Hunting Garbage Collection Related Concurrency Bugs Through Critical Condition Restoration

Hanhan Zhou, Tian Lan, Guru Venkataramani
{hanhan, tlan,guruv}@gwu.edu

ABSTRACT

With the increasing popularity of multi-core processors and multi-thread languages/frameworks, race conditions – which are non-deterministic by nature – are becoming a main root cause for concurrency bugs. It opens doors to malicious attacks such as remote code execution and denial of service attacks, potentially putting millions of users in danger. Yet, such non-deterministic racing conditions are often difficult to identify or reproduce in standard program testing. In this paper, we focus on the Garbage-Collection (GC) feature, which is known to be a frequent victim of concurrency bugs in many software systems. We develop a new approach to facilitate the testing of GC-related bugs through critical condition restoration. In particular, we propose a risk-score mechanism to quantify the risk of GC-related bugs in target functions and leverage the score to select appropriate testing parameters and garbage generation strategy, with a higher chance of producing the critical condition. Our experimental results show that the proposed approach could significantly improve the probability of finding GC-related bugs (from 0 in condition-oblivious testing to 14.8 bugs identified in our experiment) while incurring only around 26% execution overhead.

CCS CONCEPTS

• Security and privacy → Intrusion/anomaly detection and malware mitigation; **• Software and its engineering → Software testing and debugging**; **• Software and its engineering → Software reliability**;

1 INTRODUCTION

Automatic software testing helps to verify expected behavior of a program and proof it against bugs. However, for multiple reasons, software systems might behave non-deterministically, or in other words, running the same piece of code could produce different results. Except for natural and physical reasons, and apparent randomness reasons such as time and random number generator, system and software level concurrency are another leading cause for such non-deterministic behaviors. Since concurrency related bugs often trigger only under specific context, it is very hard to detect, reproduce and fix such bugs [3]. Existing research on this topic mainly pay attention on data race condition or data hazards [12][15]. For example, [21] [13] and [26] mainly combined static analysis and fuzz testing to find such concurrency vulnerabilities, using the results of static analysis of potential concurrency vulnerability to guide a coverage-based fuzzer, AFL [25] for example, trying to trigger the suspicious vulnerabilities.

However, with the wide use of modern languages like Java, JavaScript and their concurrent garbage collection design, there are still potential concurrency vulnerabilities might be introduced by its non-static (dynamic) behaviors. For example, CVE-2019-5786, CVE-2018-8174 and CVE-2018-4192 in the National Vulnerability Database [17]. Unfortunately, there are not many readily available tools and resources dealing with addressing them primarily due to GC complexity. We note that GC feature is operated on a OS level that is often inaccessible to coverage guided fuzzing for many languages. Furthermore, the GC features cannot be fully controlled by the user- for example in Java, Java Virtual Machine(JVM) has a higher priority than programmer in dealing with GC; even if an explicit GC function call is in the code, Java Virtual Machine might postpone its execution or even ignore it [19]. Even with the control of GC, many vulnerabilities regarding this rely on many conditions, timing of the program execution and the process of GC for example, without which one can hardly reproduce and trigger the bug leading to it.

The goal of this paper is to develop a mechanism for hunting GC-related concurrency bugs through critical condition identification and restoration. As an illustrative example, `Some_array.reverse()` is a one-line code shows a real-world GC-related bugs in JavaScript as illustrated later in Section 2.2, some elements in the array might be prematurely freed, causing a use-after-free scenario, due to a data race between the `array.reverse()` function and a GC process. This vulnerability is only triggered under specific conditions, a maliciously crafted website could lead to an arbitrary code execution utilizing this triggered bug, with continuous trials to increase its probability of triggering the bug. Although extremely hard to reproduce without sophisticated memory manipulation, it affects multiple versions of WebKit, the engine for the widely used web browser Safari and several other Apple products [18]. It took almost 2 years after it was originally introduced to patch this GC-related vulnerability, indicating the challenge and difficulty of finding a GC-related bug [18]. This is mainly due to the fact that such GC-related bugs often manifest only under stringent conditions (e.g., under the racing condition in `array.reverse()`) and are difficult to trigger and reproduce during testing. To demonstrate this, we have conducted experiments to show that following the example above, reversing an array with a length of 50000 for 50000 times could result in positive bug trigger with a probability rate of less than 1%, an array with less length or tried for fewer times than that could never trigger it, this also indicates the limitation for a normal fuzzer on testing a GC-related bug when they can hardly trigger it.

Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The key idea of our approach is that we can analyze and identify the bug-trigger conditions for GC-related vulnerabilities – such as timing, input preconditions and memory allocation – and then guide program testing to not only focus on code segments with a higher probability of containing such vulnerabilities but also recreate the critical bug-trigger conditions through program instrumentation (e.g., employing different garbage generation strategies). More precisely, we consider four hypotheses on recreating critical conditions for GC-related vulnerabilities, empirically validate them using a set of garbage generation policies, and finally propose a novel risk-score mechanism to guide program testing and to trigger GC related bugs more quickly. In particular, a Sensitive Analyser (SA) is developed to leverage the hypotheses and to compute a risk score quantifying the potential likelihood of GC-related concurrency vulnerabilities. The risk score is used to (i) identify high risk functions, so that we can drive program testing to focus more on these risky functions, and (ii) guide the selection of garbage generating strategy and testing parameters to trigger the vulnerabilities more efficiently during testing.

As a result, our experimental results show that the proposed approach could try several times on functions with high risks and gain a significantly improve of probability on finding GC-related bugs, from 0 in condition-oblivious testing to 14.8 bugs identified in our experiment, while incurring only around 26% time overhead.

In summary this work makes the following contributions:

- We present a new approach to find GC-related concurrency vulnerabilities by combining critical condition restoration and program testing.
- A risk score mechanism is developed to identify functions with high probability of GC-related vulnerabilities and to guide the selection of testing strategies and parameters.
- Our experimental results on three selected CVE vulnerabilities shows significantly higher probabilities of triggering the GC-related bugs with mild execution overhead.

2 BACKGROUND

2.1 Garbage Collection

In C/C++, programmers are responsible for memory allocation and de-allocation through creating and destroying objects. Failure to free the memory no longer in use could lead to memory leak and out-of-memory errors. For other modern languages like Java, Python and JavaScript, programmers do not have to free the memory they allocated, due to the use of an automatic garbage collector. Advantages for automatic garbage collection are obvious: programmers no longer have to worry about freeing objects/pointers, thus reducing the likelihood of related bugs such as double-free. However, the introduction of the GC feature requires extra memory operations and sometimes results in a period known as "stop-the-world" [20] [9], meaning that all process have to stop and wait for GC to complete, causing stalls and delay. Modern programming languages strive to minimize this "stop-the-world" period, e.g., through concurrent, increment and real-time garbage collection, although each of these techniques brings new design trade-offs. The garbage collector works by identifying reachability of the objects, i.e., when an object becomes unreachable, it shall be freed in the next round of garbage collection. However, while there are function calls available for garbage collection, for many languages like Java, calling these functions does not guarantee the timing of the next GC execution. As a result, the behavior of GC feature is often highly non-deterministic.

Mark-and-sweep is a classic algorithm for garbage collection proposed to solve the " cyclic reference" issue from the reference counting algorithm, where a series of references forms a loop, making them all have a nonzero reference count yet might be laying on the memory space never used thereafter. The algorithm consists of two stages: mark and sweep, and a GC bit map to record the status for all objects created. For mark phase, garbage collector will go through all objects and mark the reachable ones' corresponding bit on the bitmap to true. Following a search policy like depth first search starting from a certain node could traverse all objects under it and mark every one of them. In JavaScript, for example, all objects are created under one or several GC roots for DFS search in the marking phase. The Sweep phase is to clean up all the unreachable objects and make their memory space available for later object creation. In general, GC would go through the GC bitmap and free all objects whose corresponding bit on the bitmap is marked as false. There have been many implementations to enhance the classic mark-and-sweep, for example, mark-and-compact to solve the memory fragments issue, but most of them would involve 2 phases, mark the live objects and free the rest.

WebKit is the web browser engine used by Safari and many other Apple applications on multi-platforms. Since its preview release 21, a new concurrent GC called Riptide was adopted, aiming to increase the GC throughput and reduce the GC latency. It utilizes modern GC features like parallel marking and generations. By performing parallel marking, it could use up to 8 threads for GC marking phase [22]. For a generational GC, it also adopts the feature of stick mark bits. Considering newly created objects tend to have a shorter time period and are often associated with recently created objects, they are allocated and marked as eden object, those who survived several rounds of GC are then marked as tenure object, and if the memory is far from full, the marking phase might go over eden objects only as a minor GC, comparing with a full GC that scans the whole memory, to save time. It did greatly increase the general performance for the GC, yet for a large project like WebKit with great redundancy, the introducing of it also poses potential risks due to its parallelism feature against compatibility and security on previous codes [10].

Listing 1: Sample execution of array reverse

```
function reverse(arr):
  i=0, j=length(arr)-1
  while i<= j:
    swap(arr[i], arr[j])
    i++, j--
arr=Arrays(500000)
reverse(arr)
```

2.2 Case Study: CVE-2018-4192

To explain how a data race between normal program execution and GC could happen, we take a real-world vulnerability, CVE-2018-4192 [18] from JavaScript Core in WebKit. As illustrated in Listing 1 showing the implementation of reversing an array, the data race between the `array.reverse()` function and the concurrent

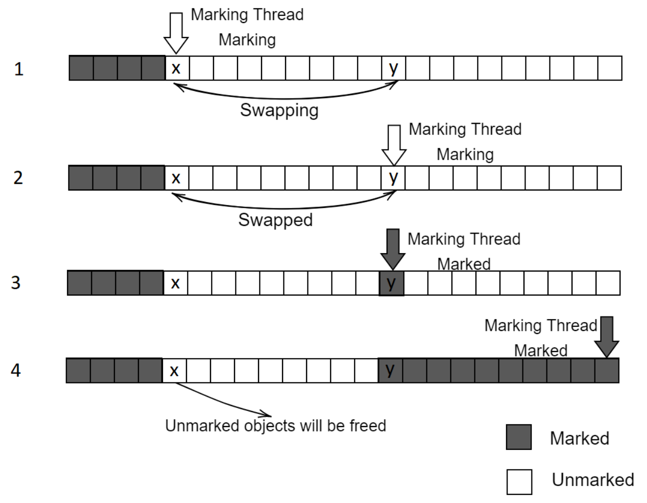

Figure 1: Illustration of CVE-2018-4192. While reversing an array during an on-going GC's marking process, some part of the array getting swapped are not marked, and will be freed later, causing a use-after-free situation.

marking process – if occurring with a certain time – could result in part of the elements not getting marked, thus getting freed later. Since Riptide GC in WebKit does not have the feature for shifting the live objects to eliminating the fragmentation, there will be freed "holes" on memory space available for later new objects allocation. This could lead to a use-after-free scenario where malicious crafted contents could be allocated at desired place and ultimately having this vulnerability leveraged to arbitrary remote code execution. A simplified example, as illustrated in the Figure 1, with reversing function at the time of finishing swapping array index x and index y, marking process is about to mark index x, but due to the data race, it accidentally marks index y and continues to mark elements starting index $y+1$. Without getting marked, elements with indexes between x and y will be marked as "garbage" and ready to be freed in the future for new objects allocation. In reality, there could be more than one threads marking the array, so the area of elements affected could be different.

Listing 2: Sample code to reproduce CVE-2018-4192

```
arr=Array(1000).fill([])
async function async_reverse(arr){
  arr.reverse()
}
for(var index = 0;index<rounds;index++){
  arr.map(function(ele,index,arr){
    my_reverse(arr);
  })
}
```

3 METHODOLOGY AND DESIGN

In this section, we present four hypotheses on recreating critical conditions for GC related vulnerabilities, empirically validate them using a set of garbage generation policies, and finally propose a novel risk-score mechanism to guide program testing and to trigger GC related bugs more quickly.

3.1 Hypothesis and Empirical Validation

Our key hypotheses are:

(1) The timing and state of a GC process (with respect to the execution of target code segment) jointly affects the occurrence of GC-related bug. Only under specific conditions such as certain timing and memory locations, GC-related bugs could be triggered during execution.
(2) The timing of a scheduled GC can be affected by the amount and type of "garbage" generated in the system. In particular, large amounts of garbage objects created within a short time could advance a scheduled GC action.
(3) Code fragments consuming higher execution time and performing more memory operations have a higher chance of leading to GC-related bugs. In fact, more memory operations could mean more memory writes/reads and thus a higher risk of race conditions.
(4) Due to non-deterministic nature of the GC feature, repeated execution of the target code (as well as the GC process) can increase the probability of producing racing conditions and eventually triggering GC-related bugs.

To test our hypothesis, we generate garbage objects based on the following policy: We continuously create objects of a doubled size and then modify its pointer, so that the pointer no longer points to itself. These objects then are becoming unreachable and will get garbage collected. There will be two parameters deciding the garbage generation policy, size and frequency: size of the last generated object and the total times of this generation for a single function. Given a desired size of garbage, it will be generated in a way similar to Listing 3.

Listing 3: Garbage Generation Example

```
function one_time_garbage(size):
  g="g"
  times=log(size)
  for(i=0;i<times;i++){
    g=g+g
  }
  g=null
```

For example: a desired size of garbage object of 16 byte, frequency of 30, then times = log(8) = 3, repeat "g+g" for 3 times makes g a string with 8 characters (i.e., 16 bytes). This would result in creating 4 garbage objects with size of 2 ,4, 8 and 16 bytes. Based on the frequency provided, the above action will be carried out 30 times to generate the desired "Garbage" during program testing.

The reasons for generating garbage objects in this way are:

(1) Generating sizes from small to large is a way of trying to cover different sizes of "holes" by doubling incremental sizes, to be found by the AddressSanitizer.
(2) Generating small sizes of "garbage" objects in a short time could trigger a minor GC that looks for newly created objects to recycle.

(3) Generating large sizes of "garbage" objects in a short time could trigger a full GC that scans the whole memory space.
(4) In this way we can try to trigger different types of GC at the desired time, covering different types of objects and different parts of the memory space.
(5) By spraying out numbers of objects could cover more memory space and increase the probability of triggering other types of memory related bugs.

To test our hypothesis, since the `array.reverse()` function (which is used to reverse an array) alone is unlikely to trigger the bug, we consider a test program by making `array.reverse()` an async function to achieve higher probability of reproducing GC-related bugs, similar to how CVE-2018-4192 was originally found by a Grammarly based fuzzer [4]. Async function is the way JavaScript enabling asynchronous, and it will return a JavaScript promise object to be resolved or throw the exceptions after its execution. As listing 2 illustrates, notice that no return value from the async function is saved, so the returned array and its promise object, when going outside the scope, can be considered as garbage objects generated once per function run with a fixed size according to the definition of our method. We choose another 2 different parameter settings manually for the garbage generation policy and compare the results with the original testing program.

Table 1 shows the successful rate of triggering the bug, we compare the results from two different settings with our design and the original test, i.e. without our modification. The two settings of parameters for garbage generation policy are manual selected.

From the results we can find see significant difference between these three results. Both settings significantly affects the successful rate of triggering the bug. With the first setting we can achieve a higher probability of triggering the bug comparing to the original testing programs with only a slightly higher average time. With the second setting we can see the test was done even in a shorter time compared to the original results, this indicates our method is triggering minor GCs, that brings originally scheduled GC ahead of its time and many objects created after that are then skipped from the GC process, thus saves some time. When looking at the max runtime we can also see a big difference, that is the max runtime from our settings are up to 3x the average runtime, while in the original results its less than 2x, this indicates we are able to trigger major GC that costs more time than a minor GC. Not only this shows our method of approach would influence how the bug would be triggered by manipulating the memory space, but it also enables the ability to trigger certain types of GC when needed, depending on the parameters for the garbage generation policy. Repeating those actions above gives us the high probability of triggering the bug. These findings together confirms our previous hypotheses. Meanwhile with both settings we are able to trigger the bug with the array length of 30000, while this bug is never triggered when testing the program without our approach, this also illustrates the need of our design as an add-on for a normal fuzzer.

Table 1: Results for tests with different array lengths and settings. Original: run test program without any modification or instruments. Setting 1: during the executing of the test program, garbage objects up to sizes of 300Kb are generated 30 times, evenly distributed throughout the program process. Setting 2: during the executing of the test program, garbage objects up to sizes of 3000Kb are generated 300 times, evenly distributed throughout the program process. For each test run JavaScript Core is restarted before a new execution, This is the average results from 100 independent runs.

setting	length	successful rate(%)	average runtime	min runtime	max runtime
original	30000	0	1.16	0.80	2.30
	50000	23	2.88	2.28	5.79
	80000	44	9.78	5.89	15.35
Setting 1	30000	5	1.23	0.80	2.75
	50000	30	3.34	2.28	7.61
	80000	56	10.26	5.89	30.43
Setting 2	30000	12	1.36	0.80	2.77
	50000	22	2.86	2.29	7.62
	80000	71	9.70	5.90	30.36

3.2 Sensitive Analysis and Rick-Score Computation

As demonstrated above, parameters for the garbage generating policy are of vital importance to successfully triggering GC-related bugs. In this section we analyze the common situations for certain bugs and propose a heuristic design of Sensitive Analyzer (SA) to help guide the fuzzing process to meet the critical conditions of GC-related bugs. More specifically, we propose a novel risk-score, which (i) quantifies the relative risk of a target function containing GC-related bugs, and (ii) can be used to guide the selection of garbage generating policy parameters to trigger the bugs more quickly. We begin with dividing a target function into three categories as follows:

- Type 1: light memory operations in a short time. This type of functions should be covered with existing techniques, also they have a lower probability related to a non-deterministic bug.
- Type 2: heavy memory operations in a long time. This type of functions has a much higher probability related to a non-deterministic bug. Also, it is harder to find as the condition of triggering it might not be satisfied during the normal fuzzing process. This is the main type our approach is trying to deal with.
- Type 3: light memory operations in a long time. Since there are not too many memory changes, this type of functions has a lower probability of having a race condition, but it is also worth testing them in the new approaches.

From our hypothesis the execution time of memory-related operations plays a significant role in deciding the successful probability of triggering GC-related bugs. To capture the impact of different execution environments, we first run a standard test program and record its execution time T_s. Let T_c be the execution time of the target program code, M the memory write operations in Kbytes, and m the memory space allocated. We propose the following risk-score:

$$R_s = \frac{T_c}{T_s} \times \log\left(\frac{M}{m}\right)$$

It is easy to see that the risk score increases if the execution time of memory-related operations becomes higher, or if more frequent memory operations are performed on a smaller memory space. According to the our hypothesis, the higher the risk score is, the higher chance the function might lead to a non-deterministic GC-related bug.

As an illustrative example, suppose to reproduce [18] we are reversing an array with 30000 empty arrays for 30000 times, the risk score is calculated as, 1 * log(32*30000) = 13.7, reversing an array with 50000 empty arrays for 50000 times the risk score is 4*log(32*50000) =57.1, indicating the latter one has a much higher risk, supposing the empty array takes 32 byes of memory space.

The risk score allows us to identify functions that have a higher chance for GC-related bugs and thus deserves more attention during testing. Finally, we propose an testing policy based on the regression results from over 10 different setting combinations. Based on the risk score $\mathbb{R}_s$, we divide our testing policy into 4 cases:

(1) $\mathbb{R}_s < 10$: Continue to test next function
(2) $10 < \mathbb{R}_s < 35$: repeat current testing for 3 times, with policy parameters (m/5,30)
(3) $35 < \mathbb{R}_s < 60$: repeat current testing for 6 times, with policy parameters (m/8,30) and (m/5,50), each parameter for 3 times.
(4) $\mathbb{R}_s > 60$: repeat current testing for 9 times, with policy parameters (m/8,30), (m/5,50) and (m/3,70), each parameter for 3 times.

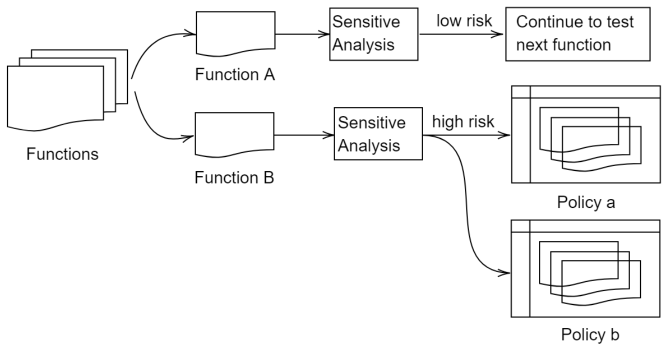

Figure 2: A system diagram of our proposed approach. After the profiling stage, functions with higher risks will be executed and tested more frequently under different polices.

3.3 Program instrumentation

Based on the risk score that is calculated from the sensitive analyzer, those functions with higher risk scores deserve more time during testing to see if they could lead to GC-related non-deterministic bugs. But only repeatedly executing these functions may not be enough, if specific conditions (such as racing condition) with respect to the GC process is not satisfied. To address this, we instrument the target functions to boost garbage generation according to policies determined by our sensitive analyzer, For example, after the initial profiling testing round we found that reversing an array of 50000 elements gives a $\mathbb{R}_s$ = 57.1, in this case, it will be executed 6 more times after its initial execution, with parameters(200Kbytes,30) and (320Kbytes,50), each for 3 times. In this paper as we are working on JavaScript, which is a scripting language, the instrumentation can be implemented as direct lines of code injection to the functions we are testing, for other language a compiler level modification or interruption design might be needed. In the next section we show though this policy is based on empirical evidence, it is yet very effective if combined with a standard fuzzer.

4 EVALUATION

4.1 Environment setup

All experiments are done on a Ubuntu desktop(16.04.5TLS) with 8 cores Intel Core i7-3770 CPU @ 3.4GHz and 16GB RAM. WebKit GTK 2.18.6 [1] debug build compiled with address sanitizer (ASan [8]) is selected as the test platform for JavaScript Core testing on reproducing CVE-2018-4192 and CVE-2018-4233 as a benchmark.

We first construct a target program that performs 12 different functions including CPU- and memory-intensive tasks. Then, for the test cases provided, we implement our risk-score mechanism to quantify the risk of different functions and to select appropriate testing parameters as well as garbage generation strategies. Each test was conducted 10 times to find its average execution time and the average occurrence of GC-related bugs. For each test,the JavaScript Core engine is restarted to clear previous program memory allocation and fragments.

4.2 Testing program

Since our goal is to improve the probability of triggering non-deterministic GC-related bugs, rather than maximizing the code coverage or generating more test cases, we created a synthetic target program with a simple structure covering 12 different types of functions, including CPU- and memory-intensive tasks as well as some known CVE vulnerabilities, to simulate a real-world program and to illustrate the effectiveness of our approach.

The target program consists of 12 functions shown in the section 3.3, including 3 new minimized proof-of-concept CVE functions:

(1) Deep copy of a large object for multiple times.
(2) Generate all permutations of a very large array of strings.
(3) Solving traveling salesman problem with brute force.
(4) Brute force guessing a magic number with random policy.
(5) Sleep for a long time.
(6) Calculating the digits of π with Spigot algorithm.
(7) Find the max value in a very large array.
(8) Perform binary search in a very large sorted array.
(9) Reverse a very large array with `async reverse` function
(10) PoC code for CVE-2018-4192
(11) PoC code for CVE-2018-4233
(12) PoC code for CVE-2017-2491

The target program starts with known inputs to decide which function to execute. All 12 functions have equal chance to be executed from the test cases provided. Since it is the functions built in the JavaScript Core we are testing, in this case, we can execute this program by iterating all the possible inputs to trigger different functions, just like how a standard fuzzer would behave.

We first execute the target program without any instrumentation to simulate a grammar-based fuzzing process that traverses all possible branches without priority in a DFS fashion (denoted by "Original"). Then we run the previous test for 10 times(denoted by

"Repeat") to show the the limitation of repeating tests without a condition restoration. Finally, we implement our risk-score mechanism to selectively (and more frequently) test high-risk functions with the proposed garbage generation strategy and testing parameters. Each of these 3 approaches were executed 10 times (with 10 runs of the target program under all test cases per execution) to obtain the average results.

Table 2: Average results for 3 tests with different settings. Since all possible inputs are known, the code coverage is able to reach 100%, tests using our approach triggered GC-related bugs significantly more times, compared to the standard execution or repeated testing.

	Time(s)	Bugs Triggered	Code Coverage	Functions Executed
Standard	230.23	0	100%	12
Repeat	2493.90	0.3	100%	120
Ours	290.84	14.8	100%	30

As shown in the Table 2, tests using our proposed approach could trigger the GC-related bugs with a significantly higher probability, compared to the two baselines. In fact, 14.8 bugs were triggered per testing with an execution time overhead of 26% on average, while only 0 and 0.3 bugs were triggered by the baseline approaches, respectively. The benefits mainly come from the intelligent selection of risky functions as well as garbage generation strategy and testing parameters. For example, the long sleep function comes with a risk score of nearly 0 and it will not be tested for a second time under our policy. Also by the version of WebKit GTK2.18.6, CVE-2017-2491 is already patched and would not be triggered, but our design also spent several runs on it. Some of the functions with long execution time and memory footprint like are also tested repeatedly although there were no bugs found. These results show that our approach has the ability to (i) identify functions with a higher chance of GC-related bugs through risk score and (ii) create the critical conditions for triggering the bugs during testing. It enables our approach to trigger more times of the two CVE bugs compared to the two baselines that are oblivious to the critical conditions.

In summary, certain types of non-deterministic GC-related bugs are different to trigger in standard testing that is oblivious of the critical conditions, even if the target problem is executed many times. Analyze the condition needed for these bugs allows us to focus on the risky functions and to recreate such critical conditions, which would greatly increase the chance of reproducing the bugs during testing. Our experiments validate the effectiveness of our proposed approach in guiding a testing process to find GC-related non-deterministic more quickly.

4.3 Discussions on Future Work

As a proof-of-concept, our approach is evaluated on a synthetic target program with test cases provided in advance. It would be interesting to fully integrate our approach (as an add-on) with a practical fuzzer like AFL. For path sensitive designs like [5], our approach can provide a feedback for paths with higher chances of finding a GC-related vulnerability, and thus effective guide it to creating a prioritized path exploration strategy.

Our approach focuses on GC-related bugs with a known critical condition of long execution time and intensive memory operations, while other factors such as the type of GC operations, e.g., full and partial GC, could also affect the probability of triggering bugs and can be integrated into our risk score mechanism. Also, our strategy for selecting testing parameters and garbage objects generation are based on linear regression with empirical experiment data. More sophisticated data-driven methods, such as machine learning or reinforcement learning may lead to better parameters or policies, and open doors to possible future work.

5 RELATED WORK

Grammarly based Fuzz testing. Current coverage guided fuzzers, for example the famous AFL [25], when dealing with programs with structured input that requires specific grammars it is limited due to its fuzzing policy. Many of its generated inputs would not pass the syntax check and are rejected at early state of fuzzing. In grammar-based whitebox fuzzing [6] the authors present a dynamic test generation algorithm that utilizes symbolic execution to generate grammar constraints so that the input can satisfy the syntax check with a grammar-based constraint solver from [7]. In Superion [24], the authors proposed a grammar-aware coverage-based greybox fuzzing that process structured inputs with a grammar-aware trimming technique and grammar-aware mutation strategies as extension to AFL to improve its code coverage. These works help fuzzing process generate well-formed inputs faster. Our design could be based on these inputs generated and explore whether those generated functions inputs could lead to a non-deterministic bug.

Hybrid Fuzzing and Path prioritization. Hybrid fuzzing, combining code coverage guided fuzzing and concolic execution, is becoming an advanced technique to find deep bugs in a faster manner. Driller [23] uses selective concolic execution to explore and solve constraints to different paths and then apply fuzzing to the code compartment following that path to increase code coverages. CollAFL [5] mitigates path collisions and utilizes new fuzzing strategies based on the coverage information to promote the speed of discovering new paths. DigFuzz [27] proposed in probabilistic path prioritization for hybrid fuzzing utilizes a discriminative dispatch strategy, with a Monte Carlo based probabilistic path prioritization model to enhance the ability of concolic execution.

Concurrency error detect. Another group of works is on detecting concurrency errors [11] [14] [2]. Existing methods mostly are based on monitoring memory access and find concurrency errors, or through static analyze of the code fragments. [16] proposes a framework that also combines static analysis and fuzzing, with static analysis to locate and analyze sensitive concurrent parts in a program based on previously categorized features of several potential concurrency errors types, with the results fed into the fuzzers in order to trigger the suspected concurrency vulnerabilities.

6 CONCLUSION

This paper propose a risk-score based mechanism to guide program testing with the goal of triggering GC-related bugs more quickly. In particular, functions in the target program are categorized through their risk scores and associated with different testing rules. We further instrument the target functions to boost garbage generation (and thus the execution frequency of GC process) according to policies determined by our sensitive analyzer. Results show that our approach could trigger different types of GC bugs with a significantly higher probability, while only introducing an execution overhead of 26%.

7 ACKNOWLEDGMENTS

We thank the support of ONR grant N00014-17-1-2786. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of ONR.

REFERENCES

[1] Apple. 2017. WebKitGTK/2.18.x-WebKit. https://trac.webkit.org/wiki/WebKitGTK/2.18.x.

[2] Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing Concurrent Programs on Relaxed Memory Models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 122–132. https://doi.org/10.1145/2001420.2001436

[3] Donald Firesmith. 2017. Seven Recommendations for Testing in a Non-Deterministic World. https://insights.sei.cmu.edu/sei_blog/2017/04/seven-recommendations-for-testing-in-a-non-deterministic-world.html.

[4] Markus Gaasedelen. 2018. Timeless Debugging of Complex Software. https://blog.ret2.io/2018/06/19/pwn2own-2018-root-cause-analysis.

[5] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696.

[6] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[7] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

[8] Google. 2018. AddressSanitizer.

[9] Google. 2018. Concurrent marking in V8. https://v8.dev/blog/concurrent-marking.

[10] IBM. 2017. Memory leak patterns in JavaScript. https://www.ibm.com/developerworks/web/library/wa-memleak/wa-memleak-pdf.pdf.

[11] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[12] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated Concurrency-Bug Fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 221–236. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/jin

[13] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 675–681.

[14] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. *SIGPLAN Not.* 51, 4 (March 2016), 517–530. https://doi.org/10.1145/2954679.2872374

[15] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 529–541.

[16] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 529–541. https://doi.org/10.1145/3274694.3274718

[17] NIST. 2020. National vulnerability database. https://nvd.nist.gov.

[18] NVD. 2019. CVE-2018-4192 Detail. https://nvd.nist.gov/vuln/detail/CVE-2018-4192.

[19] Oracle. 2014. Managing Memory and Garbage Collection. https://docs.oracle.com/cd/E19159-01/819-3681/6n5srlhqf/index.html.

[20] Oracle. 2017. Java Garbage Collection Basics. https://www.oracle.com/technetwork/tutorials/tutorials-1873457.html.

[21] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 245–254. https://doi.org/10.1145/1806799.1806838

[22] Filip Pizlo. 2017. Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector. https://webkit.org/blog/7122/introducing-riptide-webkits-retreating-wavefront-concurrent-garbage-collector/.

[23] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[24] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[25] Michal Zalewski. 2014. American fuzzy lop.

[26] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 179–192. https://doi.org/10.1145/1736020.1736041

[27] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*.