# Employing Different Program Analysis Methods to Study Bug Evolution

Charalambos Mitropoulos
Athens University of Economics and Business
Athens, Greece
charalambos.mitropoulos@gmail.com

## ABSTRACT

The evolution of software bugs has been a well-studied topic in software engineering. We used three different program analysis tools to examine the different versions of two popular sets of programming tools (GNU Binary and Core utilities), and check if their bugs increase or decrease over time. Each tool is based on a different approach, namely: static analysis, symbolic execution, and fuzzing. In this way we can observe potential differences on the kinds of bugs that each tool detects and examine their effectiveness. To do so, we have performed a qualitative analysis on the results. Overall, our results indicate that we cannot say if bugs either decrease or increase over time and that the tools identify different bug types based on the method they follow.

## CCS CONCEPTS

• **Software and its engineering → Empirical software validation**.

## KEYWORDS

Software Evolution, Symbolic Execution, Fuzzing, Static Analysis

**ACM Reference Format:**
Charalambos Mitropoulos. 2019. Employing Different Program Analysis Methods to Study Bug Evolution. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3338906.3342489

## 1 INTRODUCTION

The evolution of software bugs has been an enduring research topic [11, 13, 14, 20]. Security bugs are many times of particular interest [15, 16, 18] because they can introduce a potentially exploitable vulnerability into a software system. In the context of a bug evolution study, researchers usually employ program analysis [15, 18], or study software issues and corresponding bug fixes [4, 21], to check whether the bugs of a software component either increase or decrease over time. In this way they can validate whether programmers care for the risk posed by bugs when they release a new version of their software.

We have examined the evolution of bugs found in two well-known sets of libraries, namely: the GNU Binary Utilities [6] and GNU Core Utilities [7]. Contrary to other studies we utilize multiple well-established tools that implement different approaches to discover bugs. To do so, we have developed an automated method that given a program analysis tool and the repository of a software project, it analyzes all project versions with the tool and stores all results.

This provided us with the opportunity to not only check the evolution of bugs found in popular libraries, but also examine the different kinds of bugs each tool detects. In addition, we have performed a qualitative analysis on the results of each tool to observe the persistence of the bugs and identify corresponding CVE (Common Vulnerabilities and Exposure) entries.

## 2 METHODOLOGY

We provide details regarding the projects we selected to analyze, discuss our approach and the tools we employed.

**Selected Projects.** We have selected two sets of GNU programming tools. Specifically, we examined 13 versions of Binary Utilities ("*Binutils*"), from 2.11 to 2.32 (the most recent ones). We omitted versions from 2.15 to 2.21 because they were experimental ones and they did not include all utilities. Also, we analyzed the 7 latest versions of Core Utilities ("*Coreutils*") because the previous ones have been extensively examined before [9]. We left out versions 8.15 to 8.19 because they were incomplete, experimental releases.

**Approach.** We have developed a prototype that can be used to perform bug evolution studies with different program analysis tools. To do so, our prototype accepts two arguments: the name of the repository of the library that we want to analyze together with a version range, and the tool that we want to employ for our analysis. First, it downloads (currently from https://ftp.gnu.org/) a version and untars the file. After retrieving various statistics e.g. the lines of code (LOC) and the number of comments, it invokes the selected tool. Note that if the tool performs dynamic analysis (e.g. AFL), our prototype compiles the library. The steps above are repeated for each version included in the range. All results are stored in a JSON (JavaScript Object Notation) format.

**Tools Setup and Details.** In our study, we used (1) *Flawfinder* [22], a static analyzer [10], (2) KLEE [9], a symbolic execution tool, and (3) AFL (American Fuzzy Lop) [23], a fuzzer [12].

We installed all three tools on an Ubuntu machine with Intel Core i7 with 12GB of RAM. In particular, we employed the latest versions of both AFL (`afl-2.52b`) and Flawfinder (2.0.8). In the case of KLEE, we used a docker image (version 1.4) available in the project's web site. For each case we did a specific setup. We configured Flawfinder to search for threats of a high risk level and the arguments for AFL were configured differently for every utility. Specifically, we searched the manual pages of each utility and used

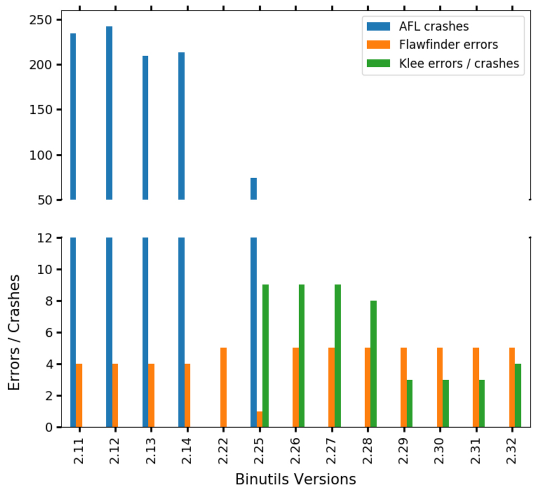

**Figure 1: Tool Results for the different versions of Binary Utilities.**

the arguments that every function works well with. Finally, we have extended AFL to automatically run the test cases that the tool produces (i.e. AFL generates a test case when the target application crashes) and examine the reasons behind the crashes.

## 3 ANALYSIS AND RESULTS

We present our results for each set of tools. For brevity, we provide details only for the case of Binary utilities. Recall that our prototype also gathers statistics regarding lines of code and comments. Overall, we observed that they both increase over time.

**Binary Utilities.** Figure 1 presents the results for the Binary utilities. Table 1 shows the results of our qualitative analysis. Specifically, it illustrates the utilities where a defect was found (together with the corresponding function). Table 1 also shows the type of the defect, when it was introduced and if and when it was removed.

Flawfinder found defects in all versions indicating that there is a slight increase in the number of bugs over time. All cases involve race condition issues. Also, a corresponding CVE entry exists [1] for the defect found in the `objcopy` utility.

Contrary to Flawfinder, KLEE showed that bugs seem to decrease over time. In addition, KLEE detected different kinds of bugs such as out-of-bounds pointers. Notably, the bugs seem to persist for four versions overall. Also, there are two CVE entries for the defects found in `cxxfilt` [3] and `objdump` [2] respectively. Finally, KLEE failed to analyze the first 6 versions due to architecture inconsistencies.

AFL indicated that the number of crashes decrease over time (they actually disappear from version 2.26 and onwards). We went one step further to examine the nature of the bugs that produced the crashes. To do so, we fed the test cases that the tool produced, to gdb. In turn, gdb reproduced each test and provided us with the details we needed. We observed that many crashes were products of the same bug (see Table 1). Note that, the bugs identified by AFL seem to persist over time.

**Core Utilities.** According to Flawfinder, all versions contained five race condition bugs. By further examining the results, we observed that two of them (found in `copy` and `cp`) were false positives (the constructs, i.e. `chown`, were found in comments). KLEE found seven defects in versions 8.10, 8.11, 8.12. The number of defects increases

**Table 1: Qualitative analysis on the results produced after analyzing Binary Utilities.**

| Tool | Utility | Method | Bug Type | Appeared | Removed |
|---|---|---|---|---|---|
| *Klee* [9] | `ar` | - | Out-of-bounds pointer | 2.25 | 2.29 |
| | | `fd` | Out-of-bounds pointer | 2.25 | 2.29 |
| | `cxxfilt` | `cplus-dem` | Out-of-bounds pointer | 2.25 | 2.28 |
| | `objdump` | `temp-file` | Abort Failure | 2.25 | - |
| *Flawfinder* [22] | `ar` | `strcpy` | Race Condition | 2.11 | 2.12 |
| | | `chmod` | Race Condition | 2.11 | - |
| | `rename` | `chmod` | Race Condition | 2.11 | - |
| | | `chown` | Race Condition | 2.11 | - |
| | `objcopy` | `chmod` | Race Condition | 2.22 | - |
| *AFL* [23] | `addr2line` | `tekhex` | Segmentation fault | 2.11 | 2.13 |
| | | `srec` | Segmentation fault | 2.11 | 2.13 |
| | | `raise` | Aborted | 2.11 | 2.13 |
| | | `opncls` | Segmentation fault | 2.13 | 2.22 |
| | `nm-new` | `raise` | Aborted | 2.11 | 2.22 |
| | | `srec` | Segmentation fault | 2.11 | 2.13 |
| | `objcopy` | `tekhex` | Bus error | 2.11 | 2.13 |
| | | `raise` | Aborted | 2.11 | 2.22 |
| | | `srec` | Segmentation fault | 2.11 | 2.13 |
| | | `opncls` | Segmentation fault | 2.13 | 2.22 |
| | objdump | `tekhex` | Bus Error | 2.11 | 2.22 |
| | | `srec` | Segmentation fault | 2.13 | 2.22 |
| | | opncls | Segmentation fault | 2.13 | 2.22 |
| | `size` | `tekhex` | Segmentation fault | 2.11 | 2.13 |
| | | `srec` | Segmentation fault | 2.11 | 2.22 |
| | | `opncls` | Segmentation fault | 2.13 | 2.22 |
| | `strip-new` | `tekhex` | Bus Error | 2.11 | 2.13 |
| | | `raise` | Aborted | 2.11 | 2.13 |
| | | `srec` | Segmentation fault | 2.13 | 2.22 |
| | | - | Segmentation fault | 2.25 | 2.26 |

in version 8.13 to eight and then decreases again in 8.20 (six). However, in the last two versions, the defect count goes to ten. In all cases the bugs involved out-of-bounds pointers. AFL did not find any bugs even though it covered many execution paths. Note that we ran the fuzzer even for one hour but with no results.

## 4 RELATED WORK

Ozment and Schechter [18] observed the code of the OpenBSD and identified a decrease in the rate at which bugs are being reported. Mitropoulos et al. [15, 16] have examined the evolution of potential bugs found in (1) software artefacts contained in the Maven Repository [5] and (2) JavaScript code used by popular websites. Their results show that it is unclear whether bug counts increase or decrease over time. Massacci et al. [14] examined 6 major versions of Firefox and found that security bugs seem to persist over time.

Focusing on bug-detection tools comparisons, Rutar et al. [19] showed that five static tools find non-overlapping bugs in Java programs, which is different from our finding. Finally, Austin and Williams [8] indicated that four bug finding techniques detect different kinds of defects, which complements our results.

## 5 CONCLUSIONS

Our results indicate that we cannot say if bugs either increase or decrease over time complementing previous related work [13–15]. We also observed that the tools find different kinds of bugs and can provide complementary results. Hence, developers can benefit by using tools based on different approaches to identify potential bugs. Moreover, Flawfinder seems to produce false alarms (a known disadvantage of static analysis tools [10, 17]) and detect only race condition errors while the dynamic tools identified more sophisticated bugs related to pointers and multi-integer overflows.

**Acknowledgements.** The author would like to thank Prof. Diomidis Spinellis who supervised this work, and Vaggelis Atlidakis and Dimitris Mitropoulos for their valuable comments.

## REFERENCES

[1] 2012. CVE-2012-3509. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3509. [Online; accessed 03-June-2019].

[2] 2018. CVE-2018-1000876. https://nvd.nist.gov/vuln/detail/CVE-2018-1000876. [Online; accessed 03-June-2019].

[3] 2018. CVE-2018-17360. https://nvd.nist.gov/vuln/detail/CVE-2018-17360. [Online; accessed 03-June-2019].

[4] 2018. Known Vulnerabilities in Mozilla Products. https://www.mozilla.org/en-US/security/known-vulnerabilities/. [Online; accessed 06-December-2018].

[5] 2018. The Maven Repository. https://mvnrepository.com/. [Online; accessed 06-December-2018].

[6] 2019. Free Software Foundation: Binutils. https://ftp.gnu.org/gnu/binutils/. [Online; accessed 03-June-2019].

[7] 2019. Free Software Foundation: Coreutils. https://ftp.gnu.org/gnu/coreutils/. [Online; accessed 03-June-2019].

[8] Andrew Austin and Laurie Williams. 2011. One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*. IEEE Computer Society, Washington, DC, USA, 97–106.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-CoverageTests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (USENIX-SS'08)*. USENIX Association, San Diego, CA, USA.

[10] Brian Chess and Gary McGraw. 2004. Static Analysis for Security. *IEEE Security and Privacy* 2, 6 (Nov. 2004), 76–79.

[11] Nigel Edwards and Liqun Chen. 2012. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. ACM, New York, NY, USA, 183–194. https://doi.org/10.1145/2382196.2382218

[12] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages.

[13] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu. 2013. A Study of Linux File System Evolution. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)*. San Jose, California.

[14] Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. 2011. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third international conference on Engineering secure software and systems (ESSoS'11)*. Springer-Verlag, Berlin, Heidelberg, 195–208.

[15] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2013. Dismal Code: Studying the Evolution of Security Bugs. In *Proceedings of the Learning from Authoritative Security Experiment Results (LASER) Workshop 2013,*. USENIX, 37–48.

[16] Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. 2019. Time Present and Time Past: Analyzing the Evolution of JavaScript Code in the Wild. In *16th International Conference on Mining Software Repositories: Technical Track (MSR '19)*. IEEE.

[17] Dimitris Mitropoulos and Diomidis Spinellis. 2017. Fatal injection: a survey of modern code injection attack countermeasures. *PeerJ Computer Science* 3 (Nov. 2017), e136.

[18] Andy Ozment and Stuart E. Schechter. 2006. Milk or wine: does software security improve with age?. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA.

[19] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. IEEE Computer Society, Washington, DC, USA, 245–256.

[20] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 771–781.

[21] Jaime Spacco, David Hovemeyer, and William Pugh. 2006. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories (MSR '06)*. ACM, New York, NY, USA, 133–136.

[22] David A. Wheeler. 2015. Flawfinder. https://dwheeler.com/flawfinder/. [Online; accessed 03-June-2019].

[23] Michal Zalewski. 2015. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. [Online; accessed 03-June-2019].