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 [9]; DD-WRT [1] 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 [8]. 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, 31], 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 XXXknown 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 [2, 16, 27] eliminates the limitations of static analysis techniques (*e.g.*, high false positives) [13, 23, 24]. 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 [3, 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 [30].

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].

Table 1: The existing performance bugs included in our study. Lang. means the underlying programming language for implementing the Markdown compiler.

Software	Lang.	# Bugs	Time Periods
CommonMark-spec	N/A	13	10/26/2014 - 02/17/2020
cmark	C	13	01/14/2017 - 12/20/2020
MD4C	C	6	03/10/2019 - 09/10/2019
commonmark.js	JS	9	09/28/2017 - 08/13/2019
markdown-it	JS	8	08/14/2019 - 11/20/2020

2.2 Network Operating Systems

3 UNDERSTANDING PERFORMANCE BUGS

In this section, we present an empirical study on several known performance bugs in mainstream Markdown compilers to help understand the characteristics of the performance bugs.

3.1 Data Collection

We investigate performance bugs in the CommonMark specification [6] and 4 representative Markdown compilers chosen from the recommended implementations of the specification [7]. In particular, cmark [4] and MD4C [19] are two high-performance Markdown compilers written in C; cmark is ported and extended for GitHub [10]. commonmark.js [5] and markdown-it [18] are two Node.js packages that are used in both client-side and server-side applications. We do not consider the software that uses Markdown compilers as one of its sub-components/modules (*e.g.*, GitLab [11]), because such software usually does not modify the internal workflow of the included compilers. We limit our manual analysis to only four representative Markdown compilers because the analysis is quite time-consuming. Furthermore, we find that our current software set already allows us to characterize the bugs and extract some general features, which we will present later in this section.

In these Markdown compilers, we manually collected 49 distinct performance bugs from their public bug disclosure channels and their GitHub repository issues. The bug distribution is presented in Table 1. We found all these performance bugs were abusing the CPU resources instead of other resources like memory. This suggests CPU resource exhaustion performance bugs are the dominant type of performance bugs. We next characterize the 49 performance bugs and present our findings.

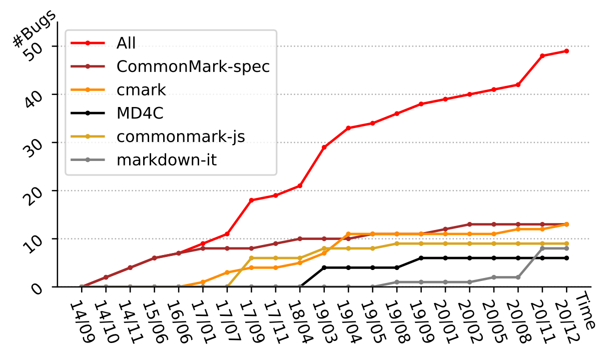

Figure 1: The number of performance bug reports over time from October 2014 to December 2020.

3.2 Disclosure and Patch Time

To understand the trend of performance bugs, we analyze the disclosure time of the 49 bugs. We depict the number of performance bugs along the time they were disclosed for each software in Figure 1. We observe that few bugs were reported before early 2015, and the number of reported bugs had been gradually growing from early 2018 till late 2020. In particular, 28 (57.14%) out of the 49 performance bugs were disclosed from April 2018 till December 2020 (22 months); 21 (42.86%) bugs were reported from August 2014 to December 2017 (41 months). It reveals that such bugs had been gradually drawing the attention from the compiler developers and security analysts.

We further analyze the time it takes to release a patch since a bug is initially reported. We were able to successfully determine the time for 32 performance bugs. For the rest bugs, either we did not find the explicit bug patch time [21], or they have not been patched yet [25]. We find that the average duration to patch performance bugs is 19 days. Further, 23 (71.88%) bugs were patched within 30 days. We particularly investigated those bugs that took much longer patch time, *e.g.*, more than 4 months. We observed that they were usually related to the ambiguity of the CommonMark specification. Thus their patches usually require some work from both the compiler developers and the specification maintainers. Some of the bugs and the patches lead to the modifications of the language specification.

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 49 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: *Super-linear algorithms.* Some normal algorithms implemented in Markdown compilers have super-linear worst-case complexity [16, 27]. Attackers can craft inputs to trigger the worst-case behaviors and lead to performance issues. The majority (25 out of 39) of bugs were related to such worst-case behaviors.

Some Markdown syntaxes (*e.g.*, links, emphasis and strong emphasis, HTML blocks) are related to the language's context-sensitive features. As discussed in **??**, supporting context-sensitive features in Markdown requires the compilers to backtrack, which could take more than linear time. The backtracking strategies can easily be abused with crafted inputs hence lead to performance issues. For instance, links were the primary vulnerable syntax in Markdown compilers, where 11 of the known performance bugs could be exploited with special inputs with links. Similarly, 8 of the bugs were caused by the buggy emphasis and strong emphasis handlers. Our study reveals that the implementation of the context-sensitive features in the Markdown compilers are prone to containing performance bugs.

One typical input pattern that exploits the context-sensitive syntax handler to trigger performance bugs is *many open tokens.* This pattern can lead the compilers to repeatedly search a close token towards the end of the input string for each such open token, and also force the compilers to backtrack to correct wrong options the compilers have selected. For example, deeply nested CDATA block open delimiters can result in an excessive compilation time. When fed with n-nested CDATA block open delimiters (*e.g.*, `'<![CDATA[<![CDATA[<![CDATA[...'`) that are not closed with the corresponding close delimiters (*i.e.*, `']]>'`) or are closed in the end of the input string, the compilers need to compare with all tokens in the input string to determine if an open delimiter can be closed or not. Once the compilers find an open delimiter cannot be closed, they switch to other possible options for that delimiter next, for instance, the open delimiter `'<!'` in `'<!A>'`, which cannot be closed either. Thus the time for handling such input strings is at least in polynomial time complexity. By providing a long input with many such open tokens, it is simple to cost the compiler several-second or even more execution time.

R2: *Inefficient code.* Some inefficient code in the Markdown compilers could also lead to performance issues. For instance, some functions do not coordinate well for certain functionalities. We find that 9 performance bugs were caused by such inefficient code. Unlike the algorithms in R1, such 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.

Minor performance issues in individual problematic functions could accumulate when the given inputs can repeatedly trigger the execution of such functions. For example, in one bug, cmark calls `S_find_first_nonspace()` to find the first non-space character from the current offset in a line. The function in a second call would still search from the initial position, even if in a previous call it has already recognized the location of the first non-space character. This means some function calls to `S_find_first_nonspace()` sometimes were unnecessary. Crafted inputs with lots of complicated and nested indents could result in repeated invocations of this function and cause performance bugs. The problem, however, can be solved by using better strategies like cashing the positions of the previously found non-space characters.

R3: *Implementation-specific issues.* 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.

3.4 Patches of Performance Bugs

We investigate the patches of performance bugs in Markdown compilers to understand how they were addressed. We manage to identify the bug fix patterns for 28 performance bugs. We present our findings below.

P1: *Enforcing limits.* The most common patch pattern is to add limits for certain conditions such as the maximum depth of the nested structure, although the CommonMark specification does not explicitly specify any such limits. When such limits are reached, the compilers directly regard the rest unanalyzed inputs as plain text. Enforcing limits can prevent excessive CPU usages caused by the worst-case exploitation of too large test cases. However, the intended functionality might be violated. It is also difficult to set a correct limit to prevent all attacks while not breaking some unusual yet legitimate inputs. Such a strategy has been applied to patch 13 out of the 28 bugs we investigate.

P2: *Logic changes.* Logic changes sometimes are necessary as the bugs are caused by the incorrect coordination among multiple program components and functions. Some inefficient code snippets need to be further optimized to eliminate the underlying performance issues. For some other performance bugs caused by incorrect regular expressions, compiler developers mainly review and rewrite the regular expressions.

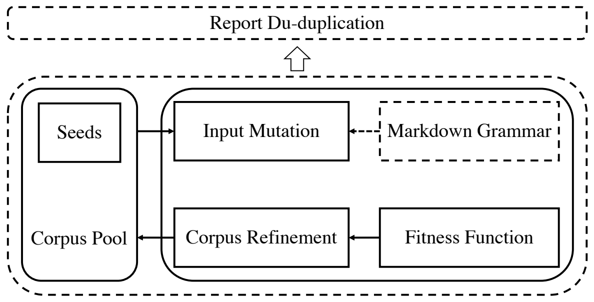

Figure 2: The architecture of NetPerfFuzz.

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 [23, 24], 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 [2, 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 2. NetPerfFuzz follows the general fuzzing workflow and is built on top of AFL [32]. 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 [12, 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 edges 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.

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

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

5 CONCLUSION

In this work, we conduct a comprehensive literature review of TCP incast control. We present an in-depth exploration of existing solutions to tackle TCP incast congestion problem. We categorize the solutions by their implemented network architecture layers and outline the general challenges of the problem. We show that the feedback mechanism is a practical approach to monitor the network traffic and prevent TCP incast congestion. We probe several limitations in prior works and introduce potential future work on this topic. Though well studied, there are still potential directions that can be further explored to improve the state-of-the-art works. We do hope this study can shed some light on future research of TCP incast control.

REFERENCES

[1] . 2021. DD-WRT. https://dd-wrt.com/.

[2] 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.

[3] 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.

[4] cmark. 2021. cmark. https://github.com/commonmark/cmark.

[5] commonmarkjs. 2021. commonmark.js. https://github.com/commonmark/commonmark.js.

[6] commonmark.org. 2021. commark-spec. https://github.com/commonmark/commonmark-spec.

[7] commonmark.org. 2021. List of CommonMark Implementations. https://github.com/commonmark/commonmark-spec/wiki/List-of-CommonMark-Implementations.

[8] 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.

[9] 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.

[10] GitHub. 2021. GitHub's markdown compiler: cmark-gfm. https://github.com/github/cmark-gfm.

[11] GitLab. 2021. GitLab. https://gitlab.com/.

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

[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] markdown it. 2021. markdown-it. https://github.com/markdown-it/markdown-it.

[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] mity. 2017. Pathological input: Unclosed inline links. https://github.com/commonmark/cmark/issues/218.

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

[23] 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.

[24] 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.

[25] nwellnhof. 2020. Quadratic behavior with smart quotes. https://github.com/commonmark/cmark/issues/373.

[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] 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.

[31] 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.

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

A FUTURE WORK

B ARTIFACT INSTRUCTIONS