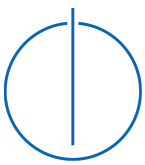
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Enhancing Software Fuzzing through Dynamic Instrumentation

Cosmin Banica



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Enhancing Software Fuzzing through Dynamic Instrumentation

Verbesserung des Software-Fuzzings durch dynamische Instrumentierung

Author:	Cosmin Banica
Examiner:	Supervisor
Supervisor:	Advisor
Submission Date:	15.01.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.01.2024 Cosmin Banica

Acknowledgments

Abstract

Contents

1 Introduction

1.1 Overview

Fuzzing [Man+19] has become a popular method of finding software vulnerabilities. At its core, it is a technique that generates random inputs for a target program, in the hope of triggering a crash. The idea is to work back from the crash to identify the root cause, which could be a dangerous bug, such as a memory corruption issue. The main advantage of fuzzing is that it is a fully automated process, and it can be run on a large scale, without requiring human intervention for extended periods of time.

One of the more interesting aspects of fuzzing is that it can be used to find bugs in closed-source software, where the source code is not available. There are many technologies that can be leveraged to achieve this, and in this thesis we will focus on QEMU [Bel05], a popular open-source emulator that has a number of features that make it suitable for fuzzing, and which has also been used in different fuzzers. One of the downsides of binary-only fuzzing is that it is difficult to get detailed information about the crash, such as the exact location of the bug. A common solution is to use some form of address sanitization.

Address sanitization [Ser+12] is a technique that adds extra checks to the program, in order to detect memory corruption issues. It is usually a good idea to use address sanitization in combination with fuzzing, but it does come with a performance penalty. In this thesis, we will explore the possibility of dynamically adding address sanitization to parts of the target program, in order to reduce the performance overhead. We will look at different implementations of this method, and evaluate their effectiveness, as well as determining which heuristics for dynamic instrumentation are most effective in practice.

1.2 Relevant concepts

For the purpose of this thesis, it is important to define some key concepts that will be used throughout. These include fuzzing, QEMU, and address sanitization. In the following section all of these concepts will be explored in more detail, as well as why they are relevant to the topic of this thesis.

1.2.1 Fuzzing

Fuzzing was first properly described in 1990 [MFS90], and has since become a popular method of finding software vulnerabilities. The basic idea is to generate pseudo-random inputs, with which to run the target program, with the goal of triggering a crash. The person running the fuzzer can then analyze the crash to determine the cause. It can be used by attackers to find vectors of attack in software, or by developers, to find and fix bugs. A short overview of the fuzzing process can be seen in Figure 1.1.

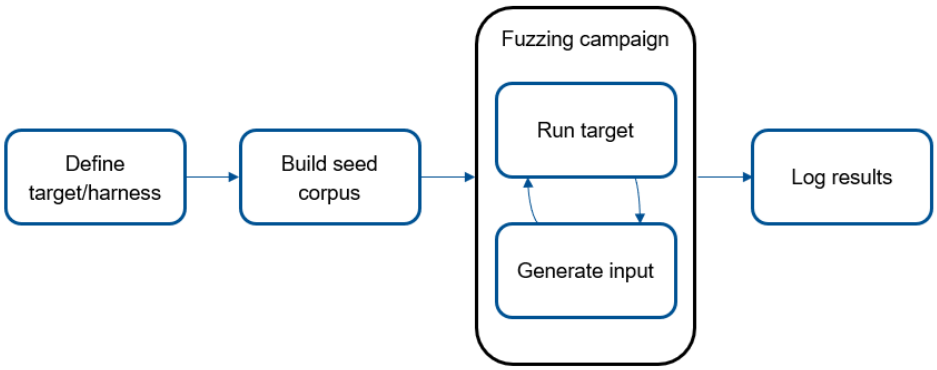


Figure 1.1: Base steps for fuzzing

No matter what fuzzer is used, there are a couple of things that are needed to start any fuzzing campaign. Firstly, the user needs to provide a seed corpus, which is a base set of inputs that the fuzzer will use to generate new inputs. Secondly, a target program needs to be provided. This may sound simple, but in practice the target will not always have a friendly entry point, or one that directly accepts input. For instance, the target may be a library, which will not commonly have a main function. Here, the user first needs to write a small program that will call some desired parts of the target's code, and then pass the input to it. This is known as a harness.

Fuzzing can be broken down into two main categories, based on the type of target program that is being fuzzed [God07]. If the target is an open-source program, then the fuzzer is commonly referred to as a white-box fuzzer. In this case, the fuzzer has access to the source code, and it can use all inherent information to aid in the fuzzing process. For example, the target can be compiled with instrumentation, a technique that adds extra flags to the code. These can be used to detect memory corruption, or to provide code coverage information to the fuzzer. Furthermore, the code coverage can then be used by the fuzzer, to guide the generation of new inputs that are more likely to reach unexplored code regions.

If the target is a binary, or a closed-source program, then the fuzzer is commonly

referred to as a black-box fuzzer. In this case, the fuzzer doesn't have access to the source code, and so it can't alter the target program at compile-time. This means that the fuzzer has to have a way of dynamically interpreting the instructions of the target. A common way to do this is to use an emulator, which can run the target in a virtual environment. This can be taken one step further, with the fuzzer dynamically adding instrumentation [Che+18] after interpreting the source code during execution.

The focus of this thesis is on black-box fuzzing. Since binary fuzzing is more difficult than white-box fuzzing, as it loses some of the directedness that comes from having access to the source code, and being able to compile the target with instrumentation, it is important to remedy this with other approaches. In the following, we will look at how address sanitization can be used to improve the effectiveness of binary fuzzing, and how QEMU can be integrated with this sanitization.

1.2.2 QEMU

QEMU [Bel05] is a machine emulator and virtualizer that can emulate several architectures, on a variety of hosts. This ability to monitor and control execution in a sandboxed environment, with many different architectures, makes it a popular choice for fuzzing. For this use case, QEMU is used to run the target application, and it provides the fuzzer with the ability to dynamically instrument the executed code. Thanks to QEMU being an open-source project, this makes it easy for fuzzer developers to create their own custom versions of QEMU, tailored to their specific implementation of a fuzzer [Fio+20]. A representation of how QEMU usually interacts with fuzzers can be seen in Figure 1.2.

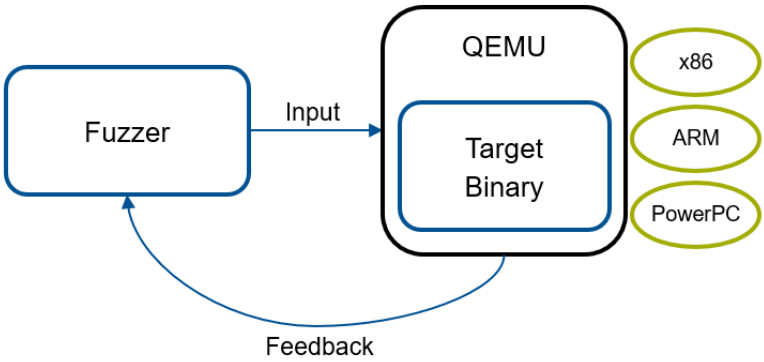


Figure 1.2: QEMU interaction with fuzzers

While QEMU is not the only tool of this kind that can be used for fuzzing, there are a number of features that make it particularly suitable. Valgrind [NS07] and Pin [Red+04] are two other popular tools that offer dynamic binary instrumentation, but they are less

versatile in terms of the architectures that they support. Moreover, QEMU also allows for full-system emulation, which presents the interesting possibility of full-system fuzzing [MFF24]. Another extremely useful feature for fuzzing, specifically in system mode, is the ability to save and restore the state of the target program, by using the snapshot capabilities of QEMU. While these snapshots are too slow to be used in a fuzzing loop, they can be used to restore the Virtual Machine (VM), without having to go through the slow boot phase.

One of the most useful consequences of being able to instrument code, is that it allows for the addition of address sanitization. This is a technique that adds extra checks to the program, in order to detect memory corruption issues. Address sanitization comes with a considerable performance penalty, but it is usually a good idea to use it when fuzzing. In the following section, we will look at how address sanitization can be used to improve the effectiveness of fuzzing, and discuss how Address Sanitizer (ASAN) and QEMU Address Sanitizer (QASAN) can be used in combination with QEMU.

1.2.3 ASAN/QASAN

ASAN [Ser+12] is a memory error detector that can be used to find memory corruption issues in programs. It was developed by Google, as part of their family of sanitizers [SS15] [SI09], and it is now widely used in the industry, also being integrated into LLVM.

There are two main components to ASAN. The first is the shadow memory. This is a representation of the target program's memory, where each byte is used to signal whether the 8 corresponding bytes in the actual memory are addressable or not. A visual representation of this can be seen in Figure 1.3.

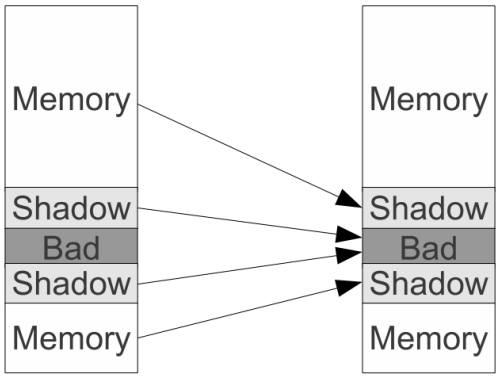


Figure 1.3: ASAN memory mapping [Ser+12]

The second main component of ASAN is the instrumentation. This is the part that

actually adds the checks to the target. Every time a memory access is made, the instrumentation will query the shadow memory to see if the bytes that are being accessed are valid or not. In order for the shadow memory to keep a track of whether or not a part of the memory is addressable, there is also instrumentation added to the allocation and deallocation functions. This way, the shadow memory is updated whenever memory is allocated or freed, and queried whenever a write or read is called. One other important aspect of ASAN is that it can also detect out-of-bounds accesses around stack and global objects. This is done by adding a poisoned 'redzone' around these objects, which trigger a crash if they are accessed.

QASAN [FDQ20] is an extension of ASAN, that is specifically designed to work with QEMU, and to be used in a fuzzing framework, for black-box fuzzing. The main idea behind QASAN is to dynamically add address sanitization to the target program, by using QEMU to interpret the instructions of the target. The Tiny Code Generator (TCG) feature of QEMU is used to translate the target's instructions into a form that can be executed on the host. This translation can be leveraged to add additional instructions, that will perform the ASAN checks. A representation of the QASAN architecture can be seen in Figure 1.4.

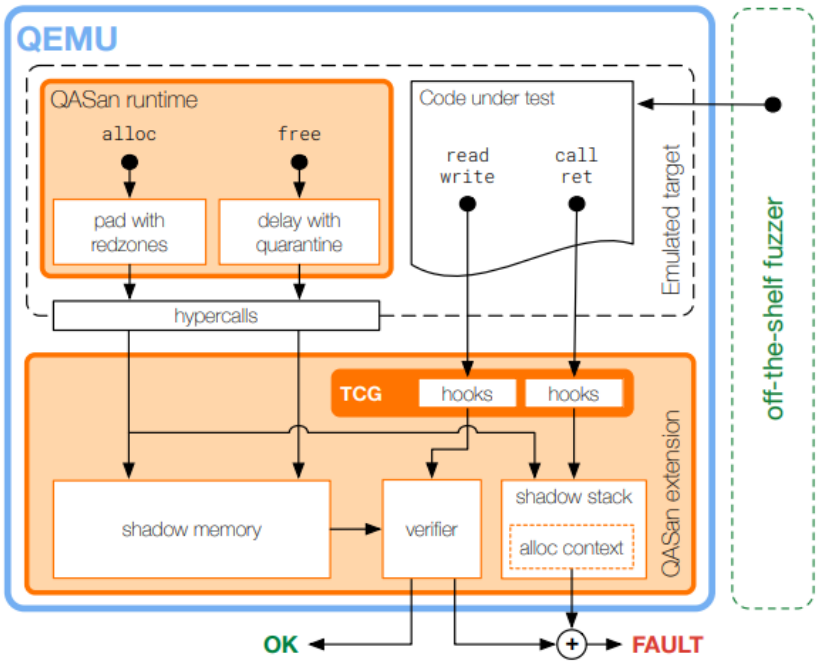


Figure 1.4: QASAN architecture [FDQ20]

1.3 Motivation

In software security, memory corruption bugs are a constant source of vulnerabilities. These bugs include buffer overflows, use-after-free, double-free, and can be exploited by malicious actors, as was the case with the Heartbleed bug [Dur+14]. Despite the widespread use of many different tools and techniques to find these bugs, such as address sanitization and fuzzing, they still remain a problem. Often times it was the wider coding community that found these bugs, and not the developers themselves. One of the largest obstacles in enabling the community to discover more bugs, is that in a lot of cases, the source code is not available, only the binary.

Here, it is important to note that black-box fuzzing is not necessarily slower than white-box fuzzing [Che+23], but it is more difficult to get the same level of coverage, and so it will be slower overall in finding bugs. Without the ability to compile the target with instrumentation, the fuzzer has much less information at its disposal with which to generate better inputs. As a result, in the case of binary-only fuzzing, it is much more important to increase the speed of the fuzzing campaign, than in standard white-box fuzzing.

As previously discussed, ASAN\QASAN comes with a significant performance penalty. The additional bookkeeping that is required to maintain the shadow memory, and the extra checks that are added to the target, slow down the fuzzer considerably. However, regardless of whether the fuzzer is running in white-box or black-box mode, it is usually a good practice to use address sanitization in the fuzzing campaigns. Using ASAN for fuzzing can reveal bugs that would otherwise go unnoticed [MFF24], and it can also provide the fuzzer with better information about the crash.

In this thesis, the focus will be on improving the performance of binary-only fuzzing with address sanitization. The goal is not simply to reduce the performance overhead of QASAN, but to do so in a way that does not negatively impact the fuzzing campaigns, by reducing the bugs that are found. The main idea is to give the fuzzer the power to decide dynamically which parts of the code should be instrumented with sanitization, and which parts shouldn't. This can be done by using different heuristics and settings, that will be evaluated, to determine which set of these would be the most effective in practice.

2 State of the art

Despite having been around for a few decades, fuzzing has begun to gain a lot more popularity in recent years [Lia+18]. While the idea of fuzzing may appear simple at its core, there have been so many advancements that have enabled popular fuzzing frameworks to implement some very interesting optimizations. Some of the main areas that these frameworks often focus on improving are the following:

- **Input generation**: This component is responsible for generating the input corpus that will be used to run the target application. The most basic form of input generation is to simply create random inputs. The next step in terms of complexity, is to use more complex algorithm-based approaches, such as bit or byte flipping. Finally, most performant fuzzers also take into account the structure of the input data, as well as leverage the knowledge provided by coverage data of the target.
- **Corpus management**: Any fuzzing campaign begins with an initial set of seed inputs, generally provided by the user. As the campaign progresses, and the fuzzer generates additional inputs, the corpus grows. This is where corpus management comes into play, as it falls to the fuzzer to decide which inputs to keep in the corpus, and which to discard. This is necessary in order to avoid wasting resources on inputs that are not likely to trigger new code paths, since otherwise the corpus would grow to huge sizes. Common heuristics of determining which inputs to keep are based on coverage data, or on the performance of the target application when processing the input, in terms of crashes, hangs, execution speed, etc.
- **Coverage data**: Keeping track of the code coverage achieved by target runs is a key component of any modern fuzzer. Firstly, this info can provide the user with relevant statistics about the campaign. Secondly, coverage info is perhaps the most important factor in determining the quality of the inputs generated by the fuzzer. Coverage data can relate to basic block coverage, edge coverage, function coverage, etc. The goal of the fuzzer is to have reliable coverage tracking, while also keeping the overhead of this tracking to a minimum.
- **Instrumentation**: In order to collect coverage data, and to also add additional features like ASAN, the fuzzer needs to add some bits of code to the target,

which perform the necessary operations. This is done either at compile time, by using special compilers [ZDZ19], or at runtime, by using dynamic binary instrumentation [CSL04].

In the following sections, an overview of some of the most popular fuzzing frameworks, AFL++ and LibAFL, will be provided. These two frameworks have been chosen because they are well integrated with QEMU, and have solid implementations for the QASAN feature for binary-only fuzzing.

2.1 AFL++

AFL++ is a fork of the original American Fuzzy Lop (AFL) coverage-guided fuzzer [Fio+20]. The main goal of AFL++ is to

2.2 LibAFL

2.3 Current shortcomings

3 Dynamic sanitization

3.1 Concept

3.2 Implementation

3.2.1 Block module

3.2.2 Edges module

3.2.3 Comparison of approaches

4 Evaluation

4.1 Setup

4.1.1 Hardware

4.1.2 Fuzzer used

4.1.3 Target programs

4.1.4 Run scripts

4.2 Metrics

4.2.1 Bugs detected

4.2.2 Execution speed

5 Future work

5.1 Fully in-memory bookkeeping

5.2 Dynamic parameter mutation

6 Conclusion

Abbreviations

VM Virtual Machine

ASAN Address Sanitizer

QASAN QEMU Address Sanitizer

TCG Tiny Code Generator

AFL American Fuzzy Lop

List of Figures

List of Tables

Bibliography

[Bel05] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005.

[Che+18] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu. "A systematic review of fuzzing techniques." In: *Computers & Security* 75 (2018), pp. 118–137. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2018.02.002.

[Che+23] Y. Cheng, W. Chen, W. Fan, W. Huang, G. Yu, and W. Liu. "IoTFuzzBench: A Pragmatic Benchmarking Framework for Evaluating IoT Black-Box Protocol Fuzzers." In: *Electronics* 12.14 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12143010.

[CSL04] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. "Dynamic Instrumentation of Production Systems." In: *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. Boston, MA: USENIX Association, June 2004.

[Dur+14] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. "The Matter of Heartbleed." In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488. ISBN: 9781450332132. DOI: 10.1145/2663716.2663755.

[FDQ20] A. Fioraldi, D. C. D'Elia, and L. Querzoni. "Fuzzing Binaries for Memory Safety Errors with QASan." In: *2020 IEEE Secure Development (SecDev)*. 2020, pp. 23–30. DOI: 10.1109/SecDev45635.2020.00019.

[Fio+20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "AFL++ : Combining Incremental Steps of Fuzzing Research." In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[God07] P. Godefroid. "Random testing for security: blackbox vs. whitebox fuzzing." In: *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. RT '07. Atlanta, Georgia: Association for Computing Machinery, 2007, p. 1. ISBN: 9781595938817. DOI: 10.1145/1292414.1292416.

[Lia+18] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. "Fuzzing: State of the Art." In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218. DOI: 10.1109/TR.2018.2834476.

[Man+19] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2019. arXiv: 1812.00140 [cs.CR].

[MFF24] R. Malmain, A. Fioraldi, and A. Francillon. "LibAFL QEMU: A Library for Fuzzing-oriented Emulation." In: *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*. San Diego (CA), United States, Mar. 2024.

[MFS90] B. P. Miller, L. Fredriksen, and B. So. "An empirical study of the reliability of UNIX utilities." In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[NS07] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[Red+04] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. "PIN: a binary instrumentation tool for computer architecture research and education." In: *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*. WCAE '04. Munich, Germany: Association for Computing Machinery, 2004, 22–es. ISBN: 9781450347334. DOI: 10.1145/1275571.1275600.

[Ser+12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: a fast address sanity checker." In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[SI09] K. Serebryany and T. Iskhodzhanov. "ThreadSanitizer: data race detection in practice." In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. New York, New York, USA: Association for Computing Machinery, 2009, pp. 62–71. ISBN: 9781605587936. DOI: 10.1145/1791194.1791203.

[SS15] E. Stepanov and K. Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++." In: *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Francisco, CA, USA, 2015, pp. 46–55.

[ZDZ19] C. Zhang, W. Y. Dong, and Y. Zhu Ren. "INSTRCR: Lightweight instrumentation optimization based on coverage-guided fuzz testing." In: *2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET)*. 2019, pp. 74–78. DOI: 10.1109/CCET48361.2019.8989335.