesses	Memory Map	Annotated Program
Segment Tracking	✗	✗	✓	✓	✗
Format Specifier Tracking	✓	✓	✗	✓	✓
Heap Object Tracking	✓	✓	✓	✗	✓
Call Stack Tracking	✓	✗	✗	✗	✗
Call Frame Tracking	✓	✓	✓	✗	✗
Stack Object Tracking	✓	✓	✓	✗	✓

Figure 12: Deployed live analysis techniques and their requirements.

They conducted a number of tests to show their effectiveness and their overhead in a fuzzing experiment. Experiments show that partial emulation significantly reduces the fuzzing throughput. However, when full emulation is possible, this setup can combine the best of both worlds and detect 100% of the corrupting inputs while improving the fuzzing efficiency beyond what could be obtained using the physical device. Available at: https://github.com/avatartwo/ndss18_wycinwyc.

15.4 Limitations/Comments

They relied on artificial bugs in the experiments, which lacks of evidents in real applications. When migrating from an artificial test scenario to real world software, the observed false positive and negative rates of the individual heuristics will vary.

The ability to fully emulate arbitrary firmware images is still an open problem. Therefore, the solution discussed in this paper may not be applicable to all scenarios and all embedded devices.

16 Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps @Security'19

16.1 Background/Problems

Smart home IoT devices have increasingly become a favorite target for the cybercriminals due to their weak security designs. To identify these vulnerable devices, existing approaches rely on the analysis of either real devices or their firmware images. These approaches, unfortunately, are difficult to scale in the highly fragmented IoT market due to the unavailability of firmware images and the high cost involved in acquiring real-world devices for security analysis.

16.2 Methods/Techniques

The authors present a platform (Fig.13) that accelerates vulnerable device discovery and analysis, without requiring the presence of actual devices or firmware. Our approach is based on two key observations: First, IoT devices tend to reuse and customize others' components (e.g., software, hardware, protocol, and services), so vulnerabilities found in one device are often present in others. Second, reused components can be indirectly inferred from the mobile companion apps of the devices; so a cross analysis of mobile companion apps may allow us to approximate the similarity between devices.

(1) app analysis: find the characteristics of a device by analyzing its companion app, and (2) cross-app analysis: find device families, i.e. cluster of devices, that have similarity in some of the characteristics found in app analysis by analyzing multiple apps. Clustering helps identify apps that have a similar set of vulnerabilities based on shared components.

16.3 Results/Evaluation

They perform a large-scale analysis involving over 4,700 devices and 2000 apps. Their study brings to light the sharing of vulnerable components across the smart home IoT devices (e.g., shared vulnerable protocol, backend services, device rebranding), and leads to the discovery of 324 devices from 73 different vendors that are likely to be vulnerable to a set of security issues.

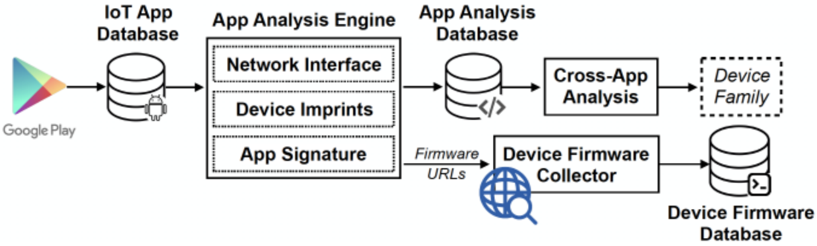


Figure 13: mirror's arch.

16.4 Limitations/Comments

The major limitation of the approach is the accuracy of the analysis results. As the analysis solely on mobile companion apps, the information obtained may not be an accurate reflection of the device. For example, a device may have patched a vulnerability and the patch did not change the device interfaces at all. In this case, the analysis will still output the device as potentially vulnerable. A multi-stage solution can help address this limitation where the first stage (i.e., the platform) narrows down the scope of analysis by identifying the potentially vulnerable devices, and the second stage automates the vulnerability confirmation with more targeted but rigorous analysis, e.g., dynamic/static analysis of firmware, device fuzzing.

Another limitation of the approach is that the network interface analysis can be rendered less effective in scenarios where IoT backend servers or cloud significantly decouple device interfaces from app interfaces. An example is the Google and Amazon devices where much of the management is done through the cloud.

Another aspect to improve on is the dimension and granularity of the similarity analysis. Further improvements to the App Analysis Engine may allow the platform to detect similarities in finer components of a device software stack (e.g., web server, PHP interpreter, web application, OS, driver) as well as other dimensions (e.g., similar developer, similar development toolchain). This would enable us to track vulnerability propagation more comprehensively and accurately.

17 PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary @NDSS'19

17.1 Background/Problems

Currently, most of the kernel's attack surface is situated along the system call boundary. Ongoing kernel protection efforts have focused primarily on securing this boundary. However, there are additional paths to kernel compromise that do not involve system calls, as demonstrated by several recent exploits. For example, by compromising the firmware of a peripheral device such as a Wi-Fi chipset and subsequently sending malicious inputs from the Wi-Fi chipset to the Wi-Fi driver, adversaries have been able to gain control over the kernel without invoking a single system call. Unfortunately, there are currently no practical probing and fuzzing frameworks that can help developers find and fix such vulnerabilities occurring along the hardware-OS boundary.

17.2 Methods/Techniques

The authors present PERISCOPE (Fig.14), a Linux kernel based probing framework that enables fine-grained analysis of device-driver interactions. PERISCOPE hooks into the kernel's *page fault handling* mechanism to either passively monitor and log traffic between device drivers and their corresponding hardware, or mutate the data stream on-the-fly using a fuzzing component, PERIFUZZ, thus mimicking an active adversarial attack. PERIFUZZ accurately models the capabilities of an attacker on peripheral devices, to expose different classes of bugs including, but not limited to, memory corruption bugs and double-fetch bugs.

17.3 Results/Evaluation

To demonstrate the risk that peripheral devices pose, as well as the value of their framework, they have evaluated PERIFUZZ on the Wi-Fi drivers of two popular chipset vendors[1], where they discovered 15 unique vulnerabilities,

[1] The Google Pixel 2 and Samsung Galaxy S6 are equipped with Qualcomm and Broadcom chipsets, respectively.

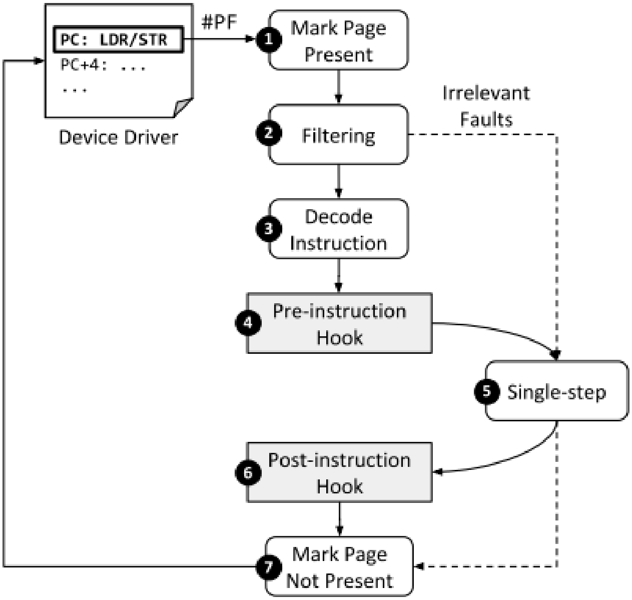


Figure 14: periscope's arch.

9 of which were previously unknown. Available at: https://github.com/securesystemslab/periscope.

17.4 Limitations/Comments

Augmenting the fuzzing engine: Like DIFUZE [35], static analysis can be introduced to infer the type of an I/O buffer, which can save fuzzing cycles by respecting the target type when mutating a value. The dependencies between device-driver interaction messages can also be inferred using static and trace analysis techniques [41], [58], which can help fuzzing stateful device-driver interaction protocols. Alternatively, developers can specify the format of an I/O buffer and/or interaction protocol in a domain-specific language [10], [75]. In addition to improving the mutation of the data stream, they could use system call fuzzers such as Syzkaller that generate different user-space programs [75]. These generated programs could actively send requests to the driver and potentially to the device, which in turn can increase reachable interrupt code paths.

Combining with Dynamic Analysis: PeriScope runs in a concrete execution environment; thus, existing dynamic analysis tools can be used to uncover silent bugs. For example, kernel sanitizers such as address sanitizer and undefined behavior sanitizer can complement their fuzzer [48], [63]. Memory safety bugs often silently corrupt memory without crashing the kernel. PeriFuzz, by itself, cannot reveal such bugs. When combined with a sanitizer, however, these bugs would be detected. Other dynamic analysis techniques such as dynamic taint tracking can also be adapted to detect security-critical semantic bugs such as passing security-sensitive values (e.g., kernel virtual addresses) to untrusted peripherals.

Crashes in kernel space cause a system reboot, which significantly lowers the throughput of any kernel fuzzer. They circumvented this problem by disabling certain code paths that contain previously discovered shallow bugs. However, this does reduce the effectiveness of PeriFuzz as it cannot traverse the subpaths rooted at these blacklisted bugs. Note that this problem also affects other kernel fuzzers, e.g., DIFUZE and Syzkaller.

Due to the significant latency involved in system restarts, whole-system fuzzers typically fuzz the system without restarting it between fuzzing iterations. This can limit the effectiveness of such fuzzers, because the internal states of the target system persist across iterations. Changing internal states can also lead to instability in the coverage-guidance, as the same input can exercise different code paths depending on the system state. This means that coverage-guidance may not be fully effective. Existing device driver checkpointing and recovery mechanisms could be adapted to alleviate the problem [46], [70], because they provide mechanisms to roll drivers back to an earlier state. Such a roll back takes significantly less time than a full system reboot.

18 PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation @Security'20

18.1 Background/Problems

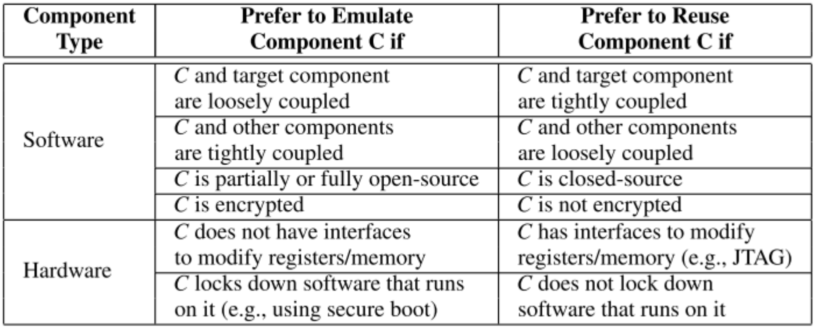
ARM's TrustZone technology is the basis for security of billions of devices worldwide, including Android smartphones and IoT devices. Because TrustZone has access to sensitive information such as cryptographic keys, access to TrustZone has been locked down on real-world devices: only code that is authenticated by a trusted party can run in TrustZone. A side-effect is that TrustZone software cannot be instrumented or monitored. Thus, recent advances in dynamic analysis techniques such as feedback-driven fuzz testing have not been applied to TrustZone software. Researchers have been restricted to primarily static reverse-engineering of binaries to find vulnerabilities in TrustZone software.

18.2 Methods/Techniques

The authors build an emulator that runs four widely-used, real-world TrustZone operating systems (TZOSes) - Qualcomm's QSEE, Trustonic's Kinibi, Samsung's TEEGRIS, and Linaro's OP-TEE - and the trusted applications (TAs) that run on them. The traditional challenge for this approach is that the emulation effort required is often impractical. However, they find that TZOSes depend only on a limited subset of hardware and software components. By carefully choosing a subset of components to emulate (Fig.15), they make the effort practical and implement the emulation on PARTEMU, a modular framework they develop on QEMU and PANDA.

18.3 Results/Evaluation

They show the utility of PARTEMU by integrating feedback-driven fuzz-testing using AFL and use it to perform a large-scale study of 194 unique TAs from 12 different Android smartphone vendors and a leading IoT vendor, finding previously unknown vulnerabilities in 48 TAs, several of which are exploitable. They identify patterns of developer mistakes unique to TrustZone development



Component Type	Prefer to Emulate Component C if	Prefer to Reuse Component C if
Software	*C* and target component are loosely coupled	*C* and target component are tightly coupled
	C and other components are tightly coupled	*C* and other components are loosely coupled
	C is partially or fully open-source	*C* is closed-source
	C is encrypted	*C* is not encrypted
Hardware	*C* does not have interfaces to modify registers/memory	*C* has interfaces to modify registers/memory (e.g., JTAG)
	C locks down software that runs on it (e.g., using secure boot)	*C* does not lock down software that runs on it

Figure 15: Criteria to decide whether to reuse or emulate a component C. As in object-oriented design, they use "loosely-coupled" to mean components that have well-defined interfaces with each other and work largely independently of each other, and "tightly-coupled" to mean the opposite, that is, components that need to know each others' internal data structure implementations, leading to complicated interfaces and deep dependencies.

that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. They also demonstrate using PARTEMU to test the QSEE TZOS itself, finding crashes in code paths that would not normally be exercised on a real device. Our work shows that dynamic analysis of real-world TrustZone software through emulation is both feasible and beneficial.

18.4 Limitations/Comments

Dealing with Stateful TAs. On a random sample of 10 TAs, AFL had basic-block coverage varying from 0.2% to 45.6% with a median of 17.7%. A major limiting factor for coverage was TA state: they noticed that several TAs had internal finite state machines and therefore required a sequence of multiple inputs to drive them to interesting states (e.g., connected, authorized, processing). Our driver currently sends a single message to a newly forked TA instance each time so that AFL does not have issues with stability. Therefore, they cannot get past state checks, which require a sequence of inputs.

Hardware Roots of Trust. PARTEMU does not emulate hardware roots of trust. An example is the factory-installed per-device private key signed by the Samsung CA and used for remote attestation. Thus, code paths in TAs that depend on remote attestation succeeding may not work. For example, Samsung Pay uses remote attestation for credit card enrollment; they cannot successfully enroll a credit card using a Samsung Pay TA running on PARTEMU because they do not have access to the attestation key that would be present on a real device.

Performance. Since they ran PARTEMU on an x86 machine, they could not take advantage of ARMv8 hardware virtualization. AFL ran at around 10-25 executions per second for QSEE, OP-TEE, and TEEGRIS, while their performance optimizations for Kinibi enabled 125 executions per second. Thus, a future work is to explore running PARTEMU directly on ARMv8 hardware.

19 Enhancing example-based code search with functional semantics @Journal of Systems and Software'20

19.1 Background/Problems

As the quality and quantity of open source code increase, effective and efficient search for code implementing certain semantics, or semantics-based code search, has become an emerging need for software developers to retrieve and reuse existing source code. Previous techniques in semantics-based code search encode the semantics of loop-free Java code snippets as constraints and utilize an SMT solver to find encoded snippets that match an input/output (IO) query.

19.2 Methods/Techniques

The authors present in this article the Quebio approach to semantics-based search for Java methods (Fig.16). Quebio advances the state-of-the-art by supporting important language features like invocation to library APIs and enabling the search to handle more data types like array/List, Set, and Map. Compared with existing approaches, Quebio also integrates a customized keyword-based search that uses as the input a textual, behavioral summary of the desired methods to quickly prune the methods to be checked against the IO examples.

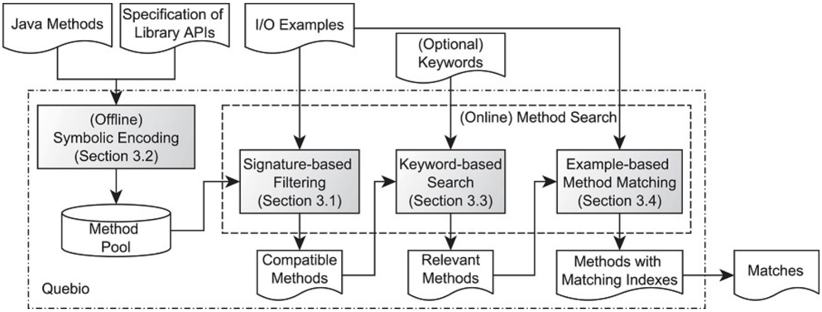

Java Methods
Specification of Library APIs
I/O Examples
(Optional) Keywords
(Offline) Symbolic Encoding (Section 3.2)
(Online) Method Search
Signature-based Filtering (Section 3.1)
Keyword-based Search (Section 3.3)
Example-based Method Matching (Section 3.4)
Method Pool
Compatible Methods
Relevant Methods
Methods with Matching Indexes
Matches
Quebio

Figure 16: semantics-based search for Java method using IO examples

19.3 Results/Evaluation

conducted experiments on 47 queries based on real-world questions from programmers to evaluate our approach. The approach was able to find correct methods from a pool of 14,792 candidates for 43 of the queries, spending on average 213.2 seconds on each query. Such results suggest our approach is both effective and efficient.

19.4 Limitations/Comments

A threat to external validity concerns the limited data types that Quebio supports in its current implementation.While Quebio has the ability to handle API calls in candidate methods, the availability and the complexity of logical formulas encoding those APIs' semantics can present challenges to Quebio in achieving similar levels of effectiveness and efficiency on other queries. In the future, we plan to extend Quebio to support more data types.

20 Active Inductive Logic Programming for Code Search @ICSE'20

20.1 Background/Problems

Modern search techniques either cannot efficiently incorporate human feedback to refine search results or cannot express structural or semantic properties of desired code.

20.2 Methods/Techniques

The authors present in this article the Quebio approach to semantics-based search for Java methods. Quebio advances the state-of-the-art by supporting important language features like invocation to library APIs and enabling the search to handle more data types like array/List, Set, and Map. Compared with existing approaches, Quebio also integrates a customized keyword-based search that uses as the input a textual, behavioral summary of the desired methods to quickly prune the methods to be checked against the IO examples.

20.3 Results/Evaluation

On average, ALICE successfully identifies similar code locations with 93% precision and 96% recall in 2.7 search iterations. ALICE achieves 100% precision and 97% recall in the first dataset, while it achieves 75% precision and 94% recall in the second dataset. The reason is that the second dataset contains code fragments that loop over a double array with no write to output operations, which is a semantic constraint imposed by Casper [20] for loop optimization. However, ALICE does not learn predicates that differentiate read and write operations on an array and therefore returns a large number of code fragments that write double arrays in a loop, which leads to low precision.

compare with critics: In six out of seven cases, ALICE achieves the same or better precision and recall with fewer iterations to converge, compared to Critics. In ID 4, ALICE has low precision because the expected code contains a switch statement, which is currently not extracted by ALICE as a logic fact. Extending current logic predicates to support more syntactic constructs remain as future work.

20.4 Limitations/Comments

we generate facts based on structural and intra-procedural control flow properties. Other types of analysis such as dataflow analysis or aliasing analysis could be used in identifying similar snippets. In addition, the query language itself can be extended to make it easier to capture the properties of desired code. For example, by introducing negations in the query language,a user can specify atoms that should not be included. There could be specializations that strictly require negations. However, in our experiments, empirically, we are always able to find a pattern without negations. As mentioned in Section III-D, our learning process is monotonic and to learn a different query, a user may need to start over. To overcome this, we may need backtracking and investigate new search algorithms that generalize and specialize the query in a different way.

21 LTRWES:A new framework for security bug report detection @Information and Software Technology'20

21.1 Background/Problems

It is important to identify SBRs quickly and accurately among bug reports (BRs) that have been disclosed in bug tracking systems. Although a few methods have been already proposed for the detection of SBRs, challenging issues still remain due to noisy samples, class imbalance and data scarcity.

21.2 Methods/Techniques

LTRWES is a content-based data filtering and representation framework that has several desirable properties not shared in other methods (Fig.17). Firstly, it exploits ranking model to efficiently filter non-security bug reports (NSBRs) that have higher content similarity with respect to SBRs. Secondly, it applies word embedding technology to transform the rest of NSBRs, together with SBRs, into low-dimensional real-value vectors.

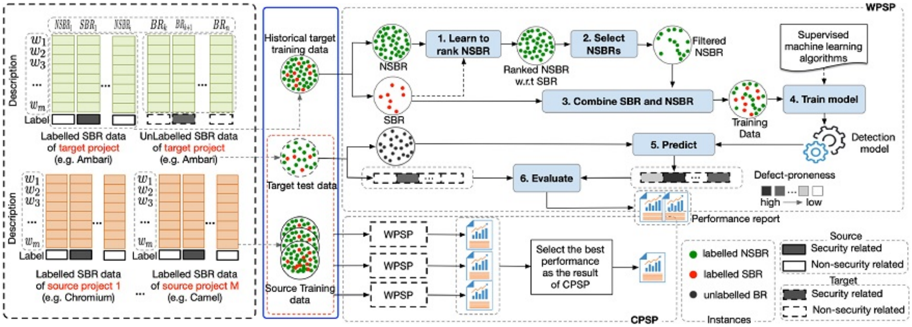


Figure 17: the general framework of security bug report detection

21.3 Results/Evaluation

The highest g-measure for each project is obtained by LTRWES. LTRWES outperforms FARSEC on average by 16.7% and 22.4% in gmeasure for WPSP and CPSP respectively. LTRWES outperforms FARSEC by 25.4% and 54.5% on average in recall for WPSP and CPSP.

21.4 Limitations/Comments

Our future work includes:

- develop an improved method that use more projects as sources to further improve the performance of crossproject SBRs prediction,
- combine textual information with side information in bug reports (such as product, component) to improve the performance of SBRs prediction,
- develop more effective methods to reduce the false positive rate while maintaining high true positive rate in detecting security bug reports,
- adopt other hyperparameter tuning methods to further improve the performance of prediction models.

22 Search-Based Test and Improvement of Machine-Learning-Based Anomaly Detection Systems @ISSTA'19

22.1 Background/Problems

The live learning mechanism, progressively injecting small portions of abnormal data. Swift the learned states to a point where harmful data can pass unnoticed.

22.2 Methods/Techniques

- **Search-Based Test of IDS(intrusion detection systems):** –Genotype;
  –Fitness function;(the distance between target and the closest cluster centroid and minimizing the consumed time budget)
  –Selectors (keeps the best individuals out of samples of three)
  –alterers(single-point crossover with a 0.2 recombination probability, while gene mutation in the offspring occurs with a probability of 0.15)

- **Search-Based Improvement of IDS:** –Genotype;
  –Fitness function(fastest detection time, minimizing resources);
  –Selectors and alterers

- **Co-Evolution of Attacks and Defences:**

22.3 Results/Evaluation

dataset: 4SICS Geek Lounge dataset, contains 18 hours of real SCADA traffic data. detecting DoS training attacks over a network. **results:** The defences resulting from the highest iterations detected 49 attacks out of 50.

22.4 Limitations/Comments

- We design a testing framework based on search algorithms that, given an IDS, can generate a test (i.e. a training attack scheme) to deceive the IDS. We instantiate this framework in a denial-of-service attack scenario, where a training attack consists of unnoticed, successive increases in data rate.

- We generalize the countermeasure proposed by Muller et al. [11] and design a genetic algorithm to search for a defence strategy (in the form of new parameter sets) that succeeds in detecting the attack induced by a given test case.

- We show that our testing framework enables the continuous improvement of IDS detection capabilities. More precisely, we set up a co-evolution process that attractively generates attacks and defences leading to new parameter sets that make the IDS resilient to all generated attacks.

23 Codebase-adaptive detection of security-relevant methods @ISSTA'19

23.1 Background/Problems

Users must configure the static tools with lists of security-relevant methods (Srm) that are relevant to their development context.aids analysis users in detecting Srm.

23.2 Methods/Techniques

- **Features:** uses a set of binary features that evaluate certain properties of the methods, constructs a feature matrix; identified 25 feature types, instantiated as 206 concrete features, **Classifiers:** Support Vector Machine (SVM), Bayes Net, Naive Bayes, Logistic Regression, C4.5, Decision Stump, and Ripper.
- take user feedback into account, allowing developers to adapt SWAN to the code base.

23.3 Results/Evaluation

datsets: twelve popular Java frameworks **results:** SWAN achieves an average precision of 0.826, which is better or comparable to existing approaches. SWANAssist requires a relatively low effort from the developer to significantly improve its precision.

23.4 Limitations/Comments

It is possible to improve the precision of static analyses by providing more granular Srm information. Not only the methods themselves are important, which objects they affect can also be important (i.e., which parameter, static variable, return variable, or base object). Additionally, we plan to extend SWAN and SWANAssist to support a larger number of CWEs. We also plan to improve SWAN' training set in a more systematic manner, to ensure a better precision of the approach, and develop a better strategy to handle potentially problematic methods in SuggestSWAN.

24 P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling @Security'20

24.1 Background/Problems

Software vulnerabilities cause the majority of attacks on MCU devices, resulting in not only digital but also physical damages. Dynamic testing or fuzzing of embedded firmware is severely limited by hardware-dependence and poor scalability, partly contributing to the widespread vulnerable IoT devices. There are four open challenges that prevent fuzzers for computer software from being effective on firmware: Hardware Dependence; Wide Range of Peripherals; Diverse OS/System Designs; Incompatible Fuzzing Interfaces.

24.2 Methods/Techniques

The authors design a framework that supports fuzzers as drop-in components to test firmware in a scalable and hardware-independent fashion. The framework aims to solve the MCU-imposed fuzzing challenges while allowing fuzzers to focus on performing and improving their own job (i.e., generating inputs and finding bugs).

The approach is novel in that it neither relies on any hardware nor emulates peripherals. They introduce a form of approximate MCU emulation for supporting firmware testing and fuzzing. More importantly, they provide a method to automatically generate approximate emulators based on firmware binaries. The approach is inspired by the observation that firmware can execute on an emulator without real or fully emulated peripherals, as long as the emulator provides the firmware with acceptable inputs from peripherals when needed. Such inputs do not have to be the same as what a real peripheral would produce. But they do need to pass firmware's internal checks to avoid disrupting firmware execution. For a given firmware, their approximate emulator uses a generic processor/ISA emulator (e.g., one for ARM Cortex-M) and a model, automatically built for the firmware, that captures what constitutes an acceptable input for each peripheral accessed by the firmware.

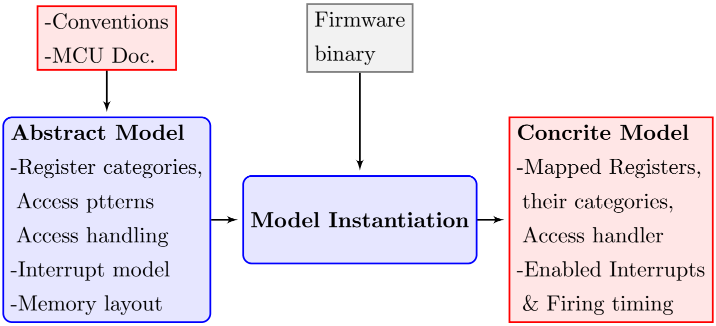


Figure 18: P2IM's workflow.

24.3 Results/Evaluation

They implemented the framework using QEMU as the base processor emulator. Their implementation includes 2,202 lines of C code added to QEMU (mostly for dynamic firmware execution instrumentation), 173 lines of C code for fuzzer integration, and 1,199 lines of Python code for the explorative execution part of P2IM. They use AFL as the drop-in fuzzer. AFL's emulation mode (used for fuzzing un-instrumented binaries) only supports user-mode emulation, which is incompatible with firmware emulation [13]. TriforceAFL [40] builds a bridge for AFL to be connected to the full system emulation mode of QEMU. They used TriforceAFL's code when implementing the fuzzer integration part of their framework.

They evaluated the framework using 70 sample firmware and 10 firmware from real devices, including a drone, a robot,and a PLC. It successfully executed 79% of the sample firmware without any manual assistance. They also performed a limited fuzzing test on the real firmware, which unveiled 7 unique unknown bugs.

24.4 Limitations/Comments

P2IM models the processorperipheral interfaces, including registers and interrupts. It does not model Direct Memory Access (DMA), which allows peripherals to directly access RAM and in turn provide input to firmware. The lack of DMA support is a limitation of the work.

They analyzed 3 MCUs that use non-ARM architectures for IoT devices: ATmega328P (AVR), PIC32MX440F256H (MIPS) and FE310-G000 (RISC-V). P2IM and the current abstract model can be extended to handle these architectural differences and in turn support these non-ARM MCU architectures.

Other types of dynamic firmware analysis that do not require fully accurate output from firmware can use the framework to achieve hardware-independence and scalability. For instance, data or code reachability analysis, such as taint analysis and certain debugging tasks, can benefit from their framework. In particular, concolic firmware execution can use their framework to generate more realistic concrete inputs (i.e., non-crashing/stalling), reduce the number of symbolic values, and avoid some infeasible code paths.

25 HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation @Security'20

25.1 Background/Problems

Given the increasing ubiquity of online embedded devices, analyzing their firmware is important to security, privacy, and safety. The tight coupling between hardware and firmware and the diversity found in embedded systems makes it hard to perform dynamic analysis on firmware.

25.2 Methods/Techniques

Firmware developers regularly develop code using abstractions, such as Hardware Abstraction Layers (HALs), to simplify their job. Therefore, the authors leverage such abstractions as the basis for the re-hosting and analysis of firmware. By providing high-level replace- ments for HAL functions (a process termed *High-Level Emulation* – HLE), they decouple the hardware from the firmware. This approach works by first identifying the library functions in a firmware sample, through binary analysis, and then providing generic implementations of these functions in a full-system emulator.

They present these ideas in a prototype system, HALucinator (Fig.19), able to re-host firmware, and allow the virtual device to be used normally. First, they introduce extensions to existing library matching techniques that are needed to identify library functions in binary firmware, to reduce collisions, and for inferring additional function names. Next, they demonstrate the re-hosting process, through the use of simplified *handlers* and *peripheral models*, which make the process fast, flexible, and portable between firmware samples and chip vendors.

Specifically, LibMatch creates a database of HAL functions to match by extracting the control-flow graph of the unlinked binary object files of the libraries, plus an In- termediate Representation (IR) of their code. It then performs the following steps to successively refine possible matches: Statistical comparison; Basic block comparison; and Contextual matching.

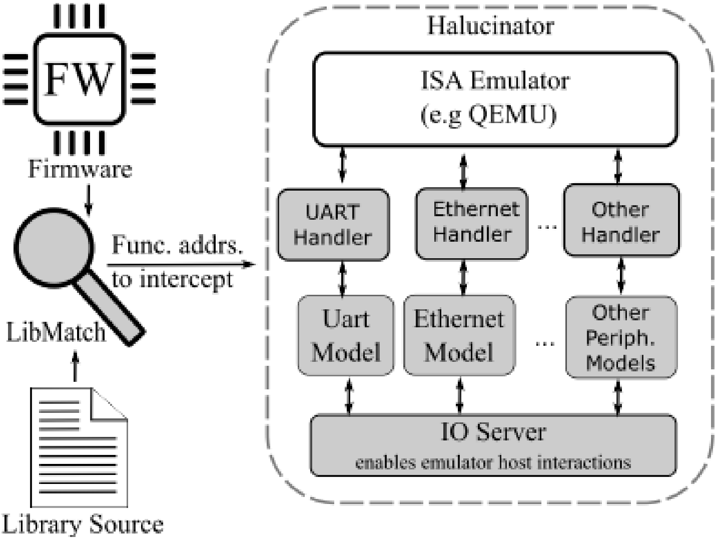


Figure 19: HALucinator's arch.

25.3 Results/Evaluation

The authors demonstrate the practicality of HLE for security analysis, by supplementing HALucinator with AFL fuzzer, to locate multiple previously-unknown vulnerabilities in firmware middleware libraries. they use 16 firmware samples provided with different development boards from Atmel, NXP, and STM. These samples were chosen for their diverse and complex hardware interactions, including serial communication, file systems on SD cards, Ethernet, 6LoWPAN, and WiFi. They also contain a range of sophisticated application logic, including wireless messaging over 6LoWPAN, a Ladder Logic interpreter, and an HTTP Server with a Common Gateway Interface (CGI). The set of included libraries is also diverse, featuring STMicroelectronics' STM32-Cube HAL [53], NXP's MCUXpresso [42], Atmel's Advanced Software Framework (ASF) [16], lwIP [37], FatFS [27], and Contiki-OS [25]. Available at: `https://github.com/embedded-sec/halucinator` based on angr+Avatar^2, and `https://github.com/ucsb-seclab/hal-fuzz` based on AFL-Unicore.

25.4 Limitations/Comments

HALucinate requires the firmware use a HAL, and the HAL must be available to the analyst (e.g., either open source, or part of the microcontroller's SDK). The compilation environment for the LibMatch database must be similar to the compilation environment for the firmware, and QEMU must support the microcontroller architecture. Even when these conditions are met, handlers and peripheral models must be developed for each HAL. If a HAL is not used in a firmware sample, or is unavailable to the analyst, then LibMatch cannot be used for identifying interfaces usable for high-level emulation.

The effectiveness of LibMatch, especially when the compiler or library versions used is unknown, is limited. This limitation comes from function matching techniques' inability to cope with compiler-induced variations in generated code.

26 KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities @Security'20

26.1 Background/Problems

The monolithic nature of modern OS kernels leads to a constant stream of bugs being discovered. It is often unclear which of these bugs are worth fixing, as only a subset of them may be serious enough to lead to security takeovers (i.e., privilege escalations). Therefore, researchers have recently started to develop automated exploit generation techniques (for UAF bugs) to assist the bug triage process.

26.2 Methods/Techniques

The authors investigate another top memory vulnerability in Linux kernel — out-of-bounds (OOB) memory write from heap. They design KOOBE to assist the analysis of such vulnerabilities based on two observations: (1) Surprisingly often, different OOB vulnerability instances exhibit a wide range of capabilities. (2) Kernel exploits are multi-interaction in nature (i.e., multiple syscalls are involved in an exploit) which allows the exploit crafting process to be modular. Specifically, they focus on the extraction of capabilities of an OOB vulnerability which will feed the subsequent exploitability evaluation process. Our system builds on several building blocks, including a novel capability-guided fuzzing solution to uncover hidden capabilities, and a way to compose capabilities together to further enhance the likelihood of successful exploitations.

26.3 Results/Evaluation

They have implemented a prototype of Koobe on top of the popular kernel fuzzer Syzkaller, binary symbolic execution framework S2E and binary analysis engine angr. It consists of 7,510 LOC of C++ to the S2E for capability summarization and exploitability evaluation, 2,271 LOC of python based on Angr to analyze vulnerabilities, and 1,106 LOC of Go to explore diverging paths with fuzzing and synthesize exploits. Availabe at: https://github.com/seclab-ucr/KOOBE

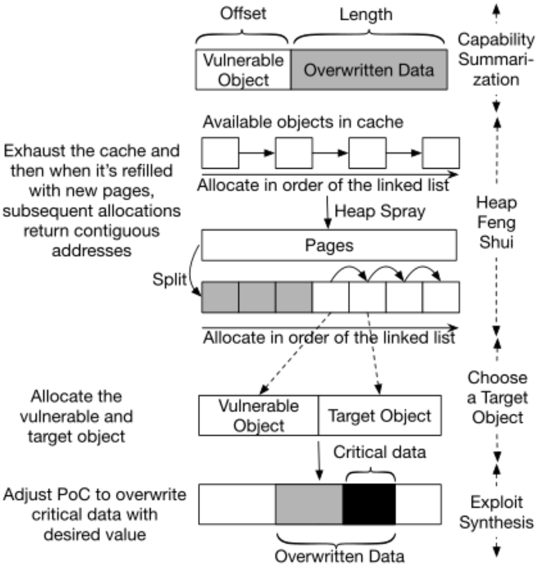

Offset
Length
Vulnerable Object
Overwritten Data
Capability Summarization
Available objects in cache
Exhaust the cache and then when it's refilled with new pages, subsequent allocations return contiguous addresses
Allocate in order of the linked list
Heap Spray
Heap Feng Shui
Pages
Split
Allocate in order of the linked list
Choose a Target Object
Allocate the vulnerable and target object
Vulnerable Object
Target Object
Critical data
Adjust PoC to overwrite critical data with desired value
Exploit Synthesis
Overwritten Data

Figure 20: Koobe's workflow.

They demonstrate the applicability of KOOBE by exhaustively analyzing 17 most recent Linux kernel OOB vulnerabilities (where only 5 of them have publicly available exploits), for which KOOBE successfully generated candidate exploit strategies for 11 of them (including 5 that do not even have any CVEs assigned). Subsequently from these strategies, they are able to construct fully working exploits for all of them.

26.4 Limitations/Comments

KOOBE does not yet produce an end-to-end exploit fully automatically. Through this study, they identify and automate the key procedures of crafting kernel heap OOB write exploits. To close the entire automation loop, they also point out several interesting places: (1) Exploring heap feng shui. Our system leverages existing heap feng shui strategies without the ability to handle complex scenarios. Prior work [28] has shed some light on this problem in the context of user applications. Following the same direction, they could automate this

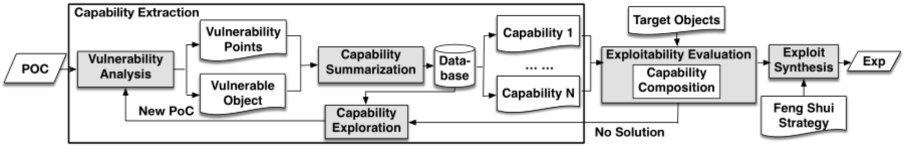


Figure 21: Koobe's architecture

process by applying fuzzing. (2) Turning IP-hijacking primitives into arbitrary code execution and privilege escalation. The recent work [56] proposes a novel solution to bypass SMEP and SMAP, given an IP hijacking primitive. By integrating this technique, leveraging side channels capable of defeating KASLR, or relying on another information disclosure vulnerability, Koobe could produce end-to-end exploits. (3) Probability configurations for fuzzing. They currently choose each queue with equal probability during fuzzing. It is a trade-off between focusing on seeds of interest and exploring uncovered paths that do not offer new capabilities yet but lead to long-term benefit. A higher probability for selecting the seeds increasing coverage allows us to quickly explore uncovered code but it also slows down finding new seeds extending existing capabilities since uncovered code is mostly irrelevant and thus a substantial amount of seeds do no contribute given the large codebase of Linux kernel. Future work would be to explore different probability configuration and design approaches to dynamically adjust it during the fuzzing execution.

Although they only consider defenses deployed in practice in this work, some fine-grained randomization based defenses [3, 14, 42] would break some of their assumptions in generating exploits (e.g., DieHard [14] and SLAB/SLUB freelist randomization [3] make heap feng shui much less predictable). Nevertheless, they believe such defenses are not bulletproof. For example, randomization-based solutions could potentially be circumvented by CPU side channels that can be integrated into Koobe.

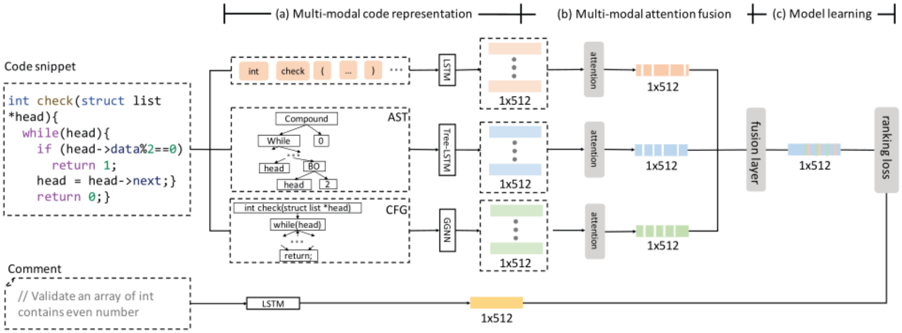
27 Multi-Modal Attention Network Learning for Semantic Source Code Retrieval @ASE'2019

27.1 Background/Problems

The paper is motivated by a paper "Deep code search," in ICSE'2018.
Code retrieval techniques and tools have been playing a key role in facilitating software developers to retrieve existing code fragments from available open-source repositories given a user query (e.g., a short natural language text describing the functionality for retrieving a particular code snippet). However, two main issues hinder them to accurately retrieve satisfiable code fragments from largescale repositories when answering complicated queries. First, the existing approaches only consider shallow features of source code such as method names and code tokens, but ignoring structured features such as abstract syntax trees (ASTs) and control-flow graphs (CFGs) of source code, which contains rich and welldefined semantics of source code. Second, although the deep learning-based approach performs well on the representation of source code, it lacks the explainability, making it hard to interpret the retrieval results and almost impossible to understand which features of source code contribute more to the final results.

27.2 Methods/Techniques

The paper proposes MMAN, a novel Multi-Modal Attention Network for semantic source code retrieval. A comprehensive multi-modal representation (Fig. 22) is developed for representing unstructured and structured features of source code, with one LSTM for the sequential tokens of code, a Tree-LSTM for the AST of code and a GGNN (Gated Graph Neural Network) for the CFG of code. Furthermore, a multi-modal attention fusion layer is applied to assign weights to different parts of each modality of source code and then integrate them into a single hybrid representation.



```
int check(struct list
*head){
  while(head){
    if (head->data%2==0)
      return 1;
    head = head->next;}
    return 0;}
```

Figure 22: MMAN's arch.

27.3 Results/Evaluation

Comprehensive experiments and analysis on a large-scale real-world dataset (28,527 commented C methods from github) show that the proposed model can accurately retrieve code snippets and outperforms the state-of-the-art methods (i.e., CodeHow, DeepCS).

27.4 Limitations/Comments

We evaluate MMAN using only two metrics, i.e., SuccessRate@R and MRR, which are both standard evaluation metrics in information retrieval. We do not use precision at some cutoff (Precision@k), since the relevant results need to be labelled manually. However, a human evaluation is also needed for the sake of fair comparison with DeepCS.

Another limitation lies in the extensibility of our proposed approach. Our model needs to be trained on a large scale of corpus, which is collected from online platforms such as GitHub. Since the writing style of different programmers may differ greatly, lots of efforts will be put into this step. In this paper, we have defined many regular expressions to extract the samples that meets our condition, at the same time, many samples are filtered. Furthermore, the CFG can only be extracted from a whole program. Therefore, it's difficult to extend our multi-modal code representation model to some contexts where the CFG are unable to be extracted, such as many code snippets from StackOverflow.

Future work: 1)conduct comprehensive experiments on other dataset of different language such as Java or Python, as well as human evaluation to further verify the effectiveness of our proposed approach. 2)it's promising to explore the potentiality of multimodal code representation on some other software engineering tasks such as code summarization and code clone detection.

28 Scalable and Practical Locking with Shuffling @SOSP'2019

28.1 Background/Problems

Locks are an essential building block for high-performance multicore system software. To meet performance goals, lock algorithms have evolved towards specialized solutions for architectural characteristics (e.g., NUMA). However, in practice, applications run on different server platforms and exhibit widely diverse behaviors that evolve with time (e.g., number of threads, number of locks). This creates performance and scalability problems for locks optimized for a single scenario and platform. For example, popular spinlocks suffer from excessive cache-line bouncing in NUMA systems, while scalable, NUMA-aware locks exhibit sub-par single-thread performance.

28.2 Methods/Techniques

Locks not only serialize data access, but also add their overhead, directly impacting application scalability. Looking at the evolution of locks and their use, we identify four main factors that any practical lock algorithm should consider. These factors are critical in achieving good performance in current architectures, but their relative importance can vary not only across architectures, but also across applications with varying requirements. Therefore, we should holistically consider all four factors when designing a lock algorithm.

1. A lock algorithm should amortize data movement from both the lock structure and the data inside the critical section, to hide non-uniform latency and minimize coherence traffic.

2. For the best performance in all scenarios, a lock algorithm should adapt to varying thread contention.

3. A lock algorithm should consider the mapping between threads and cores and whether cores are over-subscribed.

4. A lock algorithm should consider memory foot-print, as it affects both the adoption of the lock and applications performance.

We then propose a new technique, shuffling (Fig. 23), that can dynamically accommodate all these factors, without slowing down the critical path of the lock. The key idea of shuffling is to re-order the queue of threads waiting to acquire the lock in accordance with some pre-established policy. For best performance, this work is done off the critical path, by the waiter threads. Using shuffling, we demonstrate how to achieve NUMA-awareness and implement an efficient parking/wake-up strategy, without any auxiliary data structure, mostly off the critical path.

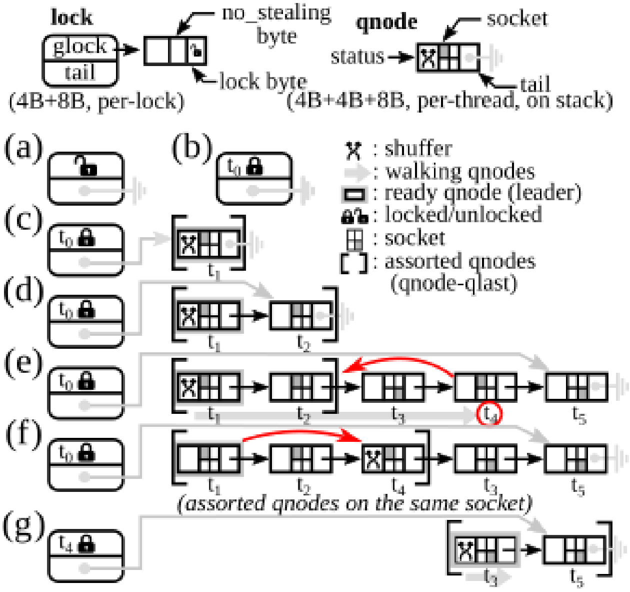


Figure 23: shflLock's example.

28.3 Results/Evaluation

The evaluation shows that our family of locks based on shuffling improves the throughput of real-world applications up to 12.5×, with impressive memory footprint reduction compared with the recent lock algorithms. For instance, ShflLocks improve performance on two fronts: they improve throughput by

1.2–1.6× while reducing the lock overhead by 35.4–95.8% at 192 threads. Availabe at https://github.com/sslab-gatech/shfllock.

28.4 Limitations/Comments

- Shuffling mechanism opens new opportunities to implement different policies based on the hardware behavior or the requirements of the application. For example, we can devise policies to support non-inclusive caches [41] or a multi-level NUMA hierarchy [43]. In this case, the shuffler optimizes the waiting list according to the NUMA node, but it also keeps track of the number of hops.

- Shuffling can also be used to avoid the priority inversion issue [28] or to devise approaches for applications that require occupancy-aware scheduling, (i.e., prioritize lock-acquire based on the time spent inside the critical section).

- Shuffling can also be beneficial in designing an adaptive readers-writer lock, in which a waiter switches among centralized, per-socket or per-CPU reader indicators, depending on workload and thread contention.

29 Typestate-guided fuzzer for discovering UAF vulnerabilities @ICSE'20

29.1 Background/Problems

Existing coverage-based fuzzers usually use the individual control flow graph (CFG) edge coverage to guide the fuzzing process, which has shown great potential in finding vulnerabilities. However, CFG edge coverage is not effective in discovering vulnerabilities such as use-after-free (UaF). This is because, to trigger UaF vulnerabilities, one needs not only to cover individual edges, but also to traverse some (long) sequence of edges in a particular order, which is challenging for existing fuzzers.

29.2 Methods/Techniques

The authors propose to model UaF vulnerabilities as typestate properties, and develop a typestate-guided fuzzer (Fig.24), named UAFL, for discovering vulnerabilities violating typestate properties. Given a typestate property, they first perform a static typestate analysis to find operation sequences potentially violating the property. The fuzzing process is then guided by the operation sequences in order to progressively generate test cases triggering property violations. In addition, they also employ an information flow analysis to improve the efficiency of the fuzzing process.

29.3 Results/Evaluation

They performed a thorough evaluation of UAFL on 14 widely-used real-world programs. The experiment results show that UAFL substantially outperforms the state-of-the-art fuzzers, including AFL, AFLFast, FairFuzz, MOpt, Angora and QSYM, in terms of the time taken to discover vulnerabilities. They discovered 10 previously unknown vulnerabilities, and received 5 new CVEs.

29.4 Limitations/Comments

Due to the high expressiveness of typestate property, their approach can be extended to detect other types of vulnerabilities. For example, we can extend

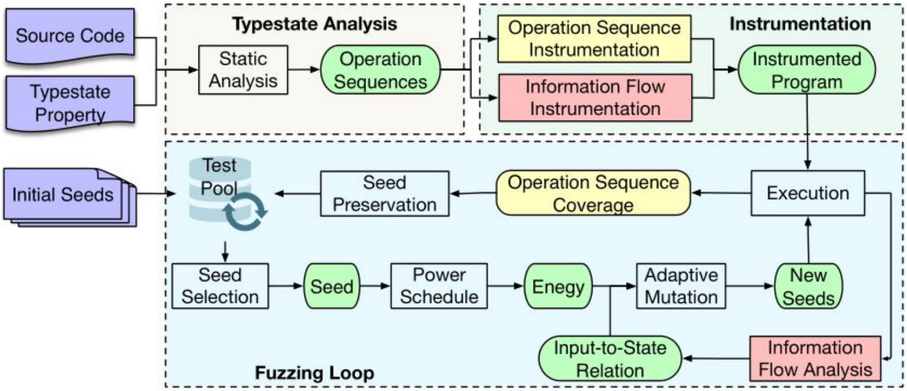


Figure 24: Workflow of UAFL.

UAFL to detect API-misuse [2], e.g., file read/write operations by customizing the typestate property.

30 Jetset: Targeted Firmware Rehosting for Embedded Systems @Security'21

30.1 Background/Problems

The ability to execute code in an emulator is a fundamental part of modern vulnerability testing. Unfortunately, this poses a challenge for many embedded systems, where firmware expects to interact with hardware devices specific to the target. Getting embedded system firmware to run outside its native environment, termed *rehosting*, requires emulating these hardware devices with enough accuracy to convince the firmware that it is executing on the target hardware. However, full fidelity emulation of target devices (which requires considerable engineering effort) may not be necessary to boot the firmware to a point of interest for an analyst (for example, a point where fuzzer input can be injected). We hypothesized that, for the firmware to boot successfully, it is sufficient to emulate only the behavior expected by the firmware, and that this behavior could be inferred automatically.

To test this hypothesis, we developed and implemented Jetset, a system that uses symbolic execution to infer what behavior firmware expects from a target device. Jetset can generate devices models for hardware peripherals in C, allowing an analyst to boot the firmware in an emulator (e.g., QEMU).

30.2 Methods/Techniques

Jetset requires the following information about the target embedded system.

- The executable code of the target, usually read out of program flash or extracted from a firmware update provided by a manufacturer.
- The memory layout of the target, specifying which regions of the address space are mapped to program memory, RAM, and device I/O registers. This information can be obtained from the datasheet of a single-chip system or from a basic analysis of the executable code. Note that Jetset does not need to know which devices are mapped where, only the address range used for MMIO.

- The entry point address where execution begins. This is often specified in the CPU datasheet.

- The program goal address that the analyst wants the program to reach. For example, this can be the address of a print instruction that reports a successful system boot.

There are two stages of Jetset operation: peripheral inference and peripheral synthesis. In the inference stage, Jetset uses symbolic execution to infer expected device behavior. Then, in the synthesis stage, the output of the inference stage is used to create a device suitable for use in an emulator (e.g., QEMU).

To mitigate the states explosion, Jetset exploits guided symbolic execution using a search strategy based on incremental control-flow graph construction. Jetset periodically injects interrupts during symbolic execution, so that each ISR is executed periodically during each execution path.

There are two key differences between P2IM's fuzzing based approach (Paper: 24), and Jetset's directed symbolic execution based approach. The first difference is that unlike P2IM, Jetset is targeted—it is designed to ignore most paths through the firmware to focus on a particular target piece of code, which allows it boot deep into large pieces of firmware. While Feng et al. showed P2IM's approach is effective at fuzzing peripheral handling code and emulating microcontroller code, it is not clear whether it scales to larger firmware. The second difference is that, while fuzzing-based approaches are efficient since they use lightweight executions, they can have trouble bypassing complex checks. Jetset is able to handle complex numerical checks, because it performs partial rehosting using symbolic execution.

30.3 Results/Evaluation

We successfully applied Jetset to 13 distinct pieces of firmware together representing 3 architectures, 3 application domains (power grid, avionics, and consumer electronics), and 5 different operating systems. We also demonstrate how Jetset-assisted rehosting facilitates fuzzing, a common security analysis technique, on an avionics embedded system, in which we found a previously unknown privilege escalation vulnerability. Available at: `https://jetset.`

aerosec.org.

30.4 Limitations/Comments

Path correctness. Jetset has no knowledge of the underlying hardware other than the behavior that is observable to the CPU. The path taken through firmware is not necessarily one that may ever be returned by the hardware; however, the path taken is one that is acceptable to the firmware—no interaction with any of the peripherals results in a boot failure. While the execution of firmware running on physical hardware is constrained in its behavior by how the physical peripherals really behave, these system constraints are external to the firmware and cannot be inferred without auxiliary information about its behavior.

Limited peripheral model. Jetset does targeted rehosting in that it only constructs an emulator that is sufficient to emulate the software component-under-test. If the firmware reads from a peripheral's address after reaching the target, Jetset replays the last satisfying value read from that address. In our tests, we found that this simple model is sufficient to perform useful analysis and bug finding (as shown in Section 6); however, more complex interactions with the peripherals may not be emulated correctly. Another limitation of Jetset's peripheral model is that Jetset has no understanding of the semantics of the devices synthesized besides what is needed to guide the firmware towards the target address. We found that our limited peripheral model caused the firmware to crash after reaching the target address in one case. In the BeagleBoard-xM, we found that our emulator attempted to execute data loaded from a serial boot from our synthetic device. Jetset had no method to detect that the data it is returning from device reads should be valid ARM code, and crashed because of it. In future work, we plan to have Jetset synthesize more complex, stateful peripheral models as well as identify known peripherals with existing emulator implementations.

No DMA support. Jetset does not support devices that perform direct memory access (DMA) to normal RAM. This is because DMA is not observable by firmware since the device accesses memory without the assistance of the CPU. In the two cases that DMA was required to boot the firmware-under-test, we either left the DMA device in the QEMU model (as described in 6.2) or manually marked the DMA region symbolic (as in the Robot firmware in 6.6). We leave automated modeling of DMA to future work.