A Comparison of Fuzzing Dynamic Analysis and Static Code Analysis

CCDC-AvMC

Authors

SEAN ALEXANDER

DR. JONATHAN HOOD

ERICA JONES

November 16, 2020

1 Abstract

The purpose of this research project is to determine the effectiveness of static code scan analysis compared to fuzzing as a form of dynamic binary analysis.

Fuzzing is becoming a more relevant approach to dynamic code analysis for software quality. There are many open source fuzzing engines available on the internet. The basic premise of fuzzing is to programmatically contrive and apply semi-random test cases to the inputs of a piece of software. Any bugs or flaws in the codebase are detected by the fuzzing engine and recorded. The failure test case can be used to find and fix vulnerable code which may have not taken into account the identified parameters from the fuzzing engine.

Static code analysis is a process where the source code itself is scanned. The static analysis software scans the lines of code and compiles a list of potential flaws in a report. The report is then used to allow the developers to review the findings and apply any fixes that are needed.

The initial expectation is that a properly implemented static code scan analysis would find any bugs that would be caught by a fuzzer. To test this hypothesis, code bases with known flaws found during fuzzing are subjected to static code scan analysis. The results found that a majority of the fuzzing flaws are not directly identified by static code analysis.

2 Introduction

Fuzzing, as an approach to performing dynamic analysis for software quality, is becoming a more popular and robust area of study. The process is sometimes characterized as passing “garbage” through the software until something breaks. This is actually a specific form of

fuzzing known as random black-box fuzzing, but there are other variations on this theme[1]. The fuzzing engine starts with a well-formed seed case and mutates it in a semi-random manner, often using Artificial Intelligence, Machine Learning, and Genetic Algorithms. The mutated input is then injected into the code being tested. The process repeats as many times as possible to increase coverage and the odds of finding a flaw. While this is an effective technique, it can be very time intensive to find failure test cases[1].

Grammar-based fuzzing is an attempt to mitigate the inefficiencies of the more random approach of black-black box. There are cases when an input is expected to be properly formatted. An example of this would be JSON, XML, or some other standard, structured format. Some fuzzers, like American Fuzzy Lop (AFL), implement a dictionary that maintains the structure of the inputs. This allows for deeper testing in the binaries by preventing test cases failing a properly validated input[2].

The previously mentioned approaches both suffer from efficiency and limited feedback issues. White-box fuzzing attempts to address these problems by instrumenting the code so it can be monitored at runtime. When the initial seed is first used the fuzzing engine maps the execution path through the binary. As the mutated inputs are passed, the engine can more intelligently map the changes in the inputs with new paths. There is an additional hope that with this approach, it can be determined that the code has been sufficiently tested so that no further testing would be necessary[3].

The OSS-Fuzz Google service for open source software is an initiative that continually fuzz tests open source products. Since coming online in 2016, OSS-Fuzz has found flaws in some open source packages that are used in defense products. OSS-Fuzz also regularly runs multiple fuzzing engines with a report comparing results[4][5].

One of the biggest advantages of fuzzing is that there are no false positive test cases. By the very nature of the analysis method, the code is exercised with the test case as failures occur. A finding indicates either a failure in the software or a failure in the harness structure.

This makes the test cases very valuable for developers to maintain and improve the code.

The fuzzing approach is not without problems. Do to the semi-random nature of its operation, it can take months to find a problem. One of the test cases used for the research in this paper requires 9 months of fuzzing and 5 CPU years to identify. It's also worth mentioning that within the five-month fuzzing with OSS-Fuzz, 264 vulnerable inputs across 47 open source projects have been identified. In our analysis, we selected a subset of 21 open source projects which result in 45 unique findings from fuzzing. The hardware ran over ten trillion test inputs per day. Fuzzing works on the probability that if you push enough inputs into the program, you will find bugs[6].

Fuzzing requires the software being fuzzed to be harnessed. A harness is a program that sits in between the fuzzing engine and the product being tested. This program acts as an interface to inject the fuzzed input parameters into the program. The amount of effort required to create these harnesses are proportional to both the complexity of the program and the number of inputs of the program being harnessed. This also introduces an important danger: if the product is not properly harnessed, for example an input is missed, that the fuzzing effort may be incorrect or incomplete. If the fuzzer doesn't return coverage data, there may be no outward signs of insufficient test coverage[7].

In contrast, static code analysis (SCA) does not require modification or harnessing of the source code. SCA operates on the as-is, fielded source code. Many SCA engines, such as Fortify and Coverity, also benefit from building the code and generating control flow graphs of parameters to check boundaries and limits through a static, symbolic execution in addition to their code analysis engines. Checkmarx does not require buildable static code at the cost of potentially generating more false positive potential findings. While fuzzing reveals 45 confirmed vulnerable parameter input combinations, SCA using Coverity and Checkmarx identified thousands of potentially weak areas of code. The nuance in finding language here should not be overlooked: while fuzzing results in unique parameter combinations

that could reflect an issue in the same line of code, SCA results in unique potential findings against lines of code.

The other issue with SCA is the need to vet through the potential findings to triage them for correctness. False positives, or even heavily mitigated positives must be analyzed against the software's specific implementation. Fuzzing harnesses can be built to simulate and more accurately predict the mitigating environment when testing the software.

While the most thorough SCA requires buildable, complete code bases, harnesses for fuzzing can be written against exposed routines inside of binaries and libraries. SCA requires access to the source code that is being scanned while fuzzing requires, at a minimum, access to and understanding of the executable binaries being scanned. These executable binaries must still be harnessed, which is usually an easier task when access to the source code is available.

3 Procedure

A fresh VM was created using the CentOS 8 operating system. VMWare Tools was installed in order to provide the VM network drivers. Once internet access was achieved, the OS was updated. The full "Development Tools" repository was installed on the VM. Some of the projects in the OSS-Fuzz require clang, which required enabling the extended tools options.

The OSS-Fuzz git repository was cloned onto the VM. This project included 24 open source projects, of which, the 21 open source projects under consideration in this research were included. There are also included scripts to auto-build and run a local fuzzer engine for each project. These scripts were modified to not execute the fuzzing portion in order to optimize the build process. The modified scripts built the 21 projects under consideration. By viewing the logs, some projects required the installation of dependencies. One project could not be

built in an updated CentOS 8 environment and is excluded from consideration in this research.

The Coverity analysis tool was installed on the machine. Coverity was then configured for gcc, clang, and go. There was an attempt to modify the script to automate the process of running coverity on the projects, but there were issues that made it more efficient to run each Coverity analysis manually.

In each project, the following steps were done. A directory was created called "coverity", and the `emit-db` intermediate directory from Coverity was obtained. The build script for the project was executed causing the codebase with the known flaw to be downloaded and built. Once complete, the make command was called with "clean" to remove the binaries. The Coverity build tool was called with the parameters "make" and the path of the project intermediate directory. The build process populated the intermediate directory with the coverity analysis data.

While the Coverity process was being run, a VM with Checkmarx was run on another machine in parallel. The repositories were copied to the Checkmarx machine. The individual code bases were zipped and a Checkmarx scan was run on all of them. The results were put into CSV files for later comparison to the fuzzer failure information.

When the Coverity builds were complete, an analysis machine was started and the intermediate directories made available through a shared network directory. The intermediate directories were processed one at a time. Once all the analysis was run, they were committed to the Coverity platform so the results could be viewed and compared.

The projects were put into a workbook with crash information that was provided by the fuzzing repositories. In some cases, a full crash stack had been provided with file locations. In others, there were links to the bug information that were used to build a partial vulnerable parameter set.

The SCA and fuzzing results were combined and compared di-

rectly. If the static analysis finding matched the line and cause of the fuzzing-identified failure, it was listed as a direct finding. If there was no direct finding, a deeper search was done to see if the static findings could have related to the crash data. There were several instances where it was reasonable that the two could be related and these comparisons were labeled “possible finding”. If there were instances were the findings intersected the static but any causality between the two was questionable, the comparison was labeled “Unlikely Finding”. The rest were labeled as “No direct finding”. While some findings that fell into the last category did often intersect with the crash stack, the likelihood of being able to link them was improbable.

4 Results

Un-vetted Static Analysis High Risk Findings

Code Project	Coverity	Checkmarx
boringssl	184	977
c-ares	7	862
freetype2	42	140
guetzli	10	62
harfbuzz	23	256
json	17	0
lcms	54	0
libarchive	91	1540
libjpeg-turbo	47	197
libpng	19	554
libssh	53	495
libxml2	343	214
openssl-1.0.1f	475	3369
openssl-1.0.2d	465	3348
openssl-1.1.0c	384	3189
openthread	23	570
pcre2	16	117
proj4	30	499
re2	8	173
sqlite	66	11
vorbis	18	260
woff2	8	17
wpantund	18	230

table 1

Detection of fuzzer flaws by Static Code Analysis
D=Direct Finding
P=Possible Finding
U=Unlikely Finding
N=No Direct Finding

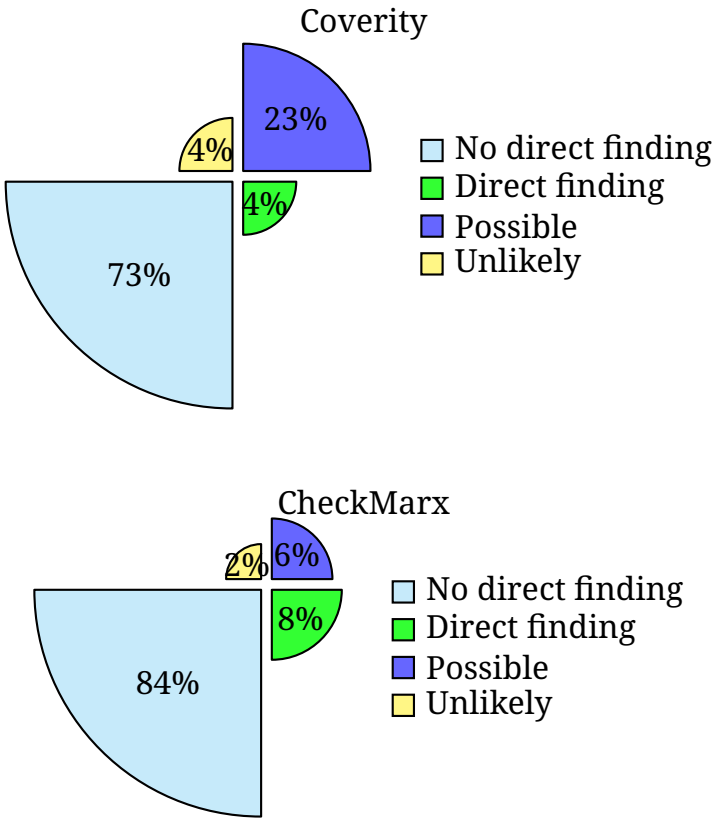
Code Project	Known Flaw	Coverity	Checkmarx
boringssl	AddressSanitizer: heap-use-after-free	N	N
c-ares	AddressSanitizer: heap-buffer-overflow	N	N
freetype2	integer-overflow	P	N
	assertion fail	P	N
guetzli	assertion fail	N	P
harfbuzz	AddressSanitizer: heap-buffer-overflow	N	P
json	assertion fail	N	N
lcms	AddressSanitizer: heap-buffer-overflow	N	N
libarchive	AddressSanitizer: heap-buffer-overflow	N	N
libpng	potential malloc failure	P	P
libssh	memory leak	N	N
libxml2	AddressSanitizer: heap-buffer-overflow	N	N
	memory leak	N	N
	AddressSanitizer: heap-buffer-overflow	P	N
openssl-1.0.1f	AddressSanitizer: heap-buffer-overflow	D	D
	memory leak	N	N
openssl-1.0.2d	assertion fail	N	N
openssl-1.1.0c	heap buffer overflow	N	N
	AddressSanitizer: heap-buffer-overflow	N	N
	malloc leak	P	U

table 2

Detection of fuzzer flaws by Static Code Analysis (cont)

openthread	Heap-buffer-overflow	N	N
	Stack-buffer-overflow	P	N
	Stack-buffer-overflow	N	N
	Stack-buffer-overflow	P	N
	Stack-buffer-overflow	N	N
	Stack-buffer-overflow	P	N
	Stack-buffer-overflow	N	N
	Stack-buffer-overflow	N	N
	Stack-buffer-overflow	N	N
	Stack-buffer-overflow	N	N
	Stack-buffer-overflow	N	N
	Null-dereference	P	N
pcre2	AddressSanitizer: heap-buffer-overflow	U	N
	AddressSanitizer: heap-buffer-overflow	U	N
proj4	LeakSanitizer: detected memory leaks	N	D
	LeakSanitizer: detected memory leaks	N	D
re2	DFA out of memory	P	N
	AddressSanitizer: heap-buffer-overflow	P	N
sqlite	Heap-use-after-free	N	N
	Heap-use-after-free	D	N
	Direct-leak	N	D
vorbis	AddressSanitizer: heap-buffer-overflow	N	N
	AddressSanitizer: heap-buffer-overflow	N	N
woff2	AddressSanitizer: heap-buffer-overflow	P	N
	libFuzzer: out-of-memory	P	N

table 2(cont)


Coverity
23%
4%
4%
73%
No direct finding
Direct finding
Possible
Unlikely
CheckMarx
2%
6%
8%
84%
No direct finding
Direct finding
Possible
Unlikely

5 Conclusion and Summary

Of the 45 failure cases found through fuzzing, neither static code analysis tool found more than ten percent of the failures directly. In many cases, there are findings that occur within the failure stack of the problem, but analysis suggests the flaws are unrelated [see table 2].

Across the projects, there are a number of high-risk findings. In a product environment, these findings would have to be vetted by an analyst to verify real findings from false positives [see table 1].

A majority of the issues that are not caught primarily involve overflows on the heap which are dynamically created and destroyed at run-time. It is worth noting that some of the flaws found through fuzzing are the result of assertion failures. This indicates that the failure case is known when the product was developed, and places the responsibility for proper error handling on the caller [see table 2].

Some of the fuzzing results are omitted from the final results. In one case the project `llvm-libcxxabi` couldn't be built due to conflicting dependencies with CentOS 8 updates. `wpantund` and `libjpeg-turbo` are coverage tests used to gauge the speed with which a given fuzzing engine can reach a certain line of code.

The purpose of this experiment is to discern if SCA can eliminate the need for fuzzing analysis. The results demonstrate that there are flaws in the software that are not discovered by the SCA tools employed. While the SCA reflects a higher quantity of potential findings, the overlap between SCA and fuzzing is between 5-30%. The results suggest that a two-prong approach of using both methods provides a more comprehensive software assurance analysis. The hypothesis that properly implemented SCA would find the issues identified by fuzzing libraries is disproved.

References

1. GODEFROID, P. Fuzzing: Hack, Art, and Science. *COMMUNICATIONS OF THE ACM.*
2. Pham, V.-T., Boehme, M., Santosa, A. E., Caciulescu, A. R. & Roychoudhury, A. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 1–1 (2019).
3. Patrice Godefroid Adam Kie ˙zun, M. Y. L. Grammar-based Whitebox Fuzzing. *PLDI '08: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.*
4. *A New Chapter for OSS-Fuzz* in (). `https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html`.
5. *04-03-2020 report* `https://www.fuzzbench.com/reports/2020-03-04/index.html`.
6. *OSS-Fuzz: Five months later, and rewarding projects* May 2017. `https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html`.
7. *Our guide to fuzzing* `https://www.f-secure.com/us-en/consulting/our-thinking/15-minute-guide-to-fuzzing`.