Fuzzing with Taint Tracing Guided Feedback

Stephen Tong

***Abstract*—Software fuzzing is an emerging technique to find bugs in software. The effectiveness of fuzzing is extremely dependent on the fuzzers' ability to generate "interesting" inputs which elicit novel or unexpected behavior from the fuzzed application. The current state-of-the-art fuzzers use coverage-based feedback to aid them. However, coverage-based feedback is limited because it provides a very limited model of the program's behavior.**

In this work, we propose and explore a new method of collecting program feedback based on taint tracing. Our system, TL-FUZZ, is able to track which parts of the input affect control flow throughout the entirety of the program's lifetime. It does so by tracking of which parts of the input were used to reach the current program state. Input bytes that affect a conditional branch are considered as promising candidates for fuzzing, and recommended to the fuzzer. To accomplish this, we used a custom instruction set architecture suitable for taint tracing, and a compiler back-end and emulator to support our custom architecture.

I. Introduction

The most successful fuzzers leverage feedback-guided fuzzing in order to generate new inputs effectively. Under feedback-guided fuzzing, the fuzzer observes the program's behavior as it executes in order to guess or deduce what inputs may trigger interesting behavior from the fuzzed program [12]. Feedback-guided fuzzers operate in a loop, similar to genetic algorithms. At each iteration, a pool of candidate inputs are fed into the target program, and the most promising ones are selected for further fuzzing. These inputs are then mutated to create a new pool of candidates for the next round of fuzzing. As this process continues, the fuzzer "learns" to approximate good inputs, and the inputs become increasingly sophisticated.

The primary form of feedback used by fuzzers is code coverage. Under coverage-guided fuzzing, fuzzers aim to maximize the amount of code coverage explored by the input pool. Inputs which expand the coverage map are selected for further rounds. On the other hand, inputs which only reach code paths that have already been visited are discarded. In theory, the fuzzer's mutations should maximize the amount of code explored in the target program, hopefully including any code containing bugs, edge cases, etc. However, popular fuzzers like AFL [12] often fail to do so and eventually get stuck. One of the main reasons this happens is due to large inputs. Faced with a large input, the fuzzer will waste a lot of time trying to mutate every part of the input before moving on. Brute force is not an option: coverage-guided fuzzers cannot tell which parts of the input are significant in controlling the program's behavior. Thus, it is crucial to keep input sizes small when fuzzing; the AFL user manual even advises users to do so [11]. However, it is oftentimes necessary to fuzz large inputs. For example, filesystem images are usually large (in the range of kilbytes), which makes them unamenable for fuzzing. In our work, we seek to address this problem by using taint tracing to automatically discover the interesting parts of the input and provide hints to guide the fuzzer towards these areas.

II. Background

Taint tracing is a common method used in program analysis to discover which parts of a program a certain piece of data (typically the input) reaches [2]. Specifically, we taint, or mark, the data, and trace how the program uses the data. At any point in the program, if a calculation uses tainted data, then the result is also marked as tainted. Thus, the taint propagates throughout the program as it executes, and records what data was used to reach a given program state.

One common method for taint tracing is to add instrumentation to code. The instrumentation is extra code which keeps track of the taint-related metadata at each step in the program. However, this adds additional complexity as the taint tracing code must be interleaved inline with the original program code. Common architectures like x86 and ARM are also not amenable to taint tracing because their instruction sets are very complex. Thus, such methods tend to be relatively coarse-grained. On the other hand, we seek an extremely fine-grained, instruction-level trace of the fuzzed program. We tackle this by creating a custom RISC ISA, TL-45, which is as simple as possible, to make modeling instruction data flow very easy. We incorporated our fuzzer and all of the tools surrounding TL-45 into one system, TL-FUZZ.

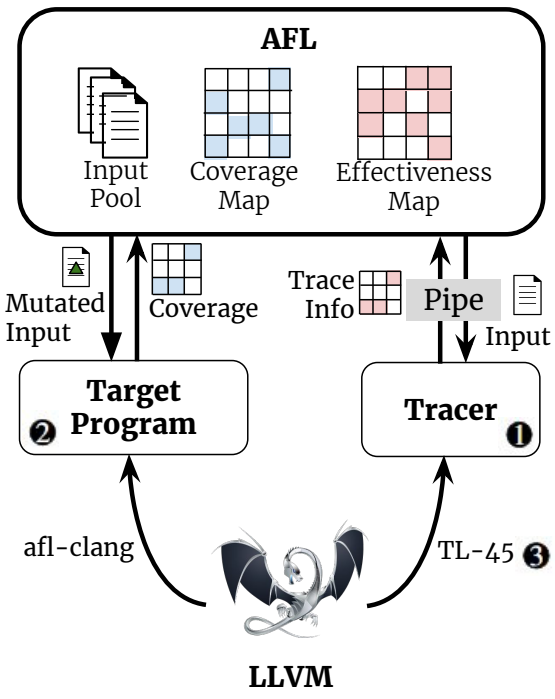

Fig. 1: The design of TL-FUZZ. LLVM is used to instrument the program for AFL, and to cross-compile to TL-45. The tracer runs the cross-compiled binary to collect effectiveness info, while AFL fuzzes the instrumented program to college coverage info.

III. Design

TL-FUZZ is designed around our custom ISA, TL-45. Figure 1 describes our system. TL-FUZZ consists of three parts: ❶ an emulator and taint tracer for TL-45, ❷ a custom AFL-based fuzzer which can use the taint data, and ❸ a custom LLVM [5] back-end and libc implementation for the TL-45 architecture to cross-compile target programs to. We will discuss each one in the following sections.

A. Taint tracer

The taint tracer is an emulator for TL-45 which also implements instruction-level taint tracing functionality. It runs code compiled to the TL-45 architecture with a provided input. The taint tracer then outputs a profile for which bytes of the input affected conditional branches. It tracks the hit counts of each conditional branch, which input bytes tainted it, and whether the branch was taken or not. This data is then used to assist the fuzzer in making more effective mutations.

B. Taint-aware fuzzer

We modified AFL to make it aware of the taint information generated by the tracer. Specifically, we integrated the taint data into AFL's map effectiveness which tracks which input bytes affect the program's control flow. AFL crudely approximates this by flipping bits at each byte of the input and monitoring if new code coverage is discovered. We augmented this with our concrete trace data. Because our trace data significantly refines the effectiveness map's accuracy, we also modified several mutation stages to rely more on the effectiveness map. For example, in the havoc stage which mutates random parts of the inputs, we preferentially modify bytes suggested by the tracer.

C. TL-45 compiler infrastructure

We implemented a custom LLVM back-end to support compiling arbitrary C and programs to the TL-45 ISA. Thanks to LLVM, the outputted code is reasonably efficient and well-optimized. This is good because smaller code is more amenable to taint tracing. Because the tracer emulates programs *bare-metal*, i.e., without an operating system, we also had to implement a bare-bones libc standard library. This allows us to compile ordinary C programs to run on our tracer.

IV. Implementation

In this section, we will describe our implementation of TL-FUZZ. The taint tracer consists of 1675 lines of C++ code, and the total modifications to AFL consisted about 100 lines of C code. For efficiency, the emulator uses virtual memory and lazy paging to simulate TL-45's 32-bit address space. We also considered JITing the TL-45 code for speed, but this would not be compatible with the taint tracing. We implemented a simple GUI front-end in Qt and a CLI for the tracer. Figure 2 is a sample output of the tracer emulating a JSON parser [3], and Figure 3 shows the code that was traced.

We also traced a few other simple programs, like a PNG parser [1].

V. Evaluation

In this section, we evaluate the effectiveness of TL-FUZZ in terms of how well it improves fuzzing performance. Turns out it doesn't work, sorry.

VI. Discussion

In the discussion section, you are usually supposed to apologize for your messy evaluation, but I don't think that applies here.

```
Input: {"memes" 129, "yeet": 99}
7904 bytes code loaded.
25 bytes input loaded.
Branch report:
00000f6c:f |  1 hits, tainted by: 14
00000f8c:f |  1 hits, tainted by: 15
000018d4:t |  1 hits, tainted by: 13
00000f18:f |  1 hits, tainted by: 1
0000084c:f |  1 hits, tainted by: 11 10 12
00000880:t |  1 hits, tainted by: 23 24
00001ed8:f |  1 hits, tainted by: 2
00001c18:f |  2 hits, tainted by: 20 7
00000ca0:t |  2 hits, tainted by: 23 24 11 10 12
000009c8:f |  2 hits, tainted by: 23 10
00000c18:f |  2 hits, tainted by: 25 13
00000fbc:t |  3 hits, tainted by: 25 15 1
0000190c:f |  5 hits, tainted by: 25
00000e8c:t |  5 hits, tainted by: 0
00000c30:f |  5 hits, tainted by: 24 23 12 11 10
00001434:f |  5 hits, tainted by: 25 24 13 12 11
00001c18:t |  9 hits, tainted by: 19 18 17 16 6 5 4 3 2
0000110c:t | 10 hits, tainted by: 21 8
000011b0:t | 10 hits, tainted by: 23 10
00001bf8:f | 11 hits, tainted by: 20 19 18 17 16 7 6 5 4 3 2
```

Fig. 2: An example run of TL-FUZZ on the code shown in Figure 3. On line 9 and 10, the tracer is able to deduce that the input bytes corresponding to the integer values 129 and 99 are individually checked in conditional branches at 0x84c and 0x880. This corresponds to lines 5 and 8 in Figure 3.

```
1  json_t mem[32];
2  json_t const* json = json_create(str, mem, 32);
3  if (!json) return 0;
4  int value = search(json, "memes");
5  if (value != 129) {
6      return 0xbad1;
7  }
8  if (search(json, "yeet") != 99) {
9      return 0xbad2;
10 }
11 return 0x1337;
```

Fig. 3: Code that was traced in Figure 2. The code parses input JSON data, looks up two values, and compares them.

We will conclude with few lessons we learned from this project:

- Creating custom architecture is really hard. You really have to do all of the legwork: compiler toolchain, libc, emulator, and debugger. Instead, just use RISC-V [9] next time.
- Taint tracing is very strong and can deduce high-level behavior that is otherwise opaque to coverage-guided or even symbolic execution-based tools.
- Just because you thought it was a good idea, does not mean it will work.

VII. Related work

QSYM [10] uses symbolic execution to get past difficult conditional branches, using partial constraint solving. They combine symbolic execution with concrete values; this technique is called *concolic* or *hybrid fuzzing*. I am a big fan of this one, mainly because it works.

Several other works have considered using neural networks for fuzzing. Some of them follow the same approach we do, where we try to hint the fuzzer to preferentially mutate certain parts of the input [7], while others try to use neural networks to generate [4] or classify [8] inputs. Turns out this doesn't really work either.

Also, a bunch of other people have already done taint-tracing with fuzzing [2] [6].

VIII. Future Work

Another benefit of using a custom ISA is that it unlocks the potential for *hardware-assisted taint tracing*. Under hardware-assisted fuzzing, the processor performs the taint-tracing functionality automatically with no overhead. The taint tracing algorithm described in this work can be easily approximated in hardware with LRU caches for each taint cell (i.e., register, memory) TL-45 already has an efficient Verilog hardware implementation; we leave adding taint tracing to this implementation as future work.

Another possibility for future work is to use the taint tracing data to augment a constraint solver for concolic fuzzing. This is promising as the concrete data flow information may alleviate the path explosion problem.

IX. Conclusion

In this work, we proposed TL-FUZZ, an instruction-granularity taint tracing system aimed to improve fuzzers. We implemented this with a custom ISA and an emulator and compiler for the ISA. We integrated the taint tracer into AFL, and found that unfortunately it doesn't actually make AFL much faster. However, the taint tracer does work and is pretty cool.

X. Acknowledgment

I would like to thank my advisor, Taesoo Kim, for his kind and thoughtful guidance throughout my undergraduate research, and my helpful and friendly colleagues.

This work was supported by Georgia Tech's PURA. Thanks to the award committee for sponsoring this work.

References

[1] elanthis. upng: very small png decoding library. https://github.com/elanthis/upng, 2010.

[2] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[3] R. Garcia. tiny-json: versatile and easy to use json parser in c suitable for embedded systems. https://github.com/rafagafe/tiny-json, 2018.

[4] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[5] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[6] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. Lippmann. Coverage maximization using dynamic taint tracing. Technical report, MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, 2007.

[7] M. Rajpal, W. Blum, and R. Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[8] Y. Wang, Z. Wu, Q. Wei, and Q. Wang. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.

[9] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

[10] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[11] M. Zalewski. american fuzzy lop (2.52b) - config.h. https://github.com/mirrorer/afl/blob/master/docs/perf_tips.txt, 2017.

[12] M. Zalewski. american fuzzy lop (2.52b). http://lcamtuf.coredump.cx/afl, 2018.