Understanding and Detecting Performance Bugs in Markdown Compilers

Paper #66

***Abstract*—Markdown compilers are widely used for translating plain Markdown text into formatted text, yet they suffer from performance bugs that cause performance degradation and resource exhaustion. Currently, there is little knowledge and understanding about these performance bugs in the wild.**

In this work, we first conduct a comprehensive study of known performance bugs in Markdown compilers. We identify that the ways Markdown compilers handle the language's context-sensitive features are the dominant root cause of performance bugs. To detect unknown performance bugs, we develop MDPERFFUZZ, a fuzzing framework with a syntax-tree based mutation strategy to efficiently generate test cases manifesting such bugs. It equips an execution trace similarity comparison algorithm to de-duplicate the bug reports. With MDPERFFUZZ, we successfully identified 216 new performance bugs in real-world Markdown compilers and applications. Our work demonstrates that performance bugs are a common, severe yet previously overlooked security issue.

I. INTRODUCTION

Markdown is an easy-to-use markup language. Its compilers analyze input text to generate formatted text with decorated styles according to the Markdown language syntaxes, and are commonly used in many scenarios such as static web page generation tools, server-side web applications, and code hosting software. For example, GitHub, the most popular code hosting website, is equipped with a Markdown compiler at its client side; marked [22], a Node.js Markdown compiler package, receives over 400 million downloads in the past 5 years [33].

Due to their popularity and critical uses, the bugs in these Markdown compilers can potentially impact many services and their users. In particular, Markdown compilers could have performance bugs, which cause excessive resource consumption and negatively affect user experiences. They can be further leveraged by attackers for launching denial-of-service (DoS) attacks [8]. By specially crafting inputs to trigger a performance bug in a Markdown compiler in a server-side application, 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 Markdown compilers, are not well investigated and understood. Some works studied performance issues in the regular expression engines [39, 46], desktop software [15], and Android applications [19]. Markdown compilers, however, have not been covered.

To this end, we conduct a comprehensive study of performance bugs in Markdown compilers to answer the following research questions:

- What are the main factors that cause performance bugs?
- How widespread are performance bugs in Markdown compilers?
- How severe are performance bugs in Markdown compilers?

We empirically analyze 49 known performance bugs in mainstream Markdown compilers 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 Markdown compilers handle the language's context-sensitive features are the dominant root cause of these performance bugs. In particular, the Markdown language requires Markdown compilers to support certain backtracking strategies [43]. The backtracking strategies can be abused or exploited by specially crafted yet syntactically legitimate inputs for causing worst-case performance issues. Such malicious inputs do not violate any specification of the Markdown language. We reveal that the developers of Markdown compilers 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 Markdown compilers.

We then seek to detect unknown performance bugs in Markdown compilers. In particular, we consider leveraging fuzz testing/fuzzing to detect performance bugs. Fuzzing [2, 18, 37] has become the go-to approach that has successfully detected thousands of vulnerabilities in real-world software. It is free from some limitations of static analysis techniques (*e.g.*, high false positives) [15, 28, 29]. However, existing fuzzers [18, 37] for performance bugs are unaware of the Markdown grammar hence cannot efficiently generate inputs to thoroughly exercise the compilers. Therefore, besides the conventional bit/byte level mutations [48], we develop a novel syntax-tree based mutation strategy [44] to preserve and extend useful Markdown syntaxes during the input mutation process. This helps efficiently generate high-quality inputs.

We focus on CPU exhaustion performance bugs, which are the dominant type of 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. Like in prior works [18, 37], our approach can potentially lead to duplicate bug reports because multiple inputs with only slight differences could actually trigger the same performance bug. It is time-consuming and impractical to leverage human efforts to manually de-duplicate them. Yet it is non-trivial to automatically de-duplicate the bug reports in the scenario of performance bugs. 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 and deterministic call stack that represents the run-time program state when a performance bug is triggered. To tackle this

challenge, we propose an execution trace similarity comparison algorithm to de-duplicate the reports. Specifically, we represent the execution trace of each report into a vector, compute the cosine similarity between vector pairs, and classify highly similar vectors (bug reports) as the same bug.

We integrate the above-mentioned techniques into MDPERFFUZZ, a fuzzer specialized in detecting performance bugs in Markdown compilers implemented in C/C++. We will release it as an open-source software. With MDPERFFUZZ, we successfully detected 7 new performance bugs in 2 standalone Markdown compilers. It also outperformed the state-of-the-art works [18, 37] with *more* unknown performance bugs detected, *better* performance slowdown, and *higher* code coverage. We summarized the exploits generated by MDPERFFUZZ into 45 attack patterns. We further applied them to detect performance bugs in other Markdown compilers and the Markdown components of popular real-world applications that are not written in C/C++ thus cannot be directly fuzzed by MDPERFFUZZ. We found 209 new performance bugs, which could potentially affect millions of websites and their users. Our evaluation results demonstrate that the performance bugs in Markdown compilers are widespread and can lead to severe security issues. More efforts from the community shall be devoted to such an emerging security problem. We are in the process of disclosing our findings to the affected parties. At the time of writing, 25 bugs have been acknowledged.

In summary, we make the following contributions.

- We presented the first empirical study of the performance bugs in Markdown compilers.
- We developed a new system, MDPERFFUZZ, to efficiently generate high-quality test inputs to detect performance bugs in Markdown compilers.
- We identified 175 new performance bugs in 17 Markdown compilers and 41 new performance bugs in 4 popular real-world applications.

II. BACKGROUND

A. Markdown

Created by Swartz and John Gruber in 2004 [42], Markdown has been a prevalent markup language for creating formatted text from plain text. To date, CommonMark [6] has become a well-recognized Markdown specification in the community. Many popular applications such as code hosting software (*e.g.*, GitLab) and web applications (*e.g.*, WordPress) currently implement their Markdown components guided by the CommonMark specification. We introduce some of its important features below.

- *Valid Markdown documents.* Any sequence of characters is a valid Markdown document, where a character is usually a Unicode code point. This is different from many other programming languages that have stricter syntactic and semantic requirements.
- *Text stylization.* Markdown provides diverse syntax supports for stylizing the text. Specifically, it treats asterisks (`*`) and underscores (`_`) as indicators of emphasis of the enclosed text. A pair of asterisks or underscores (*e.g.*, `'*text emphasis *'`) represents *text emphasis*. A pair of double asterisks or underscores (*e.g.*, `'__text strong__'`) denotes **text strong (bold)**. In addition, a pair of triple asterisks or underscores (*e.g.*, `'___text strong emphasis___'`) indicates ***text strong (bold) emphasis***.
- *Links.* Markdown allows inserting links with the format of `'[demo](url 'title')'`. The `[demo]` is the link label and is the formatted preview text. The `(url 'title')` is the link target that directs to either internal or external resources specified by the URL (`url`). The `'title'` is the link title attribute that provides additional information about the link.
- *HTML blocks.* Markdown documents use HTML blocks to insert raw HTML contents. The HTML blocks are enclosed with start and end conditions. The conditions are composed with different tags. There are commonly used ones like `'<script> </script>'`, `'<style> </style>'`, and `'<!-- -->'`. In addition, there are other legitimate but uncommon syntaxes to form HTML blocks. For instance, `'<![CDATA[ ]]>'` encloses an original HTML CDATA section; `'<? ?>'` includes other elements like PHP code; and `'<!A >'` includes customized HTML blocks where `A` can be any uppercase ASCII letter.

B. Markdown Compilers

Markdown compilers take valid Markdown strings as inputs and output formatted text, *e.g.*, HTML documents. The Markdown syntax tokens in the input strings are interpreted into the ones in the target format. Markdown compilers generally take three steps to analyze the input strings: (1) Markdown compilers scan the input strings and group the characters into tokens in the *lexical analysis*; (2) they then analyze the tokens for their syntactical and semantic meanings in the *syntax and semantic analysis*; (3) based on the results of the previous step, they further produce the final code (*e.g.*, HTML) in the *code generation* step. Unlike other compilers (*e.g.*, GCC), Markdown compilers usually do not have the intermediate code optimization step since Markdown is a markup language.

Markdown is a typical language with context-sensitive features [6]. For the same token, different behaviors can be exhibited depending on the analysis context (*i.e.*, the other tokens in the same input). To handle such features, Markdown compilers have to record the relevant compilation information as the context to determine how to process a token. Sometimes, due to the nature of the Markdown language design, several possible choices (options) are applicable for interpreting a token in a given context. For example, the token of double asterisks (`**`) can possibly be the open delimiter of text bold syntax or just double asterisks in plain text.

To support such features, modern compilers search in multiple possible options with a default order. A previously chosen option could become invalid when more information is collected along with the analysis on the input strings. Therefore, the compilers usually take the strategy of *backtracking* to explore a different choice. The compilers continue this strategy till a correct option is ultimately determined. For example, when analyzing the input string `'**text bold'`, Markdown compilers usually first prioritize the choice of open delimiter of text bold syntax for the token (`**`). When no corresponding close delimiter for the token can be found, the compilers have to backtrack and try other options for the token (`**`), *e.g.*, plain text. Since any sequence of characters in Markdown is valid, plain text is the default last option in the compilers for the tokens.

TABLE I: 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

C. 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, 15, 16]. 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 [45].

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 [24]. For example, by specially crafting inputs to trigger a performance bug, the server's resources (*e.g.*, memory and CPU) could be exhausted. In the past, performance bugs have caused several publicized failures, causing many software projects to be abandoned [15, 26].

III. Understanding Performance Bugs in Markdown Compilers

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

A. 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 [23] are two high-performance Markdown compilers written in C; cmark is ported and extended for GitHub [10]. commonmark.js [5] and markdown-it [21] 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.

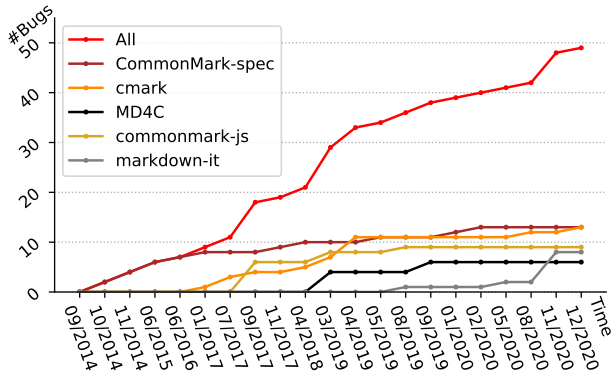

Fig. 1: The number of performance bug reports over time from October 2014 to December 2020.

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 I. 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.

B. Bug Disclosure Methods

Detecting and disclosing performance bugs is an important security and software engineering task. We investigate how performance bugs in Markdown compilers are usually detected and exposed by analyzing bug reports and relevant online discussions. We observe that manual analysis has been the dominant approach to hunting performance bugs in Markdown compilers. Some bug reports suggested that the security analysts found the bugs by crafting special test inputs according to the CommonMark specification [1]. Specifically, 44 out of the 49 performance bugs were identified and reported through manual analysis or testing. The other 5 performance bugs were detected by automated tools like OSS-fuzz [13]. Their corresponding reports usually attached proof-of-concept (PoC) exploits for reproducing the bug and pinpointing the actual root cause. This motivates us to develop better automated techniques to detect such bugs.

C. 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 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 between April 2018 and December 2020 (inclusive, 22 months); 21 (42.86%) bugs were reported between August 2014 and 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 took to release a patch since a bug was 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 [25], or they have not been patched yet [34]. We find that the average duration for patching 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.

D. 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: *Worst-cases in super-linear algorithms.* Some normal algorithms implemented in Markdown compilers have super-linear worst-case complexity [18, 37]. 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 §II-B, 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 is prone to introducing 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 the 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: *Unoptimized code.* Some unoptimized 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 unoptimized code. Unlike the algorithms in R1, such performance issues could be addressed by optimizing the unoptimized code. However, each problem needs to be separately analyzed and fixed, which could be time-consuming. We next discuss an example of such unoptimized 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 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: *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.

E. 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 with 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.

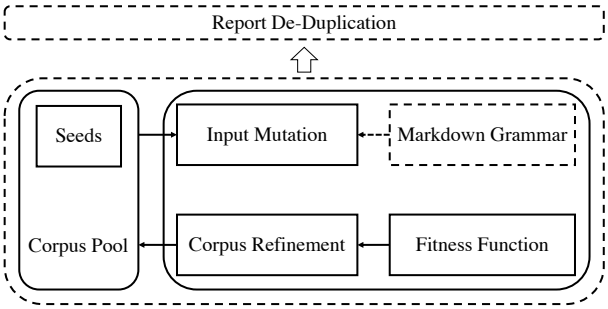

Fig. 2: The architecture of MDPERFFUZZ.

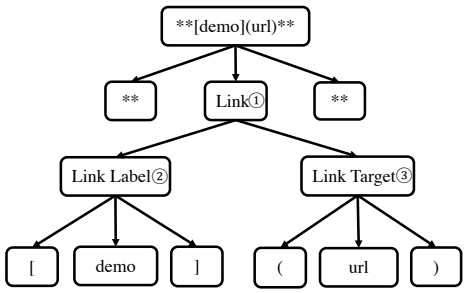

Fig. 3: AST of the statement `'**[demo](url)**'`.

IV. MDPERFFUZZ

Though we have characterized known performance bugs, it is unclear if there exist many unknown performance bugs in the Markdown compilers and the related applications. Therefore, we try to detect unknown performance bugs in real-world Markdown compilers. 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 [28, 29], we propose to use dynamic fuzz testing to detect and exploit performance bugs. To do so, we face two technical challenges. First, generating Markdown documents to test Markdown compilers and exploit the performance bugs (if any) is naturally difficult because of the huge document search space. Prior fuzzers [18, 37] are not very efficient in generating the specially formatted inputs to trigger the performance bugs in Markdown compilers (which we will discuss in §V-C). Second, 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, 37] 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 [17]) when the bugs are triggered, whereas one *performance bug* can potentially exhibit different memory footprints.

We overcome these challenges with MDPERFFUZZ. The overall methodology is depicted in Figure 2. MDPERFFUZZ follows the general fuzzing workflow and is built on top of AFL [48]. Inside the main fuzzing loop, we particularly design a grammar-aware syntax-tree based mutation strategy to efficiently generate high-quality inputs (§IV-A). The mutation strategy can preserve the syntaxes in the inputs to well exercise the Markdown compilers. 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 (§IV-B). The fitness function considers both code coverage and resource usage. To report only unique bugs, we compute the cosine similarity between each pair of the vector representations of the execution traces in bug reports and group highly similar reports as duplicate ones (§IV-C). We then present the implementation details (§IV-D).

A. Syntax-Tree Based Mutation Strategy

Above the default mutation strategies of AFL (*e.g.*, bit flipping), MDPERFFUZZ introduces a syntax-tree based mutation strategy to better preserve the Markdown syntaxes. Our syntax-tree based mutation strategy parses a test case in the corpus into an abstract syntax tree (AST) given the Markdown language grammar. It then traverses the whole AST and randomly replaces several subtrees ($tree_{src}$) with other subtrees ($tree_{dst}$) for different syntaxes. We construct the simplest ASTs each representing a Markdown syntax and include them as distinct candidates of $tree_{dst}$. We do not consider the combination of multiple Markdown syntaxes when constructing one candidate of $tree_{dst}$ because it can be achieved via replacing multiple $tree_{src}$. In this way, compared to mutation strategies that randomly flipping bits, our strategy preserves and extends the syntax of the original test case. Thus it can efficiently construct syntactically correct new test cases from the modified ASTs for later testing.

For example, given a test case of `'**[demo](url)**'`, we first parse it into the AST shown in Figure 3. We identify the basic subtrees (*i.e.*, ①, ②, ③) and randomly replace each of them to generate new test cases. For instance, we can replace the whole link (①) with an inline code span and produce a test case of `'**`random`**'`. The newly generated test case remains the text bold syntax but also exercises new syntax features. It is worth noting that the default mutation strategies of AFL can also be applied when the syntax-tree based one fails to parse a test case.

B. Fitness Function and Performance Bug Detection

MDPERFFUZZ 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 [13, 17, 36], the coverage feedback drives MDPERFFUZZ 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 [37]. The CFG edge hits, standing for the times a CFG edge is visited under a test case, enables MDPERFFUZZ to explore *computationally expensive* paths. As stated in prior work [18], many programs (*e.g.*, PHP hash functions [18, 38]) do have non-convex performance space. We thus do not use the number of executed instructions to guide MDPERFFUZZ because it might fail to find the performance issues caused by local maxima. Therefore, as in the state-of-the-art work, PerfFuzz [18], we design MDPERFFUZZ to favor those test cases that maximize certain CFG edge hits to better detect performance bugs. In this way, MDPERFFUZZ tends to select test cases to either trigger new code or exhaust certain CFG edges. Note that we do not use run-time CPU usage or concrete execution time as the metric, because they show large variations affected by many uncontrollable factors, such as the fuzzer's concurrent features and the characteristics of the applications being tested.

We design a statistical model to accurately identify performance bugs. Our statistical model first obtains the normal program execution behaviors, which help label abnormal ones as performance bugs. In particular, as in [18], we compute the total execution path length—the sum of the CFG edge hits—under a test case as the metric. 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 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 normally no more than $1/k^2$. Since only in rare cases would the execution path length significantly deviate from the normal situations, 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).

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

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

C. Report De-Duplication

Though the execution path lengths under different test cases could all meet Equation 2, they could actually trigger the same performance bug. De-duplicating the bug reports is necessary for a more precise result, whereas prior works [18, 37] 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 stacks might not be accurate enough to represent unique bugs. This is because the exactly critical call stack for a performance bug can hardly be accurately exposed. Unlike memory corruption bugs that have deterministic call stacks when the bugs are 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 compute the cosine similarity between their edge-hit vectors (*e.g.*, $\vec{v}$, $\vec{v'}$), 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 between their corresponding edge-hit vectors exceeds a threshold.

$$cosine(\vec{v}, \vec{v'}) = \frac{\vec{v} \cdot \vec{v'}}{|\vec{v}||\vec{v'}|} = \frac{\sum_{i=1}^{n} c_i c_i'}{\sqrt{\sum_{i=1}^{n} c_i^2}\sqrt{\sum_{i=1}^{n} c_i'^2}} \tag{3}$$

TABLE II: Bug detection results of the Markdown compilers in C. 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.	# Con.
cmark (0.29.0)	C	1321	7	4
MD4C (0.4.7)	C	239	3	0
cmark-gfm (0.29.0)	C	981	4	3

D. Implementation

We implemented the fuzzing part of MDPERFFUZZ on top of an AFL-based fuzzer, PerfFuzz [18]. Specifically, we compiled a simplified Markdown grammar via ANTLR4 [35] into a Markdown parser; then we introduced the syntax-tree based mutation strategy as an extension that can be flexibly plugged in; 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.

V. DETECTING PERFORMANCE BUGS VIA MDPERFFUZZ

In this section, we investigate the prevalence of performance bugs in the wild. We apply MDPERFFUZZ to detect performance bugs in several mainstream Markdown compilers.

A. Experimental Setup

Dataset. Since MDPERFFUZZ employs an AFL-based fuzzer, it is only capable to analyze Markdown compilers implemented in C/C++. Therefore, we select all the 3 Markdown compilers in C in the recommended implementation list of CommonMark specification [7] and list them in the first column of Table II.

Experiments. Each Markdown compiler is first instrumented using our enhanced C compiler. We then apply MDPERFFUZZ to detect performance bugs on the instrumented Markdown compilers. We apply the PoCs collected in §III as the initial seeds and configure MDPERFFUZZ to use a single process, a timeout of 6 hours, and an input size of 200 bytes. 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.

B. Results

We present the performance bug detection results in Table II[1]. Duplicate performance bug reports are naturally

[1]We make the raw data available for review at https://drive.google.com/drive/folders/1RUeASu9MmDoJDyEwNzlOR0_VhIDuLD0n?usp=sharing.

common during fuzzing. The fuzzing part of MDPERFFUZZ 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 Markdown compilers by from 2.31× to 7.28× compared to normal-performance cases.

We further manually check the reports to validate the performance bugs. Since MDPERFFUZZ limits the input size like in other works [13, 18, 37] due to the concerns of large search space, our manual analysis attempts to identify the severity of the performance slowdown in more realistic scenarios, *e.g.*, larger input sizes of thousands of characters. 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 Markdown 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 performance as one of their main focuses during the development [23]. Therefore, the performance bugs could be avoided with domain knowledge and special care, which are often difficult for most developers.

We have investigated those false-positive cases to understand the reasons. We find that the unique buggy cases reported by MDPERFFUZZ indeed triggered performance issues, *i.e.*, those test cases led to longer execution paths. However, the larger attack inputs we constructed manually did not manifest such performance issues. As we discussed in §III-E, developers might choose to patch the performance bugs by enforcing certain limits (*e.g.*, **P1**). Such a strategy guarantees that there is no performance issue in large-size test cases; small-size test cases, however, can still trigger worst-case behaviors. As mentioned above, MDPERFFUZZ can only detect performance bugs with small-size test cases (*e.g.*, hundreds of bytes) due to the inherent search space concerns. As a result, all the false positives were reported by using small-size inputs.

C. Comparison

We compare MDPERFFUZZ with two state-of-the-art works, SlowFuzz [37] and PerfFuzz [18]. MDPERFFUZZ and PerfFuzz are implemented above AFL whereas SlowFuzz is built on top of libFuzzer [20]. 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 [20]. Nevertheless, we evaluate all the tools with the same dataset in Table II and run them for the same amount of time (6 hours), and the same input size (200 bytes) 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. MDPERFFUZZ and PerfFuzz—AFL-based fuzzers—do not suffer from this problem.

The results show that MDPERFFUZZ outperformed PerfFuzz and SlowFuzz by detecting 3 and 5 more performance bugs, respectively. 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.

TABLE III: The performance slowdown and code coverage of MDPERFFUZZ, PerfFuzz [18], and SlowFuzz [37]. 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.
MDPERFFUZZ	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%

1) Performance Slowdown: Table III shows the performance slowdown caused by the inputs generated by MDPERFFUZZ, PerfFuzz, and SlowFuzz. We use the maximum execution path length as the performance metric, and normalize the performance slowdown using a baseline obtained from the random normal-performance cases. We notice that, though all tools caused performance slowdown on the testing applications, MDPERFFUZZ achieved a 14.71% higher average best performance slowdown over PerfFuzz, and 41.21% over SlowFuzz. Furthermore, we observe that MDPERFFUZZ 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, MDPERFFUZZ took 3.2 hours, whereas PerfFuzz and SlowFuzz used 3.9 hours and 6.0 hours, respectively. This demonstrates the high efficacy of MDPERFFUZZ in detecting performance bugs.

2) Code Coverage: We also evaluate the code coverage each tool achieves to measure the efficacy of our syntax-tree based mutation strategy. We collect the test cases generated by each tool and run on afl-cov [27], 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 III. The syntax-tree based mutation strategy of MDPERFFUZZ was effective in reaching high code coverage. It enabled MDPERFFUZZ to visit 67.97% of lines of code and 61.79% of functions on average. MDPERFFUZZ achieved *higher code coverage* than PerfFuzz and SlowFuzz in *all* testing software. In the Markdown compilers, MDPERFFUZZ outperformed PerfFuzz by 20.75% more lines of code and 13.17% more functions; MDPERFFUZZ outperformed SlowFuzz by 28.68% more lines of code and 19.80% more functions. With the mutation strategy, MDPERFFUZZ successfully fuzzed 8.02% of lines of code and 11.39% of functions that were not ever

TABLE IV: Evaluation results on other Markdown compilers and apps. ∗ denotes the security mode of the compiler is on.

	Software	Lang.	# Bugs
Other Markdown compilers	commonmark.js (0.29.3)	JS	7
	markdown-it (12.0.4)	JS	2
	marked (1.2.7)	JS	25
	Snarkdown (2.0.0)	JS	13
	commonmark-java (0.17.0)	Java	6
	flexmark-java (0.62.2)	Java	15
	commonmark.py (0.9.1)	Python	27
	php-commonmark (1.5.7)	PHP	19
	php-commonmark* (1.5.7)	PHP	0
	Parsedown (1.7.4)	PHP	8
	Parsedown* (1.7.4)	PHP	8
	php-markdown (1.2.8)	PHP	6
	markdown-go	Go	13
	Comrak (0.9.0)	Rust	11
	StackEdit	JS	9
	DILLINGE	JS	7
Apps	GitLab (13.7.3)	Ruby	6
	BitBucket (7.9.1)	Java	8
	Hugo (0.74.3)	Go	13
	Hexo (5.2)	JS	14

visited by other tools. As a result, 2 new performance bugs were identified within this proportion of code.

Summary. MDPERFFUZZ outperformed the state-of-the-art works by detecting *more* performance bugs, achieving *better* performance slowdown, and covering *more* code.

VI. STUDYING PERFORMANCE BUGS IN MORE MARKDOWN COMPILERS

Many Markdown compilers are implemented in languages other than C/C++, and they are not supported by MDPERFFUZZ and other AFL-based fuzzers. To understand if and how these Markdown compilers suffer from performance bugs, we construct an extensive dataset and utilize the exploits generated by MDPERFFUZZ in §V to detect potential bugs in them.

A. Methodology

Dataset. We construct a comprehensive dataset in Table IV, including a set of *other Markdown compilers* written not in C/C++ and another set of relevant real-world *applications*. We try to include popular Markdown compilers implemented in diverse programming languages to understand the effects of programming languages on performance bugs (if any). In particular, our dataset covers Markdown compilers written in Java, JavaScript, PHP, Python, Go, and Rust. We also include the first two Google search results (StackEdit and DILLINGE) in January 2021 into our dataset. StackEdit is also in the suggested application list for opening Markdown documents in Google Drive. We notice that some compilers (php-commonmark and Parsedown) provide options to enable additional security mode to mitigate certain bugs. We are interested in the effects of such security options, thus we present them separately with an asterisk suffix (*). Regarding real-world applications, we try to cover three main uses of Markdown compilers: (1) Markdown document rendering in code hosting software (*e.g.*, GitLab, and BitBucket); and (2) static web page generation frameworks (*e.g.*, Hugo and Hexo).

We downloaded the latest stable version of each Markdown compiler from its official website or GitHub repository in January 2021. We denote their actual software versions in the parenthesis if applicable. We install and configure them with the default settings.

Experiments. Since MDPERFFUZZ is not capable to detect performance bugs in the dataset in Table IV, we first collect the exploits generated from MDPERFFUZZ in §V and summarize them into 45 unique attack patterns. Each pattern exploits one Markdown syntax feature. We then apply them to evaluate the software in a black-box manner. Such an approach is practical and scalable, and enables us to analyze a diverse set of compilers.

We use the environment as in §V-A to test standalone Markdown compilers. For the software in the *application* category, we empirically identify the entry points for triggering the Markdown compiler components (*e.g.*, command-line API or UI operations). For those that work in the server-client model and allow self-hosting (*i.e.*, GitLab and BitBucket), we deploy them on a computer running Debian GNU/Linux 9.12 with a 4-core Intel Xeon CPU and 16GB RAM. We use another computer in the same local area network as the client to send requests and measure the network response time, client-side CPU time, and server-side CPU time after the client issues a request. We use such results to detect performance bugs and further understand whether the performance bugs appear in the server or the client.

It is hard to instrument the Markdown compilers implemented in diverse programming languages and the Markdown components in complicated software. Hence we currently are unable to de-duplicate the reports for the software in the dataset. Nevertheless, as our test cases especially exploit distinct Markdown syntax features, we believe they are most likely to trigger different performance bugs. We will further discuss it in §VIII.

B. Results

We present the bug detection results in the last column of Table IV. The performance bugs in Markdown compilers are prevalent and might have been overlooked by the compiler developers. In particular, we successfully identified 168 performance bugs in the category of *other Markdown compilers*. We can observe that the number of detected performance bugs varies significantly among Markdown compilers. Some Markdown compilers were particularly vulnerable to performance bugs, whereas some did not have any performance issues. For instance, we detected 27 performance bugs in commonmark.py but identified only 2 bugs in markdown-it. We do not observe a distinguishable difference among programming languages in terms of the number of bugs. The performance bugs could substantially impact end-user experiences. For example, the performance bugs in StackEdit and DILLINGE could lead to data loss once the browser tab was unresponsive or was forcibly killed.

The Markdown compilers in popular applications are also vulnerable to performance bugs. We successfully detected 41

performance bugs—28 on the client-side and 13 on the server-side. In particular, GitLab and BitBucket suffered from server-side performance bugs, which could be exploited to significantly degrade the server performance. They can be exploited for launching DoS attacks (see §VII-A for more details).

Responsibly disclosing the bugs can greatly benefit the software users and the whole community. We are in the process of contacting the maintainers of the buggy Markdown compilers and reporting the newly detected performance bugs. At the time of writing, 24 performance bugs have been acknowledged.

C. Effects of Security Mode

php-commonmark and Parsedown introduce a security mode to mitigate certain bugs. Our results show that their security modes have different effects. As shown in Table IV, php-commonmark in its security mode (shown as php-commonmark*) was not vulnerable to any performance bugs, whereas the security mode of Parsedown (shown as Parsedown*) did not mitigate any performance bugs.

By reading the relevant documents and the source code, we find that the security mode of Parsedown mainly mitigates cross-site scripting (XSS) bugs but does not consider performance related issues. The security mode of php-commonmark, on the other hand, applies several strategies to mitigate performance bugs. For instance, it sets a threshold to limit the depth of nested structures, escapes HTML blocks in Markdown inputs, and disallows unsafe links. These strategies together could successfully mitigate all 19 performance bugs identified in its default mode. We will further discuss the countermeasures against performance bugs in §VIII.

D. Other Types of Bugs

Though we mainly focused on performance bugs, we also detected memory corruptions in more than five Markdown compilers in our research. In particular, the Markdown compilers exhibited crashes when we fed them the test cases. The crashes happened in Markdown compilers implemented in JavaScript, Java, and Python. For example, we particularly analyzed one crash in Snarkdown, and found the maximum call stack size was exceeded when using recursive function calls to process one type of our testing inputs. There were other memory errors (*e.g.*, segmentation faults) when our test cases triggered some illegal memory read or write.

We also detected several unexpected errors in the *application* category. We observed that GitLab and BitBucket could return HTTP 500 internal server errors when the test cases contained special Unicode characters, possibly because their Markdown compilers currently did not support compiling special Unicode characters. Though such errors usually affect only the user who sends documents containing such characters, they still lead to bad user experiences and shall be fixed.

VII. Impacts of Performance Bugs

In this section, we further analyze the detected performance bugs and demonstrate their security impacts.

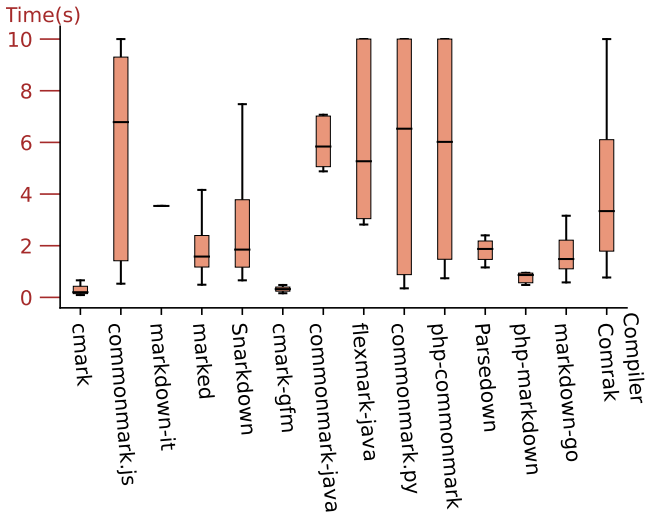

Fig. 4: Compilation time of Markdown compilers under attack inputs of the size of 50,000 characters and a 10-second threshold. The middle lines in the boxes represent the corresponding median values.

A. Impacts on Performance

To better understand the performance degradation caused by performance bugs, we depict in the salmon boxes in Figure 4 the compilation time when the performance bugs are triggered by attack inputs of size 50,000 characters. We use such a size as it can roughly represent the normal uses of Markdown compilers. The results show that performance bugs can cause significant performance degradation. In general, our attack inputs successfully exploited the performance bugs by causing over 3-second compilation time. Different performance bugs could result in different levels of performance degradation in a compiler. For example, the compilation time of commonmark.js ranged from 2 seconds up to 10 seconds (the threshold) due to performance bugs. If a threshold was not set, the exploits would even cause the compilers to run for *several hours*.

The performance bugs can potentially affect many users if they reside in server-side applications. Specifically, we present two case studies about GitLab—a self-hosted code hosting software, and Parsedown—a popular Markdown compiler module in PHP. We deploy the latest version of GitLab with its default NGINX web server; we develop a server-side PHP application that calls Parsedown to compile user-provided Markdown documents. We randomly choose a common attack input that can trigger performance issues in both GitLab and Parsedown. We then test the applications with different numbers of concurrent attacks (requests). We try at most 8 concurrent requests because our server has only 8 logical CPU cores.

We depict the server CPU usage under the attacks in Figure 5. We clearly observe the increase of CPU usage when more concurrent attack requests were issued. In particular, when 8 requests were sent, the server CPU usage promptly reached almost 100%. Therefore, an attacker can send only a few attack requests at a very low rate to significantly degrade the performance of a vulnerable server application, making it unable to responsively serve other legitimate user requests.

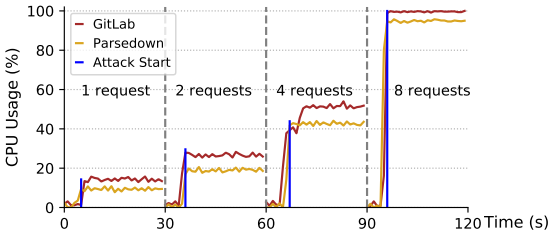

Fig. 5: Server-side CPU usage over time under attacks.

B. Impacts on End-Users

We are interested in how many websites and users the performance bugs can potentially influence. All the 17 Markdown standalone compilers combined have around 70K stars and forks. Roughly, millions of websites are vulnerable to performance bugs from download statistics of some Markdown compilers in our dataset. For example, the three Node.js packages—commonmark.js, marked, and markdown-it, receive 874,974 [30], 8,346,571 [32], and 18,392,877 [31] monthly downloads from March to April 2021, respectively. Regarding only one month of these three compilers, over 27 million installations could be potentially affected by the performance bugs if the bugs persisted in the recent month. Therefore, considering all the vulnerable Markdown compilers, we could estimate in certain confidence that millions of websites and their users can be potentially influenced by the detected performance bugs. The millions of users of applications like GitLab and BitBucket [12, 47] could also be affected by the performance bugs.

VIII. Discussion and Future Work

Mitigating performance bugs. Practical mitigation and defense techniques against performance bugs are necessary for protecting vulnerable Markdown compilers and applications. We have shown that the context-sensitive feature handlers in Markdown compilers could be abused by attackers in §III-D. Several security strategies used in the security mode of php-commonmark (*e.g.*, enforcing the limits, escaping the HTML blocks, and disallowing unsafe links) are shown to be effective in mitigating performance bugs. However, they can break some functionalities, especially those related to the context-sensitive features. For example, some HTML blocks cannot be compiled as expected because of the security strategies. Longer legitimate Markdown documents might also be blocked because of the limits. A trade-off has to be made to balance functionality and security. In the future, we hope to port such mechanisms to mitigate attacks exploiting performance bugs in Markdown compilers.

Report de-duplication. MDPERFFUZZ uses an execution trace similarity comparison method to de-duplicate performance bugs. Theoretically, the method can be applied to software implemented in diverse programming languages once the runtime CFG edge hit information is available. However, to the best of our knowledge, not all programming languages have available instrumentation tools exactly for such a purpose. It is also time-consuming and even infeasible to develop our own instrumentation tools within this work. We thus choose to not apply the method for the evaluation in §VI. In the future, we plan to further investigate the feasibility of a language-agnostic method for de-duplicating performance bugs. For example, we plan to explore transforming the software into certain intermediate representations (IRs) and obtain the necessary information from IRs to de-duplicate performance bugs.

Portability. Based on AFL, MDPERFFUZZ can only perform gray-box fuzzing on C/C++ Markdown compilers. However, we have demonstrated the exploits could be used to trigger performance bugs in other Markdown compilers. Besides, our syntax-tree based mutation strategy is effective for generating high-quality test cases, thus it can improve existing fuzzing techniques even for other types of bugs in Markdown compilers. For example, MDPERFFUZZ is naturally capable of detecting memory corruptions like out-of-bound writes. Furthermore, we can flexibly plugin other language grammars to detect bugs in other compilers.

IX. 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 [15, 49], mobile platform [19], and the web server end [15], *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 [49]. However, there is little understanding of performance bugs in the Markdown compilers. Sub *et al.* studied GCC and LLVM compilers but they did not focus on the performance issues [41]. Our work investigates an understudied problem—performance bugs in the Markdown compilers. We characterize the bug patterns and reveal the close relationship between the performance bugs and the context-sensitive Markdown syntaxes.

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, 14, 24], algorithmic complexity DoS vulnerabilities [8, 40], and other general performance issues [15, 19]. Static methods analyze the source code of the applications and diagnose vulnerable bug patterns, for example, repeated loops [28, 29]. Dynamic methods are also applied to identify performance bugs. SlowFuzz [37], PerfFuzz [18], and HotFuzz [2] propose new fuzzing solutions to detect the worst-case algorithmic complexity vulnerabilities. Our work improves existing solutions via a syntax-tree based mutation strategy and a new report de-duplication method. Our evaluation has well demonstrated its efficacy.

X. Conclusion

Performance bugs in Markdown compilers were previously understudied. This paper conducted a systematic study to understand the characteristics of such bugs. We developed MDPERFFUZZ with several new techniques to detect performance bugs in the wild. We successfully identified many new performance bugs in Markdown compilers and applications. We demonstrate the performance bugs in Markdown compilers are not only prevalent but also severe. We hope that our study could shed some light on future Markdown compiler development and performance bug detection.

References

[1] andersk. Parsing «««... takes quadratic time, 2020. https://github.com/markdown-it/markdown-it/issues/737.

[2] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele. 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, Feb. 2020.

[3] V. Chipounov, V. Kuznetsov, and G. Candea. 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, Mar. 2011.

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

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

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

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

[8] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[9] V. Durcekova, L. Schwartz, and N. Shahmehri. Sophisticated denial of service attacks aimed at application layer. In *2012 ELEKTRO*. IEEE, 2012.

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

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

[12] GitLab. Is it any good?, 2021. https://about.gitlab.com/is-it-any-good/.

[13] Google. Oss-fuzz, 2021. https://google.github.io/oss-fuzz/.

[14] H. H. Jazi, H. Gonzalez, N. Stakhanova, and A. A. Ghorbani. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Computer Networks*, 2017.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, June 2012.

[16] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. 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, Nov. 2010.

[17] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[18] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: automatically generating pathological inputs. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, July 2018.

[19] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May–June 2014.

[20] LLVM. libfuzzer, 2021. https://hammer-vlsi.readthedocs.io/en/stable/LibFuzzer.html.

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

[22] marked. Marked, 2021. https://marked.js.org/.

[23] md4c. md4c, 2021. https://github.com/mity/md4c.

[24] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee. Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

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

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

[27] mrash. afl-cov - afl fuzzing code coverage, 2021. https://github.com/mrash/afl-cov.

[28] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[29] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, Florence, Italy, May 2015.

[30] npm stat. Download statistics for package commonmark, 2021. https://npm-stat.com/charts.html?package=commonmark&from=2021-03-05&to=2021-04-05.

[31] npm stat. Download statistics for package commonmark, 2021. https://npm-stat.com/charts.html?package=markdown-it&from=2021-03-05&to=2021-04-05.

[32] npm stat. Download statistics for package marked, 2021. https://npm-stat.com/charts.html?package=marked&from=2021-03-05&to=2021-04-05.

[33] npm stat. Download statistics for package commonmark, 2021. https://npm-stat.com/charts.html?package=marked&from=2016-03-05&to=2021-04-05.

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

[35] T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 1995.

[36] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[37] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[38] PHP. Php hash, 2021. https://www.php.net/manual/en/function.hash.php.

[39] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu. Rescue: Crafting regular expression dos attacks. In *Proceedings of the 33rd IEEE/ACM International Conference on Auto-*

mated Software Engineering (ASE), Montpellier, France, Sept. 2018.

[40] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[41] C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 294–305, Saarbrücken, Germany, July 2016.

[42] Swartz and J. Gruber. Markdown, 2014. http://www.aaronsw.com/weblog/001189.

[43] A. D. Thurston and J. R. Cordy. A backtracking lr algorithm for parsing ambiguous context-dependent languages. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, 2006.

[44] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montréal, Canada, May 2019.

[45] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, Seoul, Korea, June–July 2020.

[46] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig. 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, Apr. 2017.

[47] K. Yap. Celebrating 10 million bitbucket cloud registered users, 2019. https://bitbucket.org/blog/celebrating-10-million-bitbucket-cloud-registered-users.

[48] M. Zalewski. american fuzzy lop, 2021. https://github.com/google/AFL.

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