Understanding and Detecting Bugs in Network Operating Systems

Penghui Li 1155137827
phli@cse.cuhk.edu.hk

ABSTRACT

Network systems connect the Internet world. Bugs in network systems can lead to denial-of-service, performance issues, *etc.* Understanding network system bugs can benefit both the system developers and security analysts.

However, to the best of our knowledge, the bugs in network systems have not been well understood. The potential security impact has not been well investigated as well. In this project, we aim to understand the bugs in network systems. We hope to learn lessons to further guide future bug detection in network systems.

1 INTRODUCTION

Network Operating Systems (NOSs) connect multiple computers and devices and manage network resources in the settings of Software Defined Networking (SDN). NOSs manage multiple requests concurrently and provide the security in multiuser environments. To date, there are many NOSs being proposed and matained by network service vendors. For example, Cisco Internetwork Operating System is a family of network operating systems used on many Cisco Systems routers and current Cisco network switches [10]; DD-WRT [3] is a Linux based open-sourced NOS suitable for various WLAN routers and embedded systems.

Due to their popularity and critical uses, the bugs in these network operating systems can lead to severe security consequences. In particular, network operating systems could have performance bugs, which cause excessive resource consumption and could negatively affect user experiences. They can be further leveraged by attackers for launching application-layer denial-of-service (DoS) attacks [7]. By specially crafting inputs to trigger a performance bug in the network operating systems, the attacker can exhaust the server's resources (*e.g.*, memory and CPU) and make the services unavailable to normal users.

To the best of our knowledge, there currently has little knowledge about the prevalence of performance bugs in the wild. Prior studies on compilers generally focus on memory corruptions, whereas performance bugs, especially in close-sourced network operating systems, are not well investigated and understood yet. Some works studied performance issues in the regular expression engines [29, 34], desktop software [13], and Android applications [17]. Network operating systems, however, have not been covered. To this end, we conduct a comprehensive study of the performance bugs in network operating systems to answer the following research questions:

- What are the main categories of the bugs?
- What are the main factors that cause bugs?
- How widespread are the bugs in network operating systems?
- How severe are the bugs in network operating systems?

We empirically analyze 51 known performance bugs in mainstream network operating systems and thoroughly summarize their characteristics. We observe that there has been a continuous growth in the number of reported performance bugs in the past 3 years. We further identify that the ways that network operating systems handle the language's context-sensitive features are the dominant root cause of the performance bugs. We reveal that the developers of network operating systems usually mitigate such exploitation by enforcing a hard limit for the maximum number of backtracking for certain tasks, which, however, limits the intended functionality of the network operating systems.

In this work, we also explore detecting performance bugs in network operating systems. To the best of our knowledge, there is not any tailored tools specially designed for detecting performance bugs in network operating systems. Fuzzing [5, 16, 27] eliminates the limitations of static analysis techniques (*e.g.*, high false positives) [13, 24, 25]. It has become the go-to approach with thousands of vulnerabilities in real-world software. Without a doubt, performance bugs in network operating systems can be fuzzed as well.

We mainly focus on CPU exhaustion performance bugs. To detect them, we monitor the program execution under the generated inputs. We employ a statistical model using Chebyshev inequality to label abnormal cases as performance bugs. This can potentially cause duplicate bug reports as multiple bug reports with only slight difference in the inputs actually trigger the same bug. It is time-consuming and impractical to leverage human efforts to manually de-duplicate them. Existing bug de-duplicating methods using coverage profile and call stacks are inaccurate and not applicable to performance bugs. For example, it is hard to obtain an accurate

CSCI '5570, Spring 2021, CUHK
2021.

and deterministic call stack that reveals the situation when a performance bug is triggered. Therefore, we propose a execution trace similarity comparison algorithm to de-duplicate the reports. Specifically, we represent the execution trace of each report into a vector; we compute the cosine similarity between vector pairs and classify vectors (bug reports) with high similarity as the same bug.

We integrate abovementioned techniques into NetPerfFuzz. We demonstrated NetPerfFuzz outperformed the state-of-the-art works [16, 27] *better* performance slowdown, and *higher* code coverage. NetPerfFuzz is highly effective in generating test cases to trigger new code in the testing network operating systems. Though we have not identified any new bugs yet after 6-hour testing, we plan to further keep NetPerfFuzz running for more time. Hopenfully, our techniques can help find new bugs in the future.

In summary, we make the following contributions in this work.

- We conduct a systematic study on performance bugs in network operating systems. We present an empirical understanding of the performance bugs and the patches.
- We develop a new system, NetPerfFuzz, to effectively generate useful inputs to detect performance bugs in network operating systems.
- We demonstrate NetPerfFuzz can significantly outperform the state-of-the-art works.

2 BACKGROUND

2.1 Performance Bugs

Performance bugs in a program could degrade its performance and waste computational resources. Usually, people define performance bugs as software defects where relatively simple *source-code* changes can significantly optimize the execution of the software while preserving the functionality [6, 13, 14]. There can be several different performance issues regarding different categories of resources. For example, some performance bugs could cause excessive CPU resource utilization, resulting in unexpectedly longer execution time; some other bugs could lead to huge memory consumption because of uncontrolled memory allocation and memory leak [33].

Performance bugs lead to reduced throughput, increased latency, and wasted resources in software. They particularly impact the end-user experiences. What is worse, when a buggy application is deployed on the web servers, the bugs can be exploited by attackers for denial-of-service attacks, which can impair the availability of the services [20]. In the past, performance bugs have caused several publicized failures, causing many software projects to be abandoned [13, 22].

2.2 Network Operating Systems

Network operating system is a computer operating system that facilitates to connect and communicate various autonomous computers over a network. An autonomous computer is an independent computer that has its own local memory, hardware, *etc.* It is self capable to perform operations and processing for a single user. Network operating systems can be embedded in a router or hardware firewall that operates the functions in the network layer [4]. Typical real-world network operating systems include Cisco IOS [1], DD_WRT [3], Cumulus Linux [2], *etc.*

3 UNDERSTANDING PERFORMANCE BUGS

In this section, we present an empirical study on several known performance bugs in mainstream network operating systems to help understand their characteristics.

3.1 Data Collection

We investigate performance bugs in the Cicso IOS [**?**], which is one of the most popular network operating systems. Ideally, if more network operating systems are included in the study we can present a more comprehensive characteriazation. However, due the the limited time and human labours allowed for this work, we have to somehow restrict the study scope. Nevertheless, this work, as an early stage studying performance bugs in network operating systems, shall provide some general inspirations for future network system security analysis.

We manually collected performance bugs from the database of Common Vulenrabilities and Exposures (CVE) [21]. The detailed information about the vulnerabilities included in our study is presented in Appendix A. Some bug reports might be duplicates. We remove a duplicate bug from our dataset if it has a similar cause as another one, though it can have a different exploit vector. In summary, we obtained 51 distinct performance bugs reported from July 2007 to January 2021. We found all these performance bugs were abusing the CPU resources or memory resources.

We next characterize the performance bugs and present our findings.

3.2 Bug Disclosure Over Time

To understand the trend of performance bugs, we analyze the disclosure time of them. We present the performance bug disclosure in Table 3 in Appendix A. We observe that few bugs were reported before early 2010, and the number of reported bugs had been gradually growing from early 2017

till late 2020. In particular, 41 (80.39%) out of the 51 performance bugs were disclosed from January 2015 till Janurary 2021 (72 months); 10 (19.61%) bugs were reported from July 2007 to September 2014 (86 months). It reveals that such bugs had been gradually drawing the attention from the compiler developers and security analysts.

3.3 Root Causes

Identifying the common root causes of real-world performance bugs can benefit potential future research and software developments. We manually analyzed the performance bugs and successfully figured out the root causes for 39 bugs. We classify the root causes into three categories. A bug is assigned to multiple categories if it has multiple major causes.

R1: *Inefficient code.* Some inefficient code in the network operating systems could also lead to performance issues. For instance, some functions do not coordinate well for certain functionalities. We find that 19 performance bugs were caused by such inefficient code. Such kind of performance issues could be addressed by optimizing the inefficient code. However, each problem needs to be separately analyzed and fixed, which could be time-consuming. We next discuss an example of such inefficient code.

Data parsers are widely applied in network operating systems. For example, XML parsers and TLS layer parsers. Such parsers can analyze the input data among the computers within the network systems. Minor performance issues in such parsers could accumulate when the given inputs can repeatedly trigger the execution of such functions. Crafted inputs with lots of complicated and nested indents could result in repeated invocations of the function and cause performance bugs.

Second, some performance bugs are caused by regular expressions (regex). Some vulnerable regex could lead to performance bugs if the regex engine needs to backtrack (in exponential time) when matching some specific inputs. A failed matching attempt can lead the engine to backtrack with multiple choices. Thus the total number of possible backtracking paths is exponential if the match cannot be found eventually for every input symbol. This is also known as regular expression denial-of-service (ReDoS) [8, 29, 31, 34].

R3: *Problematic implementations.* Other causes of the bugs are specific to the compiler implementations or designs. Some compilers overlooked part of the CommonMark specification, for example, Unicode support. This can lead to infinite loops when unexpected inputs are provided to the compilers. Some other bugs in this category were caused by wrong data structures. 5 performance bugs fall into this category.

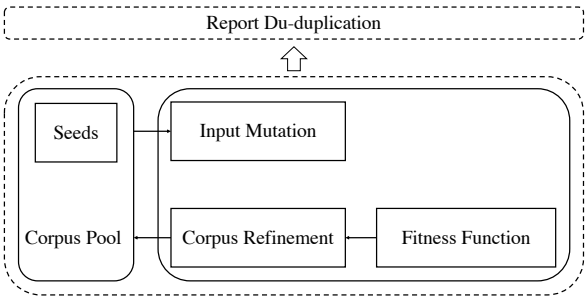

Figure 1: The architecture of NetPerfFuzz.

4 NETPERFFUZZ

Though we have characterized known performance bugs, it is unclear if there exist many unknown performance bugs in the network operating systems. Therefore, we try to detect performance bugs in real-world network operating systems. We focus on CPU resource exhaustion performance bugs in this work because they are the dominant type of performance bugs.

To avoid the high false-positive rates in static analyses [24, 25], we propose to use dynamic fuzz testing to detect and exploit performance bugs. To do so, we face a technical challenge. Since many distinct inputs can trigger the same performance bug, it is naturally challenging to accurately de-duplicate the bug reports. Prior performance bug fuzzers [5, 27] do not try to de-duplicate performance bugs. Other fuzzers for detecting *memory corruptions* de-duplicate bugs using the unique memory footprints (*e.g.*, coverage profiles and call stacks [15]) when the bugs are triggered, whereas one *performance bug* can potentially exhibit different memory footprints.

We overcome these challenges with NetPerfFuzz. The overall methodology is depicted in Figure 1. NetPerfFuzz follows the general fuzzing workflow and is built on top of AFL [35]. Inside the main fuzzing loop, to guide the fuzzer to detect CPU resource exhaustion performance bugs, we use a fitness function to measure if an input should be favored or not (§4.1). The fitness function considers both code coverage and resource usage. To report only unique bugs, we transform the execution trace of each report into a vector. We compute the cosine similarity of vector (report) pairs and group highly similar reports as duplicate ones (§4.2). We then present the implementation details (§4.3).

4.1 Fitness Function and Performance Bug Detection

NetPerfFuzz uses a fitness function to decide whether to favor a test case or not. We include both the coverage and

the control flow graph (CFG) edge hits into the fitness function. As in other works [11, 15, 26], the coverage feedback drives NetPerfFuzz to explore more newly discovered code. Only it, however, is not sufficient for our purpose as it does not consider loop iterations which are crucial for detecting high-complexity performance bugs [27]. The CFG edge hits, standing for the times a CFG edge is visited under a test case, enables NetPerfFuzz to explore *computationally expensive* paths. As stated in prior work [16], many programs (*e.g.*, PHP hash functions [16, 28]) do have non-convex performance space. We thus do not use the execution path length (*e.g.*, number of executed instructions) to guide NetPerfFuzz because it might fail to find the performance issues caused by local maxima. Therefore, as in the state-of-the-art work, PerfFuzz [16], we design NetPerfFuzz to favor those test cases that maximize certain CFG edge hits to better detect performance bugs. In this way, NetPerfFuzz tends to select test cases to either trigger new code or exhaust certain CFG edges. Note that we do not use runtime CPU usage or concrete execution time as the metric, because they show large variations affected by many uncontrollable factors, such as fuzzer's concurrent features and the characteristics of testing applications.

We design a statistical model to accurately identify performance bugs. Our statistical model first obtains the normal program execution behaviors to label abnormal ones as performance bugs. In particular, though the fitness function favors the local maxima, we still consider the total execution path length as the criteria of performance bugs. The execution path length is calculated as the sum of the CFG edge hits under a test case. We first prepare abundant random normal test cases; we feed each test case to the testing program and obtain the corresponding execution path length. We calculate the the mean (l_μ) and the standard deviation (l_σ) of the execution path lengths (l_i). We label a case as a performance bug if its execution path exceeds the normal level to a certain extent. According to Chebyshev inequality (as shown in Equation 1), the probability of the random variable l_i that is k-standard deviations away from the mean (l_μ) is no more than $1/k^2$. Since only in rare cases would the execution path length significantly deviate from the normal situations, therefore, we label a performance bug if its execution path length l_t is more than kl_σ away from the l_μ (see Equation 2).

$$P(|l_i - l_\mu| > kl_\sigma) \leq \frac{1}{k^2} \tag{1}$$

$$l_t > l_\mu + kl_\sigma \tag{2}$$

4.2 Report De-Duplication

Though different test cases could all exceed the threshold, they could actually trigger the same performance bug. De-duplicating the bug reports is necessary for a more precise result, whereas prior works [16, 27] do not apply automated methods to de-duplicate the reports. Existing fuzzing works identify unique bugs using the call stack for memory corruptions (*e.g.*, crashes). However, it does not fit well our purposes for performance bug de-duplication. Though we can possibly collect the call stack as well (*e.g.*, by forcibly terminating the program at some point), the call stack might not be accurate enough to differentiate unique bugs. This is because the exactly critical call stack for a performance bug can hardly be accurately exposed. Unlike memory corruptions that have a deterministic call stack when a bug is triggered, performance bugs might exhibit diverse call stacks depending on when to obtain them. Therefore, a better report de-duplication method is needed.

We propose a new bug de-duplicating approach by merging reports with similar execution traces. The high-level idea is that different exploiting inputs of the same performance bug should exhibit similar execution traces, *i.e.*, most CFG edges are visited in similar frequencies. In particular, we apply the test cases in the reports to the instrumented target software and obtain the CFG edge hits for each edge. We summarize the unique CFG edges that are visited in all reports during fuzz testing into an n-dimensional vector space, where n is the total number of unique CFG edges being visited and each dimension in the vector space corresponds to a CFG edge. For each report, we construct an edge-hit vector, *e.g.*, $\vec{v} = (c_1, c_2, ..., c_n)$. Each dimension ($c_i$) represents the hit count of the ith CFG edge in that report. To consider if two reports point to the same bug, we construct their edge-hit vectors (*e.g.*, $\vec{v}, \vec{v}\,'$) and compute the cosine similarity (as shown in Equation 3). Cosine similarity is based on the inner product of the two vectors and thus naturally assigns higher weights to the dimensions with larger values (*i.e.*, edges visited most). Therefore, we calculate the cosine similarity between every two reports and merge reports as the same bug if the cosine similarity of their corresponding edge-hit vectors exceeds a threshold.

$$Sim(\vec{v}, \vec{v}\,') = \frac{\vec{v} \cdot \vec{v}\,'}{|\vec{v}||\vec{v}\,'|} = \frac{\sum_{j=1}^{n} c_j c'_j}{\sqrt{\sum_{j=1}^{n} c_j^2}\sqrt{\sum_{j=1}^{n} c_j'^2}} \tag{3}$$

4.3 Implementation

We implemented the fuzzing part of NetPerfFuzz above an AFL-based fuzzer, PerfFuzz [16]. Specifically, we enhanced a C/C++ compiler to instrument the testing software; we modified AFL's `showmap` functionality to trace the execution

Table 1: The dataset of the compilers and bug detection results. Rep. is the reports from the fuzzer. U-Rep. is the unique report after de-duplicating the reports. Con. means the confirmed bugs.

Software	Lang.	# Rep.	# U-Rep.
cmark (0.29.0)	C	1321	7
MD4C (0.4.7)	C	239	3
cmark-gfm (0.29.0)	C	981	4

on the instrumented applications to obtain the CFG edge hits for report de-duplication.

5 DETECTING PERFORMANCE BUGS VIA NETPERFFUZZ

In this section, we investigate the prevalence of performance bugs in the wild. We planed to test network operating systems via fuzzing in this work. However, we temporarily failed to do so due to some practical issues. For example, 1) the time for this research project is too short; and 2) we have not found a flexibile way to apply our instrumented C/C++ compilers to compile an executable network operating systems for the fuzzing. After more than one week exploration, we decided to give it up. Give the truth, we apply NetPerfFuzz to detect performance bugs in othe systems. Since the methodology we propose in this work is general thus we test NetPerfFuzz in 3 compilers[1] to verify its efficacy. We naturally admit this limitation. However, we still believe this engineering problem can be fixed give more time. We will definitely make it as our future work.

Experiments. Each compiler is first instrumented using our enhanced C compiler in NetPerfFuzz. We then apply NetPerfFuzz to detect performance bugs on the instrumented compilers. We apply the PoCs collected in §3 as the initial seeds and configure NetPerfFuzz to use a single process and a timeout of 6 hours. Through our preliminary study, we empirically set k to 5 and the cosine similarity threshold to 0.91 for all testing software. All experiments described in this section are conducted on a server running Debian GNU/Linux 9, with an Intel Xeon CPU and 96GB RAM.

5.1 Results

We present the performance bug detection results in Table 1. Duplicate performance bug reports are naturally common during fuzzing. The fuzzing part of NetPerfFuzz reported 2,541 cases in total and our de-duplicating algorithm merged them into 14 distinct reports. We observe that all the 14 cases did successfully slow down the compilers by 2.31× to 7.28× compared to normal-performance cases.

[1]The authors choose them as they have worked on related topics before.

We further manually check the reports to validate the performance bugs. Since NetPerfFuzz limits the input size like in other works [11, 16, 27] due to the concerns of large search space, our manual analysis attempts to identify the severity of the performance slowdown in more realistic scenarios, *i.e.*, larger input sizes. To this end, we first identify the exploit input patterns in the reports that exhaust the run-time resources. With the patterns, we further construct larger test cases to verify the performance issues in practice. Finally, 7 cases in 2 compilers were confirmed as performance bugs, including 4 new bugs, after our manual analysis. We are in the process of reporting the new bugs to the concerned vendors. At the time of writing, 1 bug has been well acknowledged.

We found no bug in MD4C. The developers of MD4C explicitly mention that they seriously considered the performance as one of their main focuses during the implementation [19]. Therefore, the performance bugs could be avoided with domain knowledge and special care, which are often difficult for most developers.

5.2 Comparison

We compare NetPerfFuzz with two state-of-the-art works, SlowFuzz [27] and PerfFuzz [16]. NetPerfFuzz and PerfFuzz are implemented above AFL whereas SlowFuzz is built on top of libFuzzer [18]. SlowFuzz (libFuzzer) uses in-process fuzzing, which is much faster as it has no overhead for process start-up; however, it is also more fragile and more restrictive because it traps and stops at crashes [18]. Nevertheless, we evaluate all the tools with the same dataset in Table 1 and run them for the same amount of time—6 hours—for a fair comparison. We failed to run SlowFuzz on MD4C because of some unexpected crashes after several minutes of the execution. To the best of our knowledge, there is no way to suppress such crashes. NetPerfFuzz and PerfFuzz—AFL-based fuzzers—do not suffer from this problem.

The results show that NetPerfFuzz outperformed PerfFuzz and SlowFuzz by detecting more performance bugs. In particular, PerfFuzz reported 820/114/783 cases in cmark/MD4C/cmark-gfm, respectively; SlowFuzz reported 432/408 cases in cmark/cmark-gfm, respectively. These results also demonstrate the need of a report de-duplication method. We applied our report de-duplication algorithm to identify unique bugs and then manually confirmed the reports. Finally, PerfFuzz detected 2/0/2 real performance bugs in cmark/MD4C/cmark-gfm, respectively. SlowFuzz detected 1/1 real performance bugs in cmark/cmark-gfm, respectively.

5.2.1 Performance Slowdown. Table 2 shows the performance slowdown caused by the inputs generated by NetPerfFuzz, PerfFuzz, and SlowFuzz. We use the maximum execution path length as the performance metric, and normalize the performance slowdown using a baseline of a

Table 2: The performance slowdown and code coverage of NetPerfFuzz, PerfFuzz [16], and SlowFuzz [27]. The Best Slowdown across all tools is normalized over the baseline of the same random normal-performance case. Line Cov. and Func. Cov. denote line coverage and function coverage, respectively.

Tool	Software	Best Slowdown	Line Cov.	Func. Cov.
NetPerfFuzz	cmark	7.28×	71.90%	67.91%
	MD4C	2.31×	76.22%	58.11%
	cmark-gfm	6.54×	55.78%	57.35%
PerfFuzz	cmark	6.82×	56.21%	51.35%
	MD4C	2.21×	67.20%	50.20%
	cmark-gfm	5.05×	48.26%	44.31%
SlowFuzz	cmark	4.32×	40.28%	41.65%
	cmark-gfm	3.29×	38.30%	42.33%

random normal-performance case. We notice that, though all tools caused performance slowdown on the testing applications, NetPerfFuzz achieved a 14.71% higher average best performance slowdown over PerfFuzz, and 41.21% over SlowFuzz. Furthermore, we observe that NetPerfFuzz could generate inputs that slow down the compilers much faster than the other tools. For example, to reach a 4.32× performance slowdown on cmark, NetPerfFuzz took 3.2 hours, whereas PerfFuzz and SlowFuzz used 3.9 hours and 6.0 hours, respectively. This demonstrates the high efficacy of NetPerfFuzz in detecting performance bugs.

5.2.2 Code Coverage. We also evaluate the code coverage each tool achieves. We collect the test cases generated by each tool and run on afl-cov [23], which detects the code coverage using the overall execution traces covered by the test cases. Though SlowFuzz is not based on AFL, we believe using its test cases on afl-cov can accurately reflect the code coverage under a fair metric.

We present the results of line coverage and function coverage in Table 2. NetPerfFuzz outperformed PerfFuzz by 20.75% more lines of code and 13.17% more functions; NetPerfFuzz outperformed SlowFuzz by 28.68% more lines of code and 19.80% more functions.

6 LIMITATIONS

This work has several limitations that need to be further addressed. First, we failed to instrument some network operating systems for the evaluation and alternatively evaluated in other systems. This problem, was out of the foreseeable scope of the authors when proposing the idea. The authors also were not able to fix it given limited time. Though our technique is general, there might be some unknown challenges that we cannot foresee now. We have to tackle this limitation to make NetPerfFuzz applicable for our initial purposes. Nevertheless, our evaluation somehow demonstrated the efficacy of NetPerfFuzz. Similarly, we are in certain confidence that NetPerfFuzz can also perform excellently once the above challenge is solved. Second, also due to the lack of human efforts, the empirical study is deep enough. There are many other interesting aspects that can be studied further. For example, how are the performance bugs patched? We will further investigate these directions in the future.

7 RELATED WORK

Understanding performance bugs. Understanding the characteristics of performance bugs can help design techniques to detect and fix performance bugs. Existing studies focus on the performance bugs in programs on the desktop platform [13, 36], mobile platform [17], and the web server end [13], *etc.* For instance, Zaman *et al.* studied performance bugs in Firefox and Chrome and provided suggestions to fix the bugs and to validate the patches [36]. However, there is little understanding of performance bugs in the network operating systems. Sub *et al.* studied GCC and LLVM compilers but they did not focus on the performance issues [32]. Our work studies an understudied problem—performance bugs.

Detecting performance bugs. The detection of performance issues has drawn significant attention from researchers over the past years. Prior studies focus on application-layer DoS vulnerabilities [9, 12, 20], algorithmic complexity DoS vulnerabilities [7, 30], and other general performance issues [13, 17]. Static methods analyze the source code of the applications and diagnose vulnerable bug patterns, for example, repeated loops [24, 25]. Dynamic methods are also applied to identify performance bugs. SlowFuzz [27], PerfFuzz [16], and HotFuzz [5] propose new fuzzing solutions to detect the worst-case algorithmic complexity vulnerabilities. Our work improves existing solutions via a fitness function and a new report de-duplication method. Our evaluation has well demonstrated its efficacy.

8 CONCLUSION

Performance bugs in network operating systems are previously understudied. This paper conducted a systematic study to understand the characteristics of the performance bugs. We designed NetPerfFuzz with several new techniques to detect performance bugs in the wild. Our evaluation demonstrates report de-duplication is necessary and our solution is effective. We hope our research, as an initial step, can shed some light on future security analysis and software developments.

REFERENCES

[1] . 2021. Cisco IOS technology. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html.

[2] . 2021. Cumulus Linux. https://cumulusnetworks.com/products/cumulus-linux/.

[3] . 2021. DD-WRT. https://dd-wrt.com/.

[4] Emad Al-Shawakfa and Martha Evens. 2001. The dialoguer: An interactive bilingual interface to a network operating system. *Expert Systems* (2001).

[5] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA.

[7] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium (Security)*. Washington, DC.

[8] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL.

[9] Veronika Durcekova, Ladislav Schwartz, and Nahid Shahmehri. 2012. Sophisticated denial of service attacks aimed at application layer. In *2012 ELEKTRO*. IEEE.

[10] Bradley Edgeworth, Aaron Foss, and Ramiro Garza Rios. 2014. *IP routing on Cisco IOS, IOS XE, and IOS XR: an essential guide to understanding and implementing IP routing protocols*. Cisco Press.

[11] Google. 2021. OSS-Fuzz. https://google.github.io/oss-fuzz/.

[12] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. 2017. Detecting HTTP-based application layer DoS attacks on web servers in the presence of sampling. *Computer Networks* (2017).

[13] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China.

[14] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. 2010. Finding latent performance bugs in systems implementations. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Santa Fe, NM.

[15] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[16] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*. Amsterdam, Netherlands.

[17] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India.

[18] LLVM. 2021. libFuzzer. https://hammer-vlsi.readthedocs.io/en/stable/LibFuzzer.html.

[19] md4c. 2021. md4c. https://github.com/mity/md4c.

[20] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. 2018. Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[21] MITRE. 2021. CVE. https://cve.mitre.org/.

[22] G. E. Morris. 2004. Lessons from the colorado benefits management system disaster. www.ad-mkt-review.com/publichtml/air/ai200411.html.

[23] mrash. 2021. afl-cov - AFL Fuzzing Code Coverage. https://github.com/mrash/afl-cov.

[24] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.

[25] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA.

[26] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[27] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[28] PHP. 2021. PHP Hash. https://www.php.net/manual/en/function.hash.php.

[29] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. Rescue: Crafting regular expression dos attacks. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.

[30] R. Smith, C. Estan, and S. Jha. 2006. Backtracking Algorithmic Complexity Attacks against a NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*.

[31] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[32] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. Saarbrücken, Germany, 294–305.

[33] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Seoul, Korea.

[34] Valentin Wüstholz, Oswaldo Olivo, Marijn JH Heule, and Isil Dillig. 2017. Static detection of DoS vulnerabilities in programs that use regular expressions. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Uppsala, Sweden.

[35] Michal Zalewski. 2021. american fuzzy lop. https://github.com/google/AFL.

[36] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2012. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*.

A CVES IN OUR STUDY

We present the CVEs included in our study (§3) in Table 3.

B ARTIFACT INSTRUCTIONS

B.1 Download and Install NetPerfFuzz

We make NetPerfFuzz publically for the assessment of this project at https://github.com/peng-hui/csci5570-project. Interested readers feel free to download, install, and verify our results. Below are some details.

(1) Download or clone the project to your machine or server. The code is tested on Debian GNU/Linux 9.
(2) Compile the AFL via `cd csci5570-project/afl && make`.
(3) Compile the instrumented compile via `cd llvm_mode && make`.

B.2 Instrument the Testing Software

This part varies depending on the testing software. Please repalce the CC compiler with `csci5570-project/afl/afl-clang-fast` and CXX compiler with `csci5570-project/afl/afl-clang-fast++`. Make sure you can compile the testing software successfully.

B.3 Run NetPerfFuzz

Suppose you have compiled `cmark`. Then you only need to run `/afl-fuzz -p -i seeds -o out/ -N 64 ./cmark @@` The `seeds` is optional. The `out` is the result directory.

B.4 Report De-Duplication

Simply run `python3 cosine-similarity.py`. You might have to adjust the result directory accordingly.

Table 3: The CVEs and the empirical study. More details about each CVE can be found via searching the CVE ID.

CVE ID	Disclosure Time	Root Causes	CVE ID	Disclosure Time	Root Causes
CVE-2021-1267	01/13/2021	R1	CVE-2020-3567	10/07/2020	R2
CVE-2020-3175	02/26/2020	R2	CVE-2020-3164	03/04/2020	R2
CVE-2019-1967	08/28/2019	R2	CVE-2019-1957	08/07/2019	R1
CVE-2019-1947	02/19/2020	R2	CVE-2019-1806	05/15/2019	R1
CVE-2019-1721	04/17/2019	R1	CVE-2019-1720	04/17/2019	R1
CVE-2019-1718	04/17/2019	N/A	CVE-2019-12698	10/02/2019	R1 R2
CVE-2018-15462	05/01/2019	R1	CVE-2018-15460	01/09/2018	N/A
CVE-2018-15454	10/31/2018	R1	CVE-2018-15388	05/01/2019	R2
CVE-2018-0272	04/18/2018	R1	CVE-2018-0257	04/18/2018	R2
CVE-2018-0230	04/18/2018	R2	CVE-2018-0228	04/18/2018	R2
CVE-2018-0177	03/28/2018	R1	CVE-2018-0094	01/17/2018	R1
CVE-2017-6641	05/17/2017	R1 R2	CVE-2017-3885	04/05/2017	R1
CVE-2017-3820	02/01/2017	R1	CVE-2017-12244	10/04/2017	R1
CVE-2017-12237	08/27/2017	R2	CVE-2017-12211	09/06/2017	R1
CVE-2016-6301	08/03/2016	R2	CVE-2016-1483	09/14/2016	R1
CVE-2016-1440	06/27/2016	R2	CVE-2016-1343	04/28/2016	R2
CVE-2015-6386	09/28/2015	R2	CVE-2015-6295	09/15/2015	N/A
CVE-2015-4283	07/20/2015	R1	CVE-2015-0772	06/09/2015	N/A
CVE-2015-0765	06/03/2015	N/A	CVE-2015-0754	05/27/2015	R2
CVE-2015-0744	05/29/2015	N/A	CVE-2015-0617	02/16/2015	R2
CVE-2015-0581	01/28/2015	N/A	CVE-2014-3353	09/02/2014	N/A
CVE-2014-3293	10/27/2014	R2	CVE-2013-5566	11/06/2013	R1
CVE-2013-3453	08/21/2013	R1	CVE-2013-1230	04/30/2013	R1 R2
CVE-2012-3079	05/30/2012	N/A	CVE-2012-2472	05/07/2012	N/A
CVE-2011-3287	08/29/2011	R1	CVE-2010-4670	01/06/2011	N/A
CVE-2007-3698	07/25/2007	N/A			