OptRCA: A More Efficient and Accurate Approach for Automated Root Cause Analysis and Explanation

JINGQUAN GE, Continental-NTU Corporate Lab, Singapore
YAOWEN ZHENG, Institute of Information Engineering, Chinese Academy of Sciences, China
YUEKANG LI, University of New South Wales, Australia
WEI MA*, Continental-NTU Corporate Lab, Singapore
SHEIKH MAHBUB HABIB, Continental Automotive Technologies GmbH, Germany
PRAVEEN KAKKOLANGARA, Continental Automotive Singapore Pte Ltd, Singapore
GABRIEL WAYNE BYMAN, Elektrobit Automotive GmbH, Germany
YANG LIU, Nanyang Technological University, Singapore

With the development of automated software testing technology, software developers can get a large number of crash test cases in a short period of time. However, analyzing these crash test cases and finding their root cause is a time-consuming and labor-intensive task. Techniques based on reverse execution and backward taint analysis are proposed to locate the root cause, but can't provide context information or explanation of the underlying fault. To address these two limitations, researchers have proposed an automated root cause analysis technique called AURORA. Although this technique provides powerful root cause analysis capabilities, it also have two obvious shortcomings. First, the results of root cause analysis are not accurate enough. Second, the efficiency of root cause analysis is not high enough. In order to improve these two shortcomings, we propose OptRCA, a more efficient and accurate approach for root cause analysis and explanation. Like AURORA's fuzzing strategy, OptRCA is also designed based on AFL's crash mode. The difference between them is mainly reflected in three points. First of all, the goal pursued by OptRCA is different from that of normal fuzzing technology. OptRCA pursues maximum correlation to ensure that as many crash test cases as possible are related to the same root cause. This test case with maximum correlation can greatly improve the accuracy of root cause analysis. Second, OptRCA proposed a more efficient non-crash test case retention strategy, which we named "Hill Climbing Retention". Using the hill climbing retention method, OptRCA can obtain sufficient root cause information while retaining only a few non-crash test cases. Since the number of test cases is greatly reduced, the efficiency of OptRCA's subsequent root cause analysis process is also greatly improved. In addition, OptRCA also optimizes the analysis formula to obtain more accurate analysis results. In the evaluation experimental results, OptRCA is significantly better than AURORA in terms of accuracy and efficiency. Quantitative analysis shows that OptRCA is 65% more accurate and 61% more efficient than AURORA.

CCS Concepts: • **Security and privacy → Software security engineering**.

*Corresponding author.

Authors' addresses: Jingquan Ge, Continental-NTU Corporate Lab, Singapore, jingquan.ge@ntu.edu.sg; Yaowen Zheng, Institute of Information Engineering, Chinese Academy of Sciences, China, zhengyaowen@iie.ac.cn; Yuekang li, University of New South Wales, Australia, yuekang.li@unsw.edu.au; Wei Ma, Continental-NTU Corporate Lab, Singapore, ma_wei@ntu.edu.sg; Sheikh Mahbub Habib, Continental Automotive Technologies GmbH, Germany, sheikh.mahbub.habib@continental-corporation.com; Praveen Kakkolangara, Continental Automotive Singapore Pte Ltd, Singapore, praveen.kakkolangara@continental-corporation.com; Gabriel Wayne Byman, Elektrobit Automotive GmbH, Germany, Gabriel.Byman@elektrobit.com; Yang Liu, Nanyang Technological University, Singapore, yangliu@ntu.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/5-ART
https://doi.org/10.1145/3736718

Just Accepted

Additional Key Words and Phrases: Root Cause, Fuzzing, Crash, Predicate, Buffer Overflow, Type Confusion

1 INTRODUCTION

In recent years, fuzzing technology has developed rapidly in the field of software testing and has been widely studied in industry and academia [22, 29, 41, 43, 49, 61, 63–65]. The power of fuzzing technology is that it can find as many different paths as possible in the shortest time through high-throughput continuously mutating input, thereby generating crashes quickly and efficiently. These fuzzing techniques, known as coverage-guided fuzzing, can produce a large number of crashes in a short period of time, far exceeding the developer's ability to fix them [35]. The same bug can produce many unique crashes with different execution paths. For example, before the crash site, the program has a conditional statement. The two crash test cases execute different branches of the conditional statement respectively, and finally converge at the same crash site and cause the program to crash. Obviously, the two crash test cases each have a unique execution path, but crash at the same crash site. In other words, the two crash test cases are caused by the same bug or root cause. As a result, developers waste a lot of time investigating potential bugs that don't exist.

To reduce the duplication of work caused by a large number of crashes mapping to the same bug, Researchers propose many bucketing techniques, the most representative of which is stack hashing [26, 28, 39, 52]. The assumption of this bucketing technique is that crashes generated by the same bug have the same stack hash value. However, experimental evaluation results [40] show that this bucketing technique may generate too many buckets, or crashes caused by different bugs will be classified into the same bucket. Even if there are only a few crash test cases for analysts to investigate, finding the root cause of the crash is still a difficult task. Because in many cases, the location of the crash is not consistent with the location of the root cause. The location of the root cause may be earlier in the execution flow than the location of the crash. Therefore, analysts need to spend a lot of time analyzing the execution path to find the root cause.

Among them, the most typical example of root cause is type confusion. In a type confusion scenario, a crash occurs when a program uses a pointer that is assigned the wrong type. However, the location where this crash occurs is not the location of the root cause. The real root cause location should be where the pointer is incorrectly assigned. The easiest way for analysts to find the root cause is to utilize a debugger to inspect the stack and register values. They can start from the crash location and manually trace back along the execution flow to the location of the root cause. If there is the support of sanitizer technology such as ASAN [47], the starting point of this backtrace (illegal memory access) can be closer to the root cause than crash. However, in a type confusion scenario, since most of the code has no wrong behavior, even if the analyst obtains the sanitizer information, it is still difficult to find the root cause location. To reduce the manual workload of backtracking execution flow, researchers have proposed several technologies for automatic reverse execution and reverse taint analysis, such as RETRACER [33], POMP [56], REPT [32], and DEEPVSA [36]. However, these approaches often fail to automatically identify the root cause location when there is no direct data dependency between the root cause and the crashing instruction. Moreover, due to the limited information collected, none of the above approaches can provide an explanation for the fault.

To address these problems, a new root cause automatic analysis and explanation tool called AURORA [24] is proposed. AURORA uses coverage-guided AFL [63] to run *crash exploration*. In this mode, the fuzzer's seeds are crashing inputs, which are constantly mutated so that the target application continues to crash. *Crash exploration* can generate a large number of new crashes related to the original crash, which may be caused by different paths triggering the same bug. Since the original AFL only collects crash test cases, AURORA modifies the original AFL to record both crash and non-crash test cases. AURORA first selects a crashing input as a seed, and then obtains a large number of similar crashing and non-crashing inputs. After that, these crashing and non-crashing inputs are executed and the internal state of the binary program is tracked. The internal state includes the related register

Just Accepted

values and control flow information for each instruction. Based on these detailed traces, AURORA can analyze what kind of input will cause a crash. Although this design gives AURORA powerful analytical capabilities, it also introduces three drawbacks.

- **AURORA's root cause analysis results are not accurate enough.** Since the fuzzing strategy is coverage-guided, the large number of crash test cases generated by AURORA have very different paths to each other. Therefore, the probability that these crash test cases are caused by the same bug is not high. This can lead to inaccurate analysis results from AURORA.
- **AURORA's root cause analysis is too inefficient.** AURORA needs to increase the number of crash and non-crash test cases to improve the accuracy of the analysis results. However, with the coverage-guided fuzzing strategy, the number of crash and non-crash test cases stored by AURORA is extremely unbalanced. The number of non-crash test cases is far more than the number of crashes. In AURORA's analysis process, the impact of all non-crash test cases on the analysis results is similar to that of all crash test cases. The large number of non-crash test cases does not increase the accuracy of the results and is therefore nearly useless. On the other hand, each test case consumes the same amount of time during the analysis process. The large number of non-crash test cases consumes a large amount of analysis time without much gain in analysis results, making the AURORA analysis process very inefficient.
- **AURORA's analysis calculation formula is inaccurate.** The number of crash and non-crash outputs in AURORA's fuzzing process is very unbalanced, which leads to the use of inaccurate predicate calculation formula. This formula reduces the accuracy of root cause analysis.

To improve these three shortcomings, we propose OptRCA, a more efficient and accurate strategy for automated root cause analysis and explanation. Like the AURORA's fuzzing strategy, OptRCA is also designed based on AFL's crash mode. Therefore, OptRCA also keeps only crash test cases when selecting seeds. The difference is that seeds with greater correlation to the original crash receive greater energy values. In other words, seeds with greater correlation to the original crash will be mutated and executed more often. This energy scheduling method will greatly improve the correlation between other crash test cases and the original crash test case, and thus improve the correlation between the crash test case and the true root cause. Obviously, this method increases the amount of useful information in the crash test case, thereby improving the accuracy of root cause analysis. On the other hand, when collecting non-crash test cases, OptRCA adopts a more efficient retention strategy, which we call the "Hill Climbing Retention". The core goal of hill climbing retention is to retain as few non-crash test cases as possible to obtain enough useful information. Since the number of non-crash test cases is greatly reduced, OptRCA can greatly improve the efficiency of root cause analysis. In addition, because the number of crashing and non-crashing inputs is more balanced, OptRCA improves AURORA's analytical calculation formula to make the results more accurate. We select over 20 Common Vulnerabilities and Exposures (CVE) vulnerabilities to evaluate OptRCA's performance and accuracy. Experimental results show that OptRCA is more efficient and accurate than AURORA. On average, OptRCA is 65% more accurate and 61% more efficient than AURORA. Our main contributions are summarized as follows:

- We propose a new fuzzing strategy that is more suitable for root cause analysis and explanation. This fuzzing strategy is designed based on a new seed energy scheduling mechanism we proposed. Based on these crash test cases with maximized correlation, OptRCA can obtain more accurate root cause analysis results.
- We propose a new formula to calculate the distance between non-crash test cases. This formula can effectively represent the amount of new information in non-crash test cases.
- We propose a new non-crash test case retention algorithm called "hill-climbing retention" based on the calculated non-crash distance mentioned above. It can obtain enough information for root cause analysis

Just Accepted

while only retaining a very small amount of non-crash test cases, thus greatly speeding up the efficiency of root cause analysis.
- We propose a more balanced and accurate analysis calculation formula based on the more balanced number of test cases in OptRCA.

To foster research on this topic, we release the implementation of OptRCA at https://github.com/gejingquan/OptRCA.

2 BACKGROUND AND MOTIVATION

This section will first provide background knowledge related to root cause analysis, including sanitizer, backward taint analysis, and fuzzing technology. Finally, we will focus on the technical details of AURORA, the most representative tool in the field of root cause analysis, and its limitations.

2.1 Sanitizer

Sanitizers can detect various types of memory access bugs based on compile time instrumentation and shadow memory. There are currently two most popular sanitizer tools, namely MemorySanitizer (MSAN) [48] and AddressSanitizer (ASAN) [47], both of which can assist in root cause analysis. The types of bugs these two tools focus on are different. MSAN mainly targets uninitialized memory reads. When stack or heap allocated memory is read before it is written, and the read value affects the execution of the program, MSAN can detect this vulnerability immediately. Based on shadow memory technology, MSAN can track each initialized or uninitialized bit. Therefore, MSAN can prevent uninitialized memory from being used illegally. Unlike MSAN, ASAN targets a wider range of memory-related vulnerabilities, including use-after-free (UAF), various types of buffer overflows, use-after-return, use-after-scope, initialization order bugs and memory leaks. ASAN also uses shadow memory to determine whether a specific memory can be accessed. ASAN has two most important modules: an instrumentation module and a runtime library. The instrumentation module creates inaccessible regions around the stack and global objects, and detects overflows and underflows by checking the shadow state of each memory access. The runtime library creates inaccessible regions around allocated heap regions by replacing *malloc()*, *free()*, and related functions, and detects the reuse of freed heap regions.

MSAN and ASAN not only identify invalid memory accesses, but also provide more details about the cause and location of the crash. Therefore, these two tools are of great help in accurately analyzing the root cause, but they have three irreparable shortcomings. First, MSAN and ASAN can only detect memory errors and cannot detect non-memory bugs. Second, when the location of the root cause of the program bug is different from the location of the crash (such as type confusion), these two tools are powerless. Third, the detailed information that MSAN and ASAN can provide is very limited. They only provide memory-related information and cannot provide information such as registers and control flow.

2.2 Backward Taint Analysis

Backward taint analysis is a more accurate root cause analysis technology than MSAN and ASAN. There are three most representative tools based on backward taint analysis technology, namely RETracer [33], POMP [56] and REPT [32]. RETracer is the first tool to analyze the root cause of bugs based on program semantics reconstructed from memory dumps [51]. Based on binary-level backward taint analysis without a recorded execution trace, RETracer can analyze how functions on the stack lead to the crash. However, due to the lack of control flow trace, RETracer can only recover an approximate execution history and limited data values. REPT is an improvement on RETracer. Based on online lightweight hardware tracing of a program's control flow and offline binary analysis, REPT can reconstruct the data flow and the execution history with high fidelity. In order to record the control flow of the program with low performance overhead, REPT utilizes the hardware tracing technology Intel Procesor

Just Accepted

Trace (PT) [30]. Moreover, REPT uses the data values saved in a memory dump with control flow information to recover data flow. POMP is another tool that uses backward taint analysis for program crash analysis. Unlike RETracer and REPT, POMP only requires post-crash artifacts to execute instructions in reverse and reconstruct the data flow. Based on backward taint analysis and the recovered data flow, POMP can pinpoint the critical instructions leading up to the crash.

Using reverse execution and backward taint tracking technology, RETracer, POMP and REPT can understand and analyze bugs more deeply. However, the above three tools all rely on data flow to find the root cause of the crash. When there is no direct data flow between the root cause and the crash site, these three tools are powerless. Type confusion is a typical example. In a type confusion scenario, the location of crash is often in the constructor of the confused object. But the real root cause is often at the wrong object pointer assignment. Unfortunately, there is no direct data flow between these two locations. Therefore, it is difficult for these three tools to find the root cause of type confusion.

2.3 Coverage-Guided Fuzzing and Crash Exploration Mode

Coverage-Guided Fuzzing: Fuzzing is an automatic software testing technology to discover bugs and vulnerabilities. Due to its high efficiency and simplicity, it has become one of the most popular vulnerability mining techniques [35, 50, 63]. Among them, coverage-guided fuzzing [27, 41, 44, 46, 50, 63] is by far the most successful fuzzing technique. Coverage-guided fuzzing tracks code coverage to focus mutations on a unique set of test cases, which can reach previously-unseen code regions. The goal of this fuzzing strategy is to explore all possible execution paths of the binary program as quickly as possible. In other words, its goal is to maximize code coverage of generated test cases. This strategy of maximizing coverage is very effective for quickly finding bugs and vulnerabilities, but it is not suitable for all scenarios. Especially in the context of root cause analysis, maximizing code coverage is harmful. Since root cause analysis requires as many crashes as possible to be caused by the same root cause. Maximizing code coverage will greatly increase the probability of noise crashes (crashes caused by different root causes).

Crash Exploration Mode: *Crash exploration mode* [62] is a fuzzing mode provided by coverage-guided fuzzers such as AFL [63]. As the name suggests, this mode focuses on crash. *Crash exploration mode* uses a crash test case as a mutation seed to generate new test cases. Among new test cases, the fuzzer only keeps the crash test case in the fuzzing queue. Therefore, *crash exploration mode* will generate a large number of crash test cases, which are all mutations of the same crash seed. Intuitively guessing, these crash test cases execute different paths, but there is a high probability that they will cause the same crash. Unfortunately, this intuitive guess is not necessarily true. Through our real program fuzzing testing, we found that in large programs, the crash test cases generated by *crash exploration mode* contain a large number of crashes caused by different bugs. These crashes caused by different bugs have very low correlation with the original crash that needs to be analyzed. In other words, the amount of useful information brought by these crashes is very limited. We classify these crashes with very low correlation with the original crash as noise crashes. Adding these noise crashes has very limited improvement in the accuracy of root cause analysis, and sometimes even harmful. Therefore, the coverage-guided fuzzing strategy is not suitable for root cause analysis.

2.4 AURORA

Section 1 briefly introduces the principles, advantages and disadvantages of AURORA. This section provides a more in-depth explanation of the technical details and limitations of AURORA. The design goal of AURORA is to find the location and explanation of the software fault's root cause, given a crash input and binary program. Based on statistical analysis methods, AURORA can compare the behavioral differences between crash and non-crash inputs, and then achieve the above design goal. AURORA first creates a data set of various program behaviors

Just Accepted

related to this crash. By monitoring the program behavior of these related inputs, AURORA can compare and analyze them. There is no doubt that crash program behavior will deviate semantically from non-crash program behavior at some point. Intuitively, the first deviated program behavior is the root cause. In the design of AURORA, the monitored program behavior can provide both the location and the explanation of the root cause. The first step in AURORA is to create two sets of inputs, crash and non-crash. To obtain these two sets of inputs, AURORA uses the initial crash input as the seed and run the *crash exploration mode* of fuzzing. After getting these two sets of inputs, AURORA monitors and tracks the program behavior of each input, and then correlate these traces with the execution results. Based on statistical reasoning, AURORA can identify the difference between crash and non-crash traces. Specific predicates are synthesized to formalize these differences. Intuitively, the first predicate that can successfully predict all or most of the execution results can also provide an explanation for the root cause. The final result in AURORA is a list of explanations and addresses of possible root causes, ordered by prediction quality and execution time. The three most important technical details of AURORA are actually the answers to the following three questions.

First, how does AURORA monitor program behavior? AURORA's program behavior monitoring for each input is based on Intel PIN [31], which is a dynamic binary instrumentation tool. To gain semantic insights into (binary-only) program behavior, AURORA monitors the runtime execution of each crash and non-crash input and collects the values of various expressions. AURORA records the maximum and minimum values of registers modified by each instruction, including general registers and flag registers. For each memory write access, AURORA records the maximum and minimum values stored. In addition to registers and memory, AURORA also stores control-flow edge information to reconstruct coarse control flow graphs. Moreover, stack and heap address ranges are also collected by AURORA to test the validity of stack or heap pointers.

Second, what predicates does AURORA use to make predictions? AURORA provides a total of three categories of predicates, namely register- and memory-related predicates, control flow-related predicates, and flag-related predicates. For register- and memory-related predicates, AURORA generates expressions of $r < c$ or $r > c$ based on the maximum and minimum of all values written to registers and memory. In the expression, r is the maximum or minimum value of the register or memory, and c is a selected appropriate constant. In addition, AURORA also designs two expressions $is_heap_ptr(r)$ and $is_stack_ptr(r)$ to test whether r is a valid heap or stack pointer. For control flow-related predicates, AURORA designs two expressions, namely has_edge_to and $always_taken_to$. Given a control flow edge from x to y, has_edge_to refers to at least one transition from x to y. And $always_taken_to$ means that every edge output from x goes to y. For each instruction, the Boolean values of the two expressions are evaluated and added together. Checking whether the sum of this Boolean value is greater than or equal to $n \in \{0, 1, 2\}$ is the final result of this predicate. For the flag predicate, AURORA uses the Boolean value of each flag bit (including the carry, zero and overflow flags) to predict the difference between crash and non-crash.

Third, how does AURORA calculate prediction quality? AURORA uses actual crash and non-crash input samples to maximum likelihood estimate the misprediction probability of the predicate. The original unmodified formula of this maximum likelihood estimate is shown in Equation 1. In Equation 1, $\hat{\theta}$ represents the maximum likelihood estimate of the misprediction probability. C_f and N_f refer to the number of mispredicted crashes and non-crashes respectively. Correspondingly, C_t and N_t refer to the number of correctly predicted crashes and non-crashes respectively.

$$\hat{\theta} = \frac{C_f + N_f}{C_f + N_f + N_t + C_t} \tag{1}$$

However, due to the large gap between the number of crash and non-crash samples in AURORA, using Equation 1 to calculate the estimated value may produce a large error. Therefore, AURORA modifies this original formula

Just Accepted

to balance the impact of crash and non-crash on the maximum likelihood estimate, as shown in Equation 2. The value range of $\hat{\theta}$ is $[0, 1]$. $\hat{\theta} = 0$ means that the prediction success probability of this predict is 100%. In contrast, $\hat{\theta} = 1$ indicates that the success probability of the reverse prediction is 100%. Therefore, the closer $\hat{\theta}$ is to 0 or 1, the greater the quality of this predicate. When $\hat{\theta}$ is close to 0.5, it indicates that the quality of this predicate is small.

$$\hat{\theta} = \frac{1}{2} * (\frac{C_f}{C_f + C_t} + \frac{N_f}{N_f + N_t}) \tag{2}$$

Since the value of $\hat{\theta}$ is not positively related to the prediction quality, it is not appropriate to use $\hat{\theta}$ to characterize the prediction quality. In order to better characterize the prediction quality of the predicate, AURORA provides another variable S. The calculation formula of S is as shown in Equation 3. The same as $\hat{\theta}$, the value range of S is also $[0, 1]$. The difference is that the value of S is positively related to the prediction quality. When $S = 0$, the prediction quality is the worst. Correspondingly, $S = 1$ indicates the best prediction quality. Therefore, S can well characterize the prediction quality. AURORA uses S to select the best predicate for each instruction.

$$S = 2 * |\hat{\theta} - 0.5| \tag{3}$$

Although AURORA is a very powerful automatic root cause analysis and explanation tool, its accuracy and efficiency are far from optimal. Section 1 provides a detailed explanation of the three flaws of AURORA. This is exactly our motivation for designing OptRCA, which is to comprehensively optimize the efficiency and accuracy of AURORA.

2.5 State-of-the-Art Statistical RCA Solutions

The latest root cause analysis (RCA) techniques can be divided into two major categories, statistical RCA and non-statistical RCA. Non-statistical RCA refers to identifying and locating the root cause of certain types of vulnerabilities based on detailed and precisely formulated rules. In the non-statistical RCA category, the latest technical solutions are ARCUS[58] and Bunkerbuster [57]. Statistical RCA refers to estimating and ranking the statistical correlation between each program entity (such as statement, block or predicate) and crash based on a set of crash and non-crash test cases. Obviously, AURORA and OptRCA belong to the category of statistical RCA. In addition to AURORA and OptRCA, there are two latest technical solutions in the statistical RCA category, namely RACING [54] and BENZENE [42]. RACING uses reinforcement learning technology to reward operations involving counterexamples, thereby balancing random sampling and counterexample exploitation. RACING greatly improves the scalability and accuracy of root cause analysis, and its efficiency is an order of magnitude higher than AURORA. BENZENE is a practical end-to-end and more efficient automated root cause analysis system. BENZENE uses a new technology called under-constrained state mutation to obtain non-crash test cases that are closer to the original crash. Compared with AURORA, BENZENE is not only more efficient and accurate, but also occupies less memory footprint.

3 DESIGN

This section introduces the design details of OptRCA. We first provide an overview of OptRCA in Section 3.1. Then, Section 3.2 shows the technical details of OptRCA's fuzzing strategy, test case retention strategy, and the calculation method of root cause analysis.

3.1 Overview of OptRCA

OptRCA is an automated root cause analysis and explanation tool designed to optimize AURORA in terms of both accuracy and efficiency. Figure 1 shows the entire workflow diagram of OptRCA. Among them, the

Just Accepted

gray modules represent OptRCA's improvements to the original AURORA tool. As can be seen from Figure 1, OptRCA consists of two major stages, namely the fuzzing stage and the analysis stage, which are the same as the original AURORA. However, OptRCA optimizes the internal modules of these two stages. OptRCA optimizes three modules respectively, namely Seed Energy Scheduling for Maximizing Correlation, Hill-Climbing Retention and Calculation module. The first two modules are improvements in the fuzzing stage, and Calculation module belongs to the analysis stage.

In the fuzzing stage, like AURORA, OptRCA is also based on *crash exploration mode* of the fuzzer to find crash and non-crash sets. Different from AURORA, OptRCA optimizes the fuzzing strategy in two parts. First, OptRCA provides a seed scheduling strategy that maximizes correlation to make the found crash test cases more suitable for root cause analysis. This is completely different from AURORA's fuzzing strategy of maximizing coverage. Secondly, OptRCA improves the retention method of non-crash test cases. The original AURORA does not have any special retention policy, all non-crash is retained. OptRCA uses the hill-climbing retention method to filter non-crash test cases. In the analysis stage, OptRCA uses a more optimized root cause statistical analysis calculation formula to make its results more accurate.

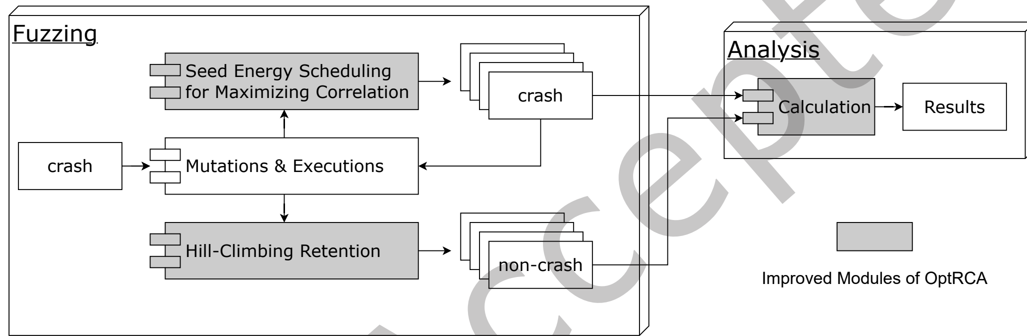


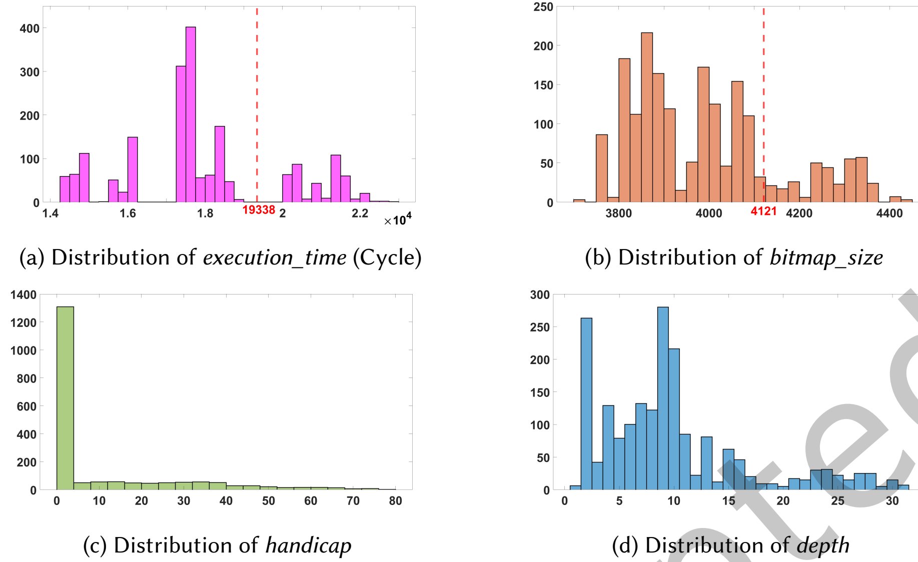
Fig. 1. The entire workflow of OptRCA. The gray modules represent OptRCA's improvements to the AURORA tool.

3.2 Technical Details

In this section, we detail the principles and functions of the above three improved modules. We first describe how OptRCA's fuzzing strategy maximizes correlation by modifying seed energy scheduling. Then, we give the principle of OptRCA's non-crash retention strategy. Finally, we introduce in detail how OptRCA optimizes the root cause analysis process.

3.2.1 Seed Energy Scheduling for Maximizing Correlation. Normal fuzzing strategies aim to maximize coverage. This is because maximizing coverage can find new crashes and vulnerabilities more quickly, thus greatly improving the efficiency of fuzzing. However, from the perspective of root cause analysis, maximizing coverage has the opposite effect. Because root cause analysis requires a large number of crash test case collections that belong to the same root cause. The crash test cases obtained by maximizing coverage will have many noisy crashes (not belonging to the same root cause). Theoretically, the greater the correlation of the collected crash test cases, the greater the probability that they belong to the same root cause, and the more suitable they are for the root cause analysis scenario. In order to make the collected crash test cases more suitable for root cause analysis, we propose a fuzzing strategy that maximizes correlation. In order to achieve maximum correlation of crash test cases, we propose a novel seed energy scheduling algorithm, as shown in Algorithm 1. The seed energy determines how

Just Accepted



(a) Distribution of *execution_time* (Cycle)

(b) Distribution of *bitmap_size*

(c) Distribution of *handicap*

(d) Distribution of *depth*

Fig. 2. Distribution of four variables related to seed energy obtained by the original AURORA fuzzing strategy (CVE-2018-10191 [1]). The first variable *bitmap_size* represents the scale of code coverage. The second variable *execution_time* represents the execution time of the fuzz target binary. The third variable *handicap* is a tuning variable that allows latecomers to run for a longer time. The fourth variable *depth* represents the number of times the current seed has been mutated since the original seed. In Figure 2a and Figure 2b, the red dashed vertical line refers to the average value.

often the seed will be fuzzed. The higher the energy, the more often it will be fuzzed, and vice versa. Therefore, seeds with higher energy will be mutated and executed more often. The energy of the seed is related to four variables, namely *bitmap_size*, *execution_time*, *handicap* and *depth*. Figure 2 shows the distribution of these four variables during the fuzz testing phase of AURORA. The detailed explanation of the variables is also given in the caption of Figure 2.

Table 1. The explanations of all the constants and variables used in Equation 4, 5, 6 and 7.

Constant/Variable	***Explanation***
OriScore	The initial constant for the seed energy value
mutated	Seed that has been mutated.
ETscore	Energy score coefficient related to execution time.
average_execution_time	Average execution time of all seeds in the seed queue.
minimum_execution_time	Minimum execution time of all seeds in the seed queue.
BSscore	Energy score coefficient related to bitmap size.
average_bitmap_size	Average bitmap size of all seeds in the seed queue.
minimum_bitmap_size	Minimum bitmap size of all seeds in the seed queue.
handicap	A tuning variable that allows latecomers to run for a longer time.
HCscore	Coefficient of energy score related to *handicap*.
depth	The number of times the current seed has been mutated since the original seed.
Dscore	Coefficient of energy score related to *depth*.

Just Accepted

The variable *execution_time* refers to the execution time of the binary. If it is too large, it will affect the efficiency of fuzzing. Ideally, high-priority seeds should have lower *execution_time*. On the other hand, the length of execution time does not theoretically have a significant impact on the results of root cause analysis. Therefore, in the design of OptRCA, our goal is to further reduce the *execution_time* of high-priority seeds to improve the efficiency of fuzzing. As can be seen from Figure 2a, *execution_time* is randomly and dispersedly distributed on both sides of the mean. This proves from the side that AURORA's fuzzing strategy is not sensitive to *execution_time*. If OptRCA's fuzzing strategy expects to select seeds with shorter execution time, the distribution of *execution_time* should be more compact and the mean value will be lower. After a lot of experimental data statistics and testing, we designed Equation 4 as the calculation formula for the seed energy coefficient of *execution_time*. The explanation of the constants and variables in Equation 4 can be found in Table 1. Equation 4 involves three parameters, namely *ETscore*, *average_execution_time* and *minimum_execution_time*. From Table 1, we can see that *ETsore* is the energy score coefficient related to the execution time. It can be seen from Equation 4 that the larger the value of *execution_time*, the smaller the *ETscore* is, and the decline is rapid. If the *execution_time* of the current mutated seed is close to *minimum_execution_time*, the *ETscore* will be very large. If the current mutated seed's *execution_time* is close to *average_execution_time*, the *ETscore* is approximately equal to 1. If the current mutated seed's *execution_time* is greater than *average_execution_time*, the ETscore will be less than 1. This formula makes the seed energy coefficient *ETscore* of OptRCA very sensitive to *execution_time* of the mutated seed. This feature of *ETscore* can make the *execution_time* of the seed sequence gather towards the minimum value. The evaluation results in Figure 3 of Section 5.2 also prove this characteristic.

$$mutated.ETscore = \frac{average_execution_time - 0.9 * minimum_execution_time}{mutated.execution_time - 0.9 * minimum_execution_time} \tag{4}$$

The parameter *BSscore* refers to the energy score coefficient associated with the *bitmap_size*. The original AURORA fuzzing strategy expects to obtain the maximum coverage, so seeds with larger *bitmap_size* will obtain larger energy coefficients. Figure 2b shows the *bitmap_size* distribution of the seed sequence obtained by the original AURORA fuzzing strategy. As can be seen from the Figure 2b, the distribution of *bitmap_size* is the same as *execution_time*, which is randomly and loosely distributed on both sides of the mean. This distribution is inconsistent with OptRCA's expectations. In order to maximize the correlation between seeds, OptRCA expects a distribution with a lower mean and a more compact distribution. Equation 5 is the calculation formula for OptRCA to calculate *BSscore*, which is very similar to the calculation method of *ETscore*. When the *bitmap_size* of the current mutated seed is close to the minimum value, the *BSscore* will be very large. On the contrary, when the *bitmap_size* of the current mutated seed is larger than the average value, the *BSscore* will be less than 1. This characteristic of *BSscore* will cause the *bitmap_size* of the seeds in the seed sequence to gather around the minimum value. The evaluation results in Figure 3 of Section 5.2 also prove this characteristic.

$$mutated.BSscore = \frac{average_bitmap_size - 0.9 * minimum_bitmap_size}{mutated.bitmap_size - 0.9 * minimum_bitmap_size} \tag{5}$$

The parameter *handicap* is used to adjust the running time of latecomers in the seed sequence. The effect of this parameter is that late seeds will get more seed energy until the late time is compensated. When a seed in each seed sequence is mutated for the first time, its *handicap* will be assigned an initial value, that is, the number of queue rounds minus one. This means that the later the seed is, the larger the initial value of its *handicap*. The parameter *handicap* decreases as the number of times the seed is fuzzed increases. When the *handicap* of the seed is greater than or equal to 4, the *handicap* decreases by 4 each time the seed is fuzzed. When the *handicap* of the seed is less than 4, the *handicap* decreases by 1 each time the seed is fuzzed. As shown in Table 1, the parameter *HCscore* represents the seed energy coefficient related to handicap. In the original AURORA fuzzing strategy, when $handicap >= 4$, the *HCscore* is 4, and when $0 < handicap < 4$, the *HCscore* is 2. This

Just Accepted

HCscore feature will greatly increase the frequency of fuzzing seeds with larger handicap. The distribution of *handicap* shown in Figure 2c also illustrates this impact of *HCscore*. According to our statistical and theoretical analysis, *handicap* value of the current seed has no effect on its correlation with the original seed. Therefore, in the design of OptRCA, we expect the influence of *handicap* on *HCscore* to be reduced. This can make the value of larger *handicap* drop to the low value range faster, thereby reducing the advantage of late seeds. Equation 6 is the formula for calculating *HCscore* of OptRCA. The difference between Equation 6 and the original formula of AURORA is that the value of *HCscore* is halved in both cases. The *HCscore* calculated based on Equation 6 can reduce the distribution probability of larger *handicap* and increase the distribution probability of smaller *handicap*. This characteristic can also be proved by evaluation results in Figure 3 of Section 5.2.

$$mutated.HCscore = \begin{cases} 2 & (\text{if } mutated.handicap >= 4), \\ 1 & (\text{if } 0 < mutated.handicap < 4). \end{cases} \tag{6}$$

The parameter *depth* refers to the number of times the current mutated seed has been mutated from the original seed. Obviously, this parameter has a great impact on the correlation between the current mutated seed and the original seed. Figure 2d shows the distribution of the parameter *depth* of the original AURORA fuzzing strategy. In the original design of AURORA, the energy score coefficient *Dscore* related to *depth* is positively correlated with *depth*. This is mainly because normal fuzzing expects to find new and different types of crashes, and a larger *depth* means a greater probability of finding new and different types of crashes. However, OptRCA's fuzzing strategy is opposite to AURORA's requirements. OptRCA's fuzzing strategy expects to obtain the mutated seed with the greatest correlation with the original seed. On the other hand, from a statistical point of view, the larger the *depth* value of the mutated seed, the farther it is from the original seed, and the lower the correlation. Therefore, the distribution of OptRCA's *depth* should be as compact as possible and close to the minimum value. In order to obtain this distribution characteristic, we designed Equation 7 as the calculation formula for OptRCA's coefficient of energy score related to *depth*. Equation 7 makes the relationship between *Dscore* and *depth* linearly negatively correlated, so that the distribution of *depth* is closer to the minimum value and more compact. Figure 3 in Section 5.6 also fully demonstrates this characteristic of *Dscore*.

$$mutated.Dscore = \frac{1}{mutated.depth} \tag{7}$$

$$\begin{aligned} mutated.score = OriScore * mutated.BSscore * mutated.ETscore* \\ mutated.HCscore * mutated.Dscore \end{aligned} \tag{8}$$

Equation 8 is the final formula for calculating the energy value of the mutated seed. In Equation 8, *OriScore* is the initial constant of the seed energy value. From Equation 8, we can see that the energy of the seed is the product of the initial value and the four seed energy score coefficients. Algorithm 1 is the seed energy scheduling algorithm designed according to Equation 8, which we call "Seed Energy Scheduling for Maximizing Correlation". From line 6 of Algorithm 1, we can see that when the seed is mutated, it is first determined whether the mutated test case causes a crash. If the mutated test case causes a crash, it continues to determine whether it covers a new path, as shown in line 7 of Algorithm 1. When the mutated test case causes both a crash and a new path, the test case is stored in the seed queue as a new seed, as shown in line 8 of Algorithm 1. After the test case is stored in the seed queue, Algorithm 1 will calculate the energy value according to Equation 4, 5, 6, 7 and 8, as shown in lines 9 to 24 of Algorithm 1. When the test case does not cause a crash, OptRCA will determine whether to store the non-crash test case according to Algorithm 2, as shown in line 26 of Algorithm 1.

Just Accepted

Algorithm 1: Seed Energy Scheduling for Maximizing Correlation

```
   Input: Crash test case to be analyzed: Crash_Case;
   Output: Energy score of mutated seed: mutated.score;
1  Add Crash_Case to the seed queue.
2  while true do
3      for Seed in Queue do
4          mutated ← Mutate(Seed);
5          Execute Program with mutated as input;
6          if mutated is crashing then
7              if mutated has new paths then
8                  Add mutated to the seed queue;
9                  Update mutated.bitmap_size;
10                 Update average_bitmap_size;
11                 mutated.BSscore = (average_bitmap_size−0.9∗minimum_bitmap_size) / (mutated.bitmap_size−0.9∗minimum_bitmap_size);
12                 Update mutated.execution_time;
13                 Update average_execution_time;
14                 mutated.ETscore = (average_execution_time−0.9∗minimum_execution_time) / (mutated.execution_time−0.9∗minimum_execution_time);
15                 Update mutated.handicap;
16                 if mutated.handicap >= 4 then
17                     mutated.HCscore = 2;
18                 else
19                     mutated.HCscore = 1;
20                 Update mutated.depth;
21                 mutated.Dscore = 1 / mutated.depth;
22                 mutated.score =
                     OriScore ∗ mutated.BSscore ∗ mutated.ETscore ∗ mutated.HCscore ∗ mutated.Dscore;
23             else
24                 continue;
25         else
26             Run Algorithm 2;
```

3.2.2 Hill-Climbing Retention: During the root cause analysis process, each crash and non-crash test case requires a certain amount of time to be tracked, processed and calculated. Too many test cases will greatly reduce the processing speed of root cause analysis. On the other hand, not every test case has the same amount of new information. Low-information test cases have limited impact on the accuracy of the results and can be approximated as noise. Therefore, how to reduce the number of test cases as much as possible while ensuring sufficient information is of great help to improve the efficiency of root cause analysis.

Intuitively, the amount of information in each crash test case is greater than the amount of information in each non-crash test case. From the perspective of information theory, the conclusion remains the same. This is not only because the number of crash test cases is small, but also because each crash test case is most likely caused

Just Accepted

Algorithm 2: Hill-Climbing Retention

```
  Input: Crash test case to be analyzed: Crash_Case;
  Output: Non-crashing mutated seed: mutated_NC;
1 Add Crash_Case to the seed queue.
2 while true do
3     for Seed in Queue do
4         i = 1 ;
5         j = 1 ;
6         mutated ← Mutate(Seed);
7         Execute Program with mutated as input;
8         if mutated is crashing then
9             Run Algorithm 1;
10        else
11            if mutated has new paths then
12                i += 1 ;
13                if mutated has new tuples then
14                    j += 1 ;
15                Update new_tuple_counter_i ;
16                Update new_hit_counter_i ;
17                new_information_i = (j * new_tuple_counter_i)^2 + new_hit_counter_i^2;
18                if new_information_i is larger than any of the previous ones then
19                    mutated_NC = mutated;
20                    Retain mutated_NC with high probability or 100%;
21                else
22                    mutated_NC = mutated;
23                    Retain mutated_NC with low probability or zero;
24            else
25                continue;
```

by the same root cause. In other words, the execution path of each crash will most likely go to the same crash site, so their execution paths will overlap. It can be inferred that there is a high probability that crash test cases are strongly correlated. Therefore, each crash test case has a lot of useful information. Correspondingly, since the execution path of the non-crash test case will not reach the crash site, there is little overlap with the execution path of the crash to be analyzed. It can be inferred that the non-crash test cases are weakly related to the crash test cases to be analyzed. Therefore, the amount of useful information possessed by non-crash test cases is likely to be much smaller than that of crash test cases. Therefore, OptRCA's retention strategy has two key points. First, keep all crash test cases. Secondly, on the premise of ensuring enough useful information, reduce the number of non-crash test cases as much as possible.

In the design of OptRCA, we propose the hill-climbing retention method to reduce the number of unnecessary non-crash test cases. We first design a calculation formula for the amount of new information for non-crash test

Just Accepted

cases. Suppose $new_tuple_counter_i$ represents the number of newly added path tuples in the execution path of the i-th test case, $new_hit_counter_i$ represents the number of newly changed hit-counts for a particular path tuple in the execution path of the i-th test case, j is the case counter of the tuple being changed, and $new_information_i$ represents the amount of new information brought by the i-th test case. Here, tuple refers to the smallest element in the code path, and hit-count refers to the number of executions of a certain tuple. According to our theoretical analysis of the amount of new information, the greater the $new_tuple_counter_i$ or $new_hit_counter_i$ of the test case, the greater the amount of new information brought. Moreover, the influence of $new_tuple_counter_i$ of new test cases on the amount of new information will increase with the increase of the number of test cases, but the influence of $new_hit_counter_i$ on the amount of new information will not change with the increase of the number of test cases. Therefore, the formula for the amount of new information we designed is as follows:

$$new_information_i = (j * new_tuple_counter_i)^2 + new_hit_counter_i^2 \tag{9}$$

As can be seen from Equation 9, the variable $new_information_i$ is equal to the square of $j * new_tuple_counter_i$ plus the square of $new_hit_counter_i$. The counter j is added to balance the influence of $new_tuple_counter_i$ and $new_hit_counter_i$ on $new_information_i$. Based on the definition of the newly added information ($new_information_i$), we design the retention strategy of test cases. When the amount of new information of the current test case is larger than all previous test cases ($new_information_{i-1}, \cdots, new_information_1$), OptRCA will retain the test case with a high probability or 1. If the amount of new information $new_information_i$ is not the largest, the probability of being retained will be very small or zero. Based on this strategy, OptRCA always retains the test cases with the largest amount of new information with a high probability, and discards the test cases with insufficient information. During the entire process of OptRCA's fuzzing, the number of retained non-crash test cases increases one by one. Within a certain period of time, the number of non-crash test cases can reach a peak and no longer increase. From the perspective of information volume, the amount of new information of the retained test cases increases step by step like climbing a mountain, so we named this retention strategy "hill-climbing retention".

Algorithm 2 shows the hill-climbing retention method for non-crash test cases. Here, i and j respectively represent the non-crash sequence number and the non-crash sequence number with new tuples. The non-crash retention algorithm also needs to complete 4 tasks. First, i and j are updated by adding 1 separately conditionally. When *mutated* does not cause the program to crash but there are new execution paths, the sequence number i is updated by incrementing 1, as shown in line 12. Under the above conditions, when new tuples appear in *mutated* bitmap, the sequence number j is updated by adding 1, as shown in line 14. Second, update the number of *mutated*'s new tuples ($new_tuple_counter_i$) and the number of *mutated*'s new hit-counts ($new_hit_counter_i$), as shown in lines 15 and 16. Third, use Equation 9 to calculate *mutated*'s newly added amount of information, in line 17 of Algorithm 2. Fourth, based on the judgment result of whether $new_information_i$ of the current test case is the largest element in the seed mutation set, determine whether to retain this test case. Lines 18 to 23 illustrate this retention method. If the judgment result is true, then *mutated* is retained with high probability or 100%. If the judgment result is false, then retain *mutated* with small probability or 0.

3.2.3 Calculation: OptRCA's third optimization for AURORA is to modify the calculation method of root cause analysis. From Section 2.4, we can know that AURORA uses Equation 2 and 3 to calculate the prediction quality of each predicate. Equation 2 calculates the crash prediction quality (based on C_f and C_t) and the non-crash prediction quality (based on N_f and N_t) separately, and then averages them. This calculation method results in crash and non-crash having an equal impact on the final result, which obviously does not comply with the principles of statistics. In fact, compared to non-crash test cases, the number of crash test cases is much smaller. Statistically speaking, non-crash will definitely have a much greater impact on the final result than crash. Therefore, by combining crash and non-crash calculations, the results obtained will be more realistic. In

Just Accepted

other words, the calculation result of Equation 1 is more realistic than Equation 2. The reason why AURORA chooses Equation 2 is because the huge imbalance between the number of crashes and the number of non-crash will greatly increase the error of Equation 1. Fortunately, OptRCA's non-crash retention algorithm overcomes this quantity imbalance defect of AURORA. Since the hill-climbing retention method filters out most useless non-crash test cases, the number of crashes and non-crash in OptRCA is already on the same order of magnitude. Therefore, the error calculated by OptRCA using Equation 1 is within the acceptable range. OptRCA combines Equation 1 with Equation 3, and the final calculation formula is shown in Equation 10.

$$S = 2 * |\frac{C_f + N_f}{C_f + N_f + N_t + C_t} - 0.5| \quad (10)$$

As can be seen from Equation 10, OptRCA does not distinguish crash and non-crash predicates separately, but treats them as the same sample. There is no doubt that the results obtained by Equation 10 are more realistic and certainly more accurate than AURORA. The definitions of C_f, N_f, C_t and N_t here are the same as in Section 2.4. C_t and C_f are the number of correct and incorrect predictions in crash samples respectively, and N_t and N_f are the number of incorrect predictions in non-crash samples. From Equation 10, we can know that OptRCA first calculates the proportion of samples with incorrect predictions to the total number of samples ($\frac{C_f+N_f}{C_f+N_f+N_t+C_t}$). OptRCA then calculates how far this proportion is from 0.5 and multiplies it by 2. The result obtained represents the prediction quality of the predicate.

4 IMPLEMENTATION

To prove the practical feasibility of OptRCA, we implemented a prototype of OptRCA. In this section, we introduce the implementation details of OptRCA.

Fuzzing and Non-Crashing Retention. In order to obtain as many crash test cases as possible that belong to the same root cause as the original seed, we use AFL's *crash exploration mode* [62] to fuzz the program. However, the original AFL crash mode can only store crashing test cases, and non-crashing test cases will be discarded. Therefore, we modified *save_if_interesting()* function to save non-crashing test cases with new execution paths. We set the global variables $new_tuple_counter$, $new_hit_counter$ and the global array $new_information$, whose definition is the same as Equation 9. We modify *has_new_bits()* function in the AFL source code to update $new_tuple_counter$ and $new_hit_counter$ of each test case. As mentioned above, the modified *save_if_interesting()* function is responsible for the retention of non-crashing. Therefore, the implementation of the hill-climbing retention method also requires modification of *save_if_interesting()* function. We modify *save_if_interesting()* function to achieve two tasks, one is to update $new_information$ of the current test case, and the other is to determine whether to retain the current non-crash test case based on $new_information$. Moreover, in the AFL code, the function that assigns a value to the seed energy is *calculate_score()*. We modified function *calculate_score()* based on the statements in lines 11 to 22 in Algorithm 1 to achieve the goal of maximizing correlation.

Monitoring Program Behavior. We utilize Intel PIN [31] to monitor program behavior for each test case. Relying on the rich API provided by Intel Pin, we can get context information such as general registers, memory reading and writing, and flag register contents. The implementation of this part is the same as AURORA [25].

Explanation Synthesis. The implementation of this part is modified based on AURORA's Rust project [25]. We utilize the Intel *ptrace* syscall to set conditional breakpoints to monitor the execution flow of each predicate. Then, in order to get the source file, function name and source code line of the root cause, we used the *binutils*'s *addr2line* tool [34]. Most importantly, we modified *PredicateAnalyzer::evaluate_predicate()* function in AURORA's Rust project to implement a more accurate calculation method for Equation 10.

Just Accepted

5 EXPERIMENTAL EVALUATION

In this section, we detail the seed and test case distribution, crash exploration efficiency, analysis accuracy, analysis efficiency, comparison with state-of-the-art solutions, other statistical analysis and ablation study of OptRCA. First, our evaluation setup is introduced in Section 5.1. Then we describe seed and test case distribution of OptRCA in Section 5.2. In Section 5.3, we show the efficiency of OptRCA in exploring crashes. Section 5.4 demonstrates the accuracy of OptRCA's root cause analysis. The root cause analysis efficiency of OptRCA is provided in Section 5.5. Section 5.6 shows the comparison between OptRCA and state-of-the-art solutions. Section 5.7 describes other statistical analysis of OptRCA. Finally, the ablation study of OptRCA is presented in Section 5.8.

5.1 Evaluation Setup

Table 2. The information of target program bugs.

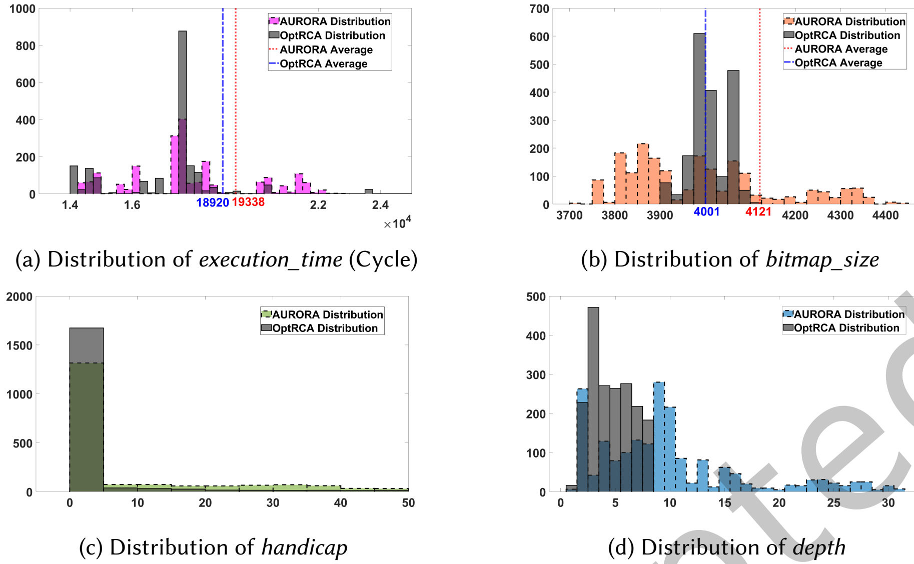
Program Bug	Version	Fuzzing Subject	Program Bug	Version	Fuzzing Subject
#1 mruby [1]	mruby-1.4.0	mruby @@	#11 php [5]	php-7.2.26	php @@
#2 mruby [12]	mruby-3.0.0	mruby @@	#12 nasm [19]	nasm-2.16.02rc1	nasm @@
#3 mruby [4]	mruby-2.1.1	mruby @@	#13 nasm [15]	nasm-2.16rc0	nasm -M @@
#4 mruby [9]	mruby-2.1.1	mruby @@	#14 nasm [3]	nasm-2.15.04rc3	nasm @@
#5 mruby [8]	mruby-2.1.0	mruby @@	#15 nasm [2]	nasm-2.15rc10	nasm @@
#6 lua [16]	lua-5.4.5	lua @@	#16 objdump [18]	objdump-2.40.50	objdump -x @@
#7 lua [20]	lua-5.4.5	lua @@	#17 objdump [17]	objdump-2.40.50	objdump -S @@
#8 lua [13]	lua-5.4.3	lua @@	#18 nm-new [14]	nm-new-2.39.50	nm-new -aD @@
#9 php [7]	php-8.0.0	php @@	#19 readelf [10]	readelf-2.35.50	readelf –dyn-syms @@
#10 php [6]	php-7.4.0	php @@	#20 nm-new [11]	nm-new-2.35.50	nm-new –synthetic @@

All our experiments are conducted on a cloud server with 80 cores (based on Intel Xeon Gold 6248, 2.50GHz) and 188 GiB RAM. The operating system of the cloud server is Ubuntu 22.04. We disabled Address Space Layout Randomization (ASLR) to facilitate deterministic analysis. We evaluate various aspects of OptRCA using 19 CVEs in 7 real-world programs and 1 recent segfault bug. The set of target program bugs in OptRCA is different from that in AURORA. This is mainly because the CVEs and target program bugs selected by AURORA are all very old software versions, many of which have stopped being updated and maintained. OptRCA selects newer CVEs and target program bugs in newer software versions, which can make the experimental results more valuable and reasonable. All program bugs and fuzzing subjects are listed in Table 2. We compared OptRCA with AURORA [24] in various aspects, including the distribution of non-crash test cases, crash exploration capabilities, the accuracy and efficiency of root cause analysis. All fuzzing evaluations are performed without a dictionary.

5.2 Distribution of Seeds and Test Cases

This section mainly contains two aspects. First, the experimental data comparison distribution diagram is used to describe the characteristics of OptRCA's maximum correlation fuzzing strategy. Second, the experimental data scatter plot is used to describe the hill climbing retention method of OptRCA's non-crash test cases. Figure 3 shows the impact of OptRCA's fuzzing strategy based on Algorithm 1 on the distribution of four parameters of seeds. The fuzzing phase of the experiment in Figure 3 ran for a total of 15 minutes, and the four parameters of each fuzzed seed were collected. The impact of OptRCA's retention strategy based on Algorithm 2 on the distribution of non-crash test cases is shown in Figure 4. The experiment in Figure 4 ran for a total of 30 minutes, and the two parameters $j * new_tuple_counter$ and $new_hit_counter$ of all non-crash test cases were collected.

Just Accepted



(a) Distribution of *execution_time* (Cycle)

(b) Distribution of *bitmap_size*

(c) Distribution of *handicap*

(d) Distribution of *depth*

Fig. 3. Comparison of the distribution of four parameters between AURORA and OptRCA (CVE-2018-10191 [1]). The definitions of the four parameters are the same as those in the caption of Figure 2. The experimental data collection time is 15 minutes. The parameter distribution data in the figure is the average of the collected data of 10 experiments.

Just Accepted

The four small pictures in Figure 3 show the distribution of the four parameters of the seeds. The colored bar graphs in the four pictures represent the distribution of the four parameters of the original AURORA, and the gray bar graphs represent the distribution of the four parameters of OptRCA. From Figure 3a, we can see that compared with the original AURORA, the average value of OptRCA's *execution_time* is smaller and the distribution is more compact. The smaller average value of the parameter *execution_time* makes OptRCA's fuzzing efficiency higher than AURORA. The more compact distribution of the parameter *execution_time* can reasonably be inferred that the execution paths of the seeds are more similar. And more similar execution paths mean greater correlation. As can be seen from Figure 3b, the distribution of the parameter *bitmap_size* of OptRCA is also more compact, and the average value is smaller. This is enough to show that the number of new paths for the same number of seeds is reduced. The reduction in new paths can reasonably be inferred that the correlation between seeds has increased. Figure 3c shows the distribution of the parameter *handicap*. From Figure 3c, we can see that the distribution of OptRCA's *handicap* is more concentrated in the interval less than 5, and the distribution of OptRCA's *handicap* greater than 5 is significantly reduced. This shows that OptRCA reduces the encouraging effect of parameter *handicap* on late seeds in the seed queue. Because the correlation between late seeds and original seeds is not greater than that of early seeds, it is reasonable to reduce the impact of the parameter *handicap*. Figure 3d shows the distribution of the parameter *depth*. As can be seen from Figure 3d, the distribution of the parameter *depth* of OptRCA is more concentrated in the interval less than 10. Because the parameter *depth* represents the number of mutations between the current seed and the original seed, a larger *depth* means a smaller correlation. Therefore, the distribution of OptRCA can undoubtedly increase the correlation between seeds more than AURORA.

As described in the technical details in Section 3.2, OptRCA utilizes the hill-climbing retention method to greatly reduce the number of retained non-crash test cases while retaining the necessary amount of new information. In order to more intuitively demonstrate OptRCA's optimization in retaining non-crash test cases, we used AURORA and OptRCA to fuzz and analyze mruby's heap-use-after-free bug (CVE-2018-10191 [1]), and sorted out

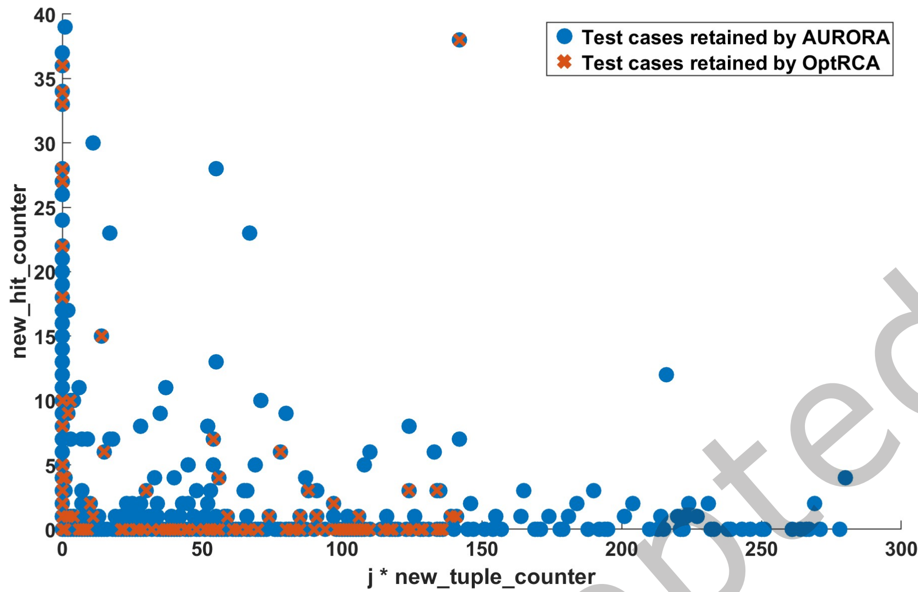


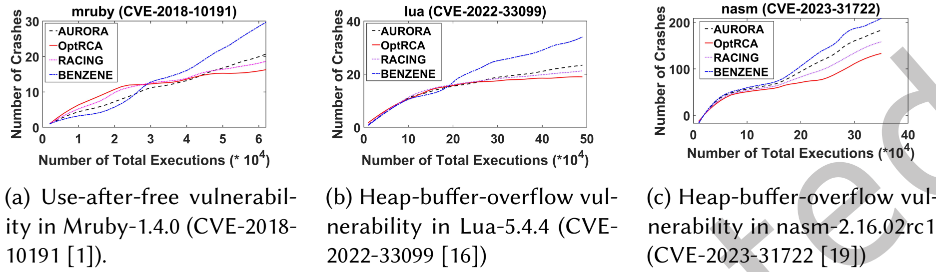
Fig. 4. The point distribution of non-crashing test cases with new paths (CVE-2018-10191 [1]). The total duration of fuzzing is 30 minutes. The abscissa represents the number of newly added path tuples in the execution path multiplied by the case counter of the tuple being changed ($j * new_tuple_counter$), and the ordinate represents the number of newly changed hit-counts for a particular path tuple in the execution path ($new_hit_counter$). The blue "•" are the non-crash test cases retained in the fuzzing phase of AURORA, and the red "×" represent the non-crash test cases retained in the fuzzing phase of OptRCA.

Just Accepted

the retained results of the non-crash test cases are shown in Figure 4. As can be seen from Figure 4, compared to AURORA, the test case distribution of OptRCA has three characteristics. First, the distribution range of OptRCA's non-crash test cases is smaller than AURORA in the dimension of $j * new_tuple_counter$. That is, if a non-crash test case's $j * new_tuple_counter$ is too large, OptRCA will give up retaining it. This is because if $j * new_tuple_counter$ is too large, it indicates that the path difference between this non-crash test case and the original crash seed is too large. This huge path difference shows that the correlation between it and the original crash is very small, so the help for root cause analysis is very limited. OptRCA abandons and retains it, which not only improves efficiency, but also has minimal impact on the accuracy of root cause analysis. Second, in the $new_hit_counter$ dimension, the distribution of OptRCA remains consistent with AURORA, but the distribution density is greatly reduced. As can be seen from the description in Section 3.2, $new_hit_counter$ represents the number of newly changed hit-counts for a particular path tuple in the execution path (hit-counts represents the number of times a specific tuple is accessed). Among the many types of bugs in software, only one type of bug is related to the access count, and that is overflow caused by too many accesses. However, this kind of overflow caused by too many accesses accounts for a very small proportion of the entire bug collection. Therefore, for bugs that are not access count overflow, the distribution density in the $new_hit_counter$ dimension has almost no impact on the accuracy of root cause analysis. The bug (CVE-2018-10191 [1]) shown in Figure 4 is not a bug with overflow of access count. Third, in the $new_hit_counter$ dimension, the distribution of OptRCA test cases is concentrated in the small $new_hit_counter$ range, which is also consistent with AURORA. This is because test

cases with small *new_hit_counter* bring a greater amount of new information. Take the access count overflow bug as an example. This type of bug is often caused by the overflow of access count for a specific tuple or several tuples. The small *new_hit_counter* already covers all tuples corresponding to the overflow of access count. In other words, for bugs with access count overflow, a large *new_hit_counter* test case does not bring a greater amount of information.

5.3 Crash Exploration



(a) Use-after-free vulnerability in Mruby-1.4.0 (CVE-2018-10191 [1]).

(b) Heap-buffer-overflow vulnerability in Lua-5.4.4 (CVE-2022-33099 [16])

(c) Heap-buffer-overflow vulnerability in nasm-2.16.02rc1 (CVE-2023-31722 [19])

Fig. 5. Number of unique crashes discovered by original AURORA, OptRCA, RACING and BENZENE fuzzing. The total duration of fuzzing is 30 minutes. The experiment was repeated 10 times, and all the curves in the figure are the average values of the data collected from 10 experiments.

This section mainly uses the crash growth curve to discuss OptRCA's ability to explore crashes. As described in Section 3.2, OptRCA's fuzzing strategy is based on optimizing the energy scheduling of seeds to maximize the correlation between seeds. Although the output of this fuzzing strategy is more suitable for root cause analysis, it will inevitably have a certain impact on the efficiency of crash exploration. In order to study this impact more deeply, we compare the crash exploration efficiency of OptRCA, RACING [54], BENZENE [42] and AURORA using three representative software bugs. Among them, RACING and BENZENE are two state-of-the-art root cause analysis techniques. The three small pictures in Figure 5 show the crash growth curve comparison of the four fuzzing solutions.

The horizontal axis of Figure 5 does not select the execution time of the program, but the number of executions of the program. This is because the fuzzing efficiency of the four schemes is consistent overall, that is, the average time of each iteration of the four schemes is roughly equal. One thing that needs to be emphasized is that although OptRCA modified the fuzzing code of AURORA, the average fuzz iteration time between the two is almost the same. This shows that the modification of the fuzz code does not significantly affect the efficiency of fuzzing. Not only that, the fuzzing strategies of RACING and BENZENE are also different from those of AURORA and OptRCA, but the average fuzz iteration time is not significantly different from that of AURORA and OptRCA. This is mainly because, in each iteration of fuzz, the execution time of the program accounts for the vast majority, while the seed scheduling and mutation time only occupies a small part. Therefore, the modification of seed scheduling and mutation will not significantly affect the average iteration time of fuzz. Through our observation of the results of a large number of repeated experiments, the factor affecting each iteration time is mainly the difference in the execution time of each input. Especially in the early stage of the fuzzing process, the difference in the number of iterations caused by the difference in execution time is still very large. When fuzzing lasts for a longer time (for example, more than 10 hours), this difference will gradually narrow to a negligible level. Therefore, the difference in the number of iterations caused by the difference in execution time can be regarded as a random error. However, in the scenario of automated root cause analysis, the duration of the fuzzing process of OptRCA and AURORA is relatively short. In the experiment shown in Figure 5, the total duration of fuzzing is

Just Accepted

Table 3. Number of crash, noise crash and non-crash test cases found by AURORA and OptRCA. #Total Executions refers to the number of times the program is executed during the fuzzing phase. #Crash means the number of crash test cases obtained during the fuzzing phase. # Noise Crash means the number of noise crash test cases obtained during the fuzzing phase. #Non-Crash represents the number of non-crash test cases obtained in the fuzzing phase. "↑" and "↓" represent the increase and decrease of quantity respectively. "=" represents equal quantity.

Target	#Total Executions	#Crash		# Noise Crash		#Non-Crash	
		AURORA	OptRCA	AURORA	OptRCA	AURORA	OptRCA
#1 mruby	65000	21	17↓	2	0 ↓	1169	212↓
#2 mruby	65000	21	12↓	0	0 =	377	101↓
#3 mruby	100000	26	11↓	3	0 ↓	1443	453↓
#4 mruby	50000	34	31↓	1	0 ↓	955	297↓
#5 mruby	75000	20	18↓	1	1 =	1413	567↓
#6 lua	50000	24	19↓	3	0 ↓	524	228↓
#7 lua	75000	22	12↓	2	1 ↓	604	264↓
#8 lua	120000	15	7↓	0	0 =	748	335↓
#9 php	20000	55	35↓	10	2 ↓	679	230↓
#10 php	20000	95	83↓	19	4 ↓	994	192↓
#11 php	20000	93	65↓	13	2 ↓	941	167↓
#12 nasm	20000	186	131↓	20	3 ↓	727	345↓
#13 nasm	50000	80	41↓	31	4 ↓	516	188↓
#14 nasm	10000	200	218 ↑	15	6 ↓	700	383↓
#15 nasm	20000	71	68↓	8	0 ↓	811	328↓
#16 objdump	100000	47	34↓	3	1 ↓	108	70↓
#17 objdump	100000	121	88↓	7	0 ↓	101	58↓
#18 nm-new	100000	140	150 ↑	26	3 ↓	229	140↓
#19 readelf	100000	23	37 ↑	0	0 =	223	109↓
#20 nm-new	100000	165	129↓	43	5 ↓	259	167↓

Just Accepted

only 30 minutes. This leads to a significant difference in the number of iterations in each experiment. In order to reduce the impact of this random error, we use the number of iterations as the horizontal axis. In addition, in order to further eliminate the impact of random errors, the experiment was conducted a total of 10 times. All the data in Figure 5 are the average of the collected data of 10 experiments. The purpose of repeating 10 times is to eliminate random errors as much as possible and obtain a smoother curve. These smooth curves can roughly infer the trend of crash exploration of each solution.

Among them, Figure 5a, Figure 5b and Figure 5c are respectively the fuzzing test results of mruby's use-after-free (CVE-2018-10191 [1]), lua's heap-buffer-overflow (CVE-2022-33099 [16]) and nasm's heap-buffer-overflow (CVE-2023-31722 [19]) respectively. As can be seen from the three small pictures in Figure 5, OptRCA's crash exploration efficiency is lower than that of the other three schemes. This is mainly because the fuzzing strategy that maximizes correlation has a certain impact on OptRCA. However, from the perspective of root cause analysis, OptRCA's crash test case is closer to the original crash seed than that of the other three solutions. In other words, OptRCA's crash test cases are more closely related to the original crash seeds and contain more information. Therefore, although the number of crash test cases discovered by OptRCA has decreased, the amount of information has not necessarily decreased. From an informatics perspective, OptRCA has a higher signal-to-noise ratio than that of the other three solutions, so it is more suitable for root cause analysis.

5.4 Accuracy of Root Cause Analysis

This section mainly compares the accuracy of AURORA and OptRCA. From an informatics perspective, the number of crash and non-crash test cases has a great impact on the accuracy of root cause analysis. On the other

hand, the number of noise crashes directly determines the signal-to-noise ratio of all crash test cases, which is positively correlated with the accuracy of root cause analysis. Therefore, before showing the results of root cause analysis, this section first provides a comparison of the number of crash, noise crash and non-crash test cases used in root cause analysis. Table 3 shows the comparison of the number of crash , noise crash and non-crash test cases obtained by AURORA and OptRCA respectively under the same number of program executions. In Table 3, the execution time of each fuzzing target is not fixed. The fuzzing time of each fuzzing target mainly depends on the time when OptRCA or AURORA get the best result. The best result means the best *#Predicates to BugFix* and *#Source Lines to BugFix* in Table 4. The minimum number of crash and non-crash test cases required to get the best result corresponds to the shortest time (minimum number of executions) required to get the best result. Obviously, the fuzzing time (number of executions) required for OptRCA and AURORA to get the best result is different. Suppose that for a certain fuzzing target, OptRCA performs better than AURORA. That is, compared with AURORA, OptRCA can get the best result with less fuzzing time (number of executions). With the same fuzzing time (number of executions), AURORA can only get worse results. For this fuzzing target, *#Total Executions* in Table 3 represents the minimum number of executions (shortest fuzzing time) required for OptRCA to get the best result. As can be seen from Table 3, overall, the number of non-crash test cases of OptRCA is significantly reduced compared to AURORA. For a specific target, the greater the number of non-crash test cases in AURORA, the greater the reduction in the number of non-crash test cases in OptRCA. This is mainly due to the non-crash hill-climbing retention method. When the number of retained non-crash test cases is small, the probability of new non-crash test cases being retained is still high. However, when the number of retained non-crash test cases is already large, the probability of new non-crash test cases being retained will be greatly reduced. Therefore, for a specific target, the greater the number of non-crash, the more obvious the optimization effect of OptRCA relative to AURORA. On the other hand, the number of crash test cases of OptRCA is likely to be less than that of AURORA. This is mainly due to the effect of maximizing correlation fuzzing strategies. However, there are three exceptions (#14, #18 and #19). The number of crash test cases for these three exceptions does not decrease but increases. The special feature of these three exceptions is that their crash distributions are concentrated near the original crash seeds, so maximizing correlation fuzzing can discover more crashes. The middle column of Table 3 compares the number of noise crashes. In general, the number of noise crashes of OptRCA is significantly reduced compared with AURORA. For all 20 targets in Table 3, the number of noise crashes of OptRCA is less than 10. When the number of noise crashes of AURORA is relatively small (less than 10), the number of noise crashes of OptRCA is close to or even equal to 0. This is enough to prove that the signal-to-noise ratio of the crash test case of OptRCA is much better than that of AURORA.

The result of the root cause analysis of AURORA and OptRCA is a candidate list. Each root cause candidate contains its own predicate and location. The definition of this predicate is as described in Section 2.4, and this location is the location of the source code line. The entire candidate list is arranged according to the accuracy of the predicate's crash and non-crash predictions. The more accurate this predicate is in predicting between crash and non-crash, the higher it will be ranked in the list. The accuracy of root cause analysis is also determined by the two factors predicate and location. First, the higher the correct root cause's predicate ranks in the candidate list, the more accurate the analysis results are. Second, after finding the correct predicate in the candidate list, the closer the candidate location is to the correct root cause location, the more accurate the result will be.

Table 4 shows the comparison of the accuracy of the root cause analysis results of AURORA and OptRCA, which takes the test cases shown in Table 3 as input. There are two most important indicators in Table 4 , *#Predicates to BugFix* and *#Source Lines to BugFix*. Among them, *#Predicates to BugFix* refers to the distance between the correct predicate and the location of the real bug fix. The correct predicate here refers to the predicate candidate closest to the location of the bug fix, not necessarily the location of the real root cause. This correct predicate may be in the same function as the location of the bug fix, or in the same *if else* conditional statement, or in the same *for* loop. It is called the correct predicate because when the developer sees the location of this predicate, he can easily

Just Accepted

Table 4. Results of root cause analysis and explanations (taking the test cases shown in Table 3 as input). #Predicates to BugFix represents the number of predicates that a human analyst needs to check untile the location is reached where the developers applied the bug fix. #Source Lines to BugFix represents the number of source lines that a human analyst needs to check untile the location is reached where the developers applied the bug fix. ↑ and ↓ represent the increase and decrease of quantity respectively.

Target	Root Cause	Crash Type	#Predicates to BugFix		#Source Lines to BugFix	
			AURORA	OptRCA	AURORA	OptRCA
#1 mruby	integer overflow	heap-use-after-free	1	1	32	32
#2 mruby	missing check	heap-buffer-overflow	2	3 ↑	6	6
#3 mruby	logic flaw	heap-buffer-overflow	66	31 ↓	1	1
#4 mruby	missing check	segmentation fault	67	38 ↓	6	6
#5 mruby	logic flaw	segmentation fault	22	12 ↓	10	10
#6 lua	logic flaw	heap-buffer-overflow	285	76 ↓	1	1
#7 lua	missing check	segmentation fault	157	34 ↓	1	5 ↑
#8 lua	type confusion	segmentation fault	3	5 ↑	1	1
#9 php	logic flaw	stack-buffer-overflow	263	57 ↓	66	66
#10 php	missing check	heap-buffer-overflow	2	4 ↑	1	1
#11 php	missing check	global-buffer-overflow	1	1	36	18 ↓
#12 nasm	missing check	heap-buffer-overflow	8	8	53	53
#13 nasm	logic flaw	heap-buffer-overflow	19	11 ↓	21	21
#14 nasm	logic flaw	double-free	25	17 ↓	14	14
#15 nasm	integer overflow	heap-use-after-free	224	95 ↓	99	99
#16 objdump	missing check	heap-buffer-overflow	1	1	7	7
#17 objdump	logic flaw	heap-buffer-overflow	1	1	18	18
#18 nm-new	null pointer dereference	segmentation fault	4	4	27	27
#19 readelf	logic flaw	stack-buffer-overflow	6	6	43	43
#20 nm-new	missing check	heap-use-after-free	4	1 ↓	81	81

guess the root cause of the bug. In fact, the definition of this correct predicate is not the first to be created by OptRCA, AURORA also uses a similar definition. The definition of *#Source Lines to BugFix*, as the name suggests, refers to the distance between the correct predicate and the location where the real bug is fixed. Overall, most of the arrows in Table 4 are downward, indicating that the root cause analysis accuracy of OptRCA is likely to be better than that of AURORA. In Table 4, the *#Source Lines to BugFix* of most targets are not significantly different, while more than half of the *#Predicates to BugFix* have changed. This result shows that OptRCA's optimization of the element content of the candidate list is very limited, but it greatly optimizes the order of the candidate list. A more in-depth analysis of the results in Table 4 shows that when the number of *#Predicates to BugFix* is very large, OptRCA can be optimized with a high probability. But if the number of *#Predicates to BugFix* is very small, OptRCA has no optimization effect or even worsens the results. In other words, when the root cause analysis result of AURORA is very poor (the number of *#Predicates to BugFix* is very large), the optimization effect of OptRCA is very obvious.

In general, OptRCA's accuracy is 65% higher than AURORA on average. OptRCA outperformed AURORA in 50% of the 20 vulnerabilities we evaluated. Meanwhile, OptRCA and AURORA were equally accurate in 35% of the vulnerabilities. OptRCA was less accurate than AURORA in only 15% of the vulnerabilities. In scenarios where AURORA analysis accuracy was very poor (#Predicates to BugFix was greater than 100), OptRCA improved its accuracy by an average of 71.8%. In scenarios where AURORA analysis accuracy was relatively poor (#Predicates to BugFix was between 50 and 100), OptRCA improved its accuracy by an average of 48%. However, in scenarios where AURORA analysis accuracy was relatively good (#Predicates to BugFix was less than 50), OptRCA improved its accuracy by an average of 24%.

Just Accepted

Table 5. Time spent on tracing, predicate analysis and ranking of each target (in seconds). The input is the crash and non-crash test cases shown in Table 3. ↑ and ↓ represent the increase and decrease of quantity respectively.

Target	Tracing		Predicate Analysis		Ranking		Total Time	
	AURORA	OptRCA	AURORA	OptRCA	AURORA	OptRCA	AURORA	OptRCA
#1 mruby	1004	213	264	62	59	89	1327	364↓
#2 mruby	206	87	100	39	17	12	323	138↓
#3 mruby	973	292	320	113	75	87	1368	492↓
#4 mruby	701	250	238	76	41	117	980	443↓
#5 mruby	762	307	284	115	29	120	1075	542↓
#6 lua	86	36	60	31	90	111	236	178↓
#7 lua	91	53	65	35	11	32	167	120↓
#8 lua	115	54	87	49	29	39	231	142↓
#9 php	638	220	306	98	93	150	1037	468↓
#10 php	890	226	515	146	134	50	1539	422↓
#11 php	950	204	466	173	150	60	1566	437↓
#12 nasm	28	14	31	19	1	16	60	49↓
#13 nasm	18	7	16	9	1	6	35	23↓
#14 nasm	28	19	24	16	1	1	53	36↓
#15 nasm	28	13	23	15	1	1	52	29↓
#16 objdump	4	3	2	2	1	1	7	6↓
#17 objdump	5	3	4	3	1	1	10	7↓
#18 nm-new	7	5	7	4	1	1	15	10↓
#19 readelf	4	2	4	3	1	1	9	6↓
#20 nm-new	8	5	7	4	1	1	16	10↓

5.5 Efficiency of Root Cause Analysis

This section compares the root cause analysis efficiency of OptRCA and AURORA. From the technical details introduced in Sections 2.4 and 3.2, it can be seen that after collecting enough crash and non-crash test cases in the fuzzing phase, AURORA and OptRCA will enter the analysis phase. The entire analysis stage is divided into three small steps, namely Tracing, Predicate Analysis and Ranking. Tracing step refers to using Intel Pin tool to monitor the execution flow of each test case and collect information. Predicate Analysis step refers to using Equation 2 and 3 or Equation 10 to calculate the prediction accuracy of each predicate for crash and non-crash. Ranking step refers to ranking each candidate using the prediction accuracy and position of each predicate. Table 5 shows the comparative experimental results of the total time consumption of root cause analysis and the time consumption of each step. As can be seen from Table 5, the total time consumption of OptRCA is much better than AURORA. For a specific target, the more time AURORA consumes, the more significant the optimization effect of OptRCA is. The main reason for this experimental result is that Tracing step takes up most of the time. Furthermore, the biggest factor affecting the time consumption of Tracing is the number of test cases. From the experimental results in Table 3, we can know that OptRCA's non-crash hill-climbing retention method greatly reduces the number of non-crash test cases. Therefore, OptRCA greatly reduces the time consumption of the Tracing step. In Table 5 we can also find that the time consumption of Ranking step of OptRCA is mostly greater than that of AURORA. This is mainly because the improvement of OptRCA's analysis accuracy may lead to more elements in the candidate list. Sorting more elements will inevitably consume more time. Therefore, the ranking time of OptRCA will most likely be longer than that of AURORA. However, since Ranking step consumes a small proportion of the time in the entire analysis phase, it does not affect OptRCA's efficiency optimization too much.

In general, OptRCA is significantly more efficient than AURORA, which is also OptRCA's biggest advantage. In the tracing stage, OptRCA is 69% higher than AURORA on average. In the predicate analysis stage, OptRCA is 64% higher than AURORA on average. In the ranking stage, OptRCA performs worse than AURORA, 18% lower than AURORA on average. Among all 20 evaluated vulnerabilities, the number of vulnerabilities where OptRCA's ranking efficiency is lower than AURORA accounts for 50% of the total, while the number of vulnerabilities

Just Accepted

where OptRCA's ranking efficiency is better than AURORA accounts for only 15% of the total. If these three stages are combined, OptRCA is 61% higher than AURORA on average.

5.6 Accuracy and Efficiency Comparison with State-of-the-Art Solutions

In order to more accurately study the efficiency and accuracy of OptRCA, we run the root cause analysis process of the four solutions OptRCA, RACING, BENZENE and AURORA respectively, and the results are shown in Table 6 below. From Table 6, we can see that in general, in terms of accuracy (#Predicates to BugFix), BENZENE and OptRCA perform best. Among the 20 analysis targets, BENZENE's accuracy is better than OptRCA's in 12 targets. However, OptRCA's accuracy is better than BENZENE's in only 5 targets. Therefore, BENZENE's accuracy performance is better than OptRCA. A careful comparison of the differences between OptRCA and BENZENE also reveals that when the analysis results of OptRCA and BENZENE are both poor, the results of OptRCA are relatively better than those of BENZENE. When the analysis results of OptRCA and BENZENE are both good, the results of BENZENE are better than those of OptRCA. In other words, when the root cause of the target is difficult to analyze, OptRCA has more advantages. When the root cause of the target is easier to analyze, BENZENE has stronger capabilities. Among the results of the four schemes, the result of RACING is slightly worse than that of AURORA. This is mainly because RACING's counterexample mechanism based on reinforcement learning reduces the number of crash and non-crash test cases collected. The reduction in the number of test cases affects the accuracy of root cause analysis.

Table 6. Accuracy and efficiency comparison with state-of-the-art solutions. The number of executions and execution time are the same as those in Table 3. #Predicates to BugFix represents the number of predicates that a human analyst needs to check until the location is reached where the developers applied the bug fix. Total Analysis Time represents the total time of Tracing + Predicate Analysis + Ranking.

Target	#Predicates to BugFix				Total Analysis Time (in seconds)			
	OptRCA	RACING	BENZENE	AURORA	OptRCA	RACING	BENZENE	AURORA
#1 mruby	1	3	1	1	364	33	632	1327
#2 mruby	3	8	1	2	138	12	254	323
#3 mruby	31	69	10	66	492	76	751	1368
#4 mruby	38	49	9	67	443	70	667	980
#5 mruby	12	27	5	22	542	358	813	1075
#6 lua	76	195	126	285	178	25	212	236
#7 lua	34	214	49	157	120	64	180	167
#8 lua	5	3	1	3	142	39	205	231
#9 php	57	278	125	263	468	81	594	1037
#10 php	4	10	2	2	422	127	732	1539
#11 php	1	1	1	1	437	92	746	1566
#12 nasm	8	12	4	8	49	13	38	60
#13 nasm	11	21	1	19	23	10	36	35
#14 nasm	17	31	6	25	36	9	45	53
#15 nasm	95	246	173	224	29	11	46	52
#16 objdump	1	2	2	1	6	5	18	7
#17 objdump	1	1	1	1	7	7	19	10
#18 nm-new	4	2	1	4	10	7	25	15
#19 readelf	6	20	3	6	6	5	17	9
#20 nm-new	4	15	1	1	10	6	21	16

From the perspective of efficiency (Total Analysis Time), the RACING solution is far ahead. OptRCA is not as efficient as RACING, but much better than AURORA and BENZENE. RACING is the most efficient, as it should be. Because RACING's counterexample mechanism based on reinforcement learning greatly reduces the number of non-crash and crash test cases that need to be analyzed, its analysis speed is the fastest. OptRCA's efficiency performance is second only to RACING, also because of its fuzzing strategy of maximizing correlation and non-crash hill-climbing retention strategy, which reduces the number of test cases. BENZENE's efficiency is only

Just Accepted

slightly better than AURORA, and much worse than RACING and OptRCA. This is because BENZENE's biggest focus is on the seed mutation strategy, and it does not focus on the selection of test cases. Therefore, BENZENE needs to analyze more test cases than RACING and OptRCA, which requires longer analysis time.

In order to more intuitively see the advantages of OptRCA, we conducted a quantitative analysis of the accuracy and efficiency of these four solutions. In the test of all 20 vulnerabilities, OptRCA's accuracy is 66% higher than RACING on average, and 28% higher than BENZENE. Compared with RACING, the number of vulnerabilities where OptRCA's accuracy is better than RACING accounts for 80% of the total, 10% is equal to RACING, and 10% is worse than RACING. Compared with BENZENE, the number of vulnerabilities where OptRCA's accuracy is better than BENZENE accounts for 25% of the total, 15% is equal, and 60% is worse than BENZENE. In terms of efficiency, OptRCA is 274% slower than RACING and 35% faster than BENZENE. Among all 20 vulnerabilities tested, the number of vulnerabilities where OptRCA is slower than RACING accounts for 100%, and the number of vulnerabilities where OptRCA is faster than BENZENE accounts for 100%.

5.7 Other Statistical Analysis

Table 7. Results of Mann-Whitney U test and Vargha Delaney's A Measure. The data collected in this table is the same as Figure 5, which is the crash exploration curve. The collection lasted for 30 minutes, repeated 10 times and averaged. In this table, OPT-RAC refers to the statistical analysis results between OptRCA and RACING. Similarly, OPT-BEN refers to the statistical analysis results between OptRCA and BENZENE. OPT-AUR refers to the statistical analysis results between OptRCA and AURORA.

Target	Mann-Whitney U test (p-value)			Vargha Delaney's A Measure (A)		
	OPT-RAC	OPT-BEN	OPT-AUR	OPT-RAC	OPT-BEN	OPT-AUR
#1 mruby	0.0090	0.0055	0.0077	0.4610	0.4258	0.4524
#2 mruby	0.0294	0.0041	0.0188	0.4407	0.1830	0.3579
#3 mruby	0.0371	0.0152	0.0207	0.4602	0.2869	0.4283
#4 mruby	0.0487	0.0093	0.0379	0.4621	0.3048	0.4310
#5 mruby	0.0074	0.0036	0.0048	0.4143	0.1949	0.3817
#6 lua	0.0157	0.0001	0.0143	0.4169	0.2745	0.4140
#7 lua	0.0325	0.0090	0.0193	0.3869	0.2056	0.2995
#8 lua	0.0289	0.0014	0.0129	0.4648	0.2803	0.3483
#9 php	0.0336	0.0008	0.0210	0.4945	0.2914	0.4077
#10 php	0.0321	0.0059	0.0195	0.4007	0.2638	0.3063
#11 php	0.0316	0.0030	0.0222	0.4504	0.1139	0.2490
#12 nasm	0.0154	0.0006	0.0070	0.4567	0.2587	0.3720
#13 nasm	0.0106	0.0003	0.0067	0.3693	0.1234	0.3332
#14 nasm	0.0228	0.0051	0.0153	0.4454	0.0417	0.3130
#15 nasm	0.0413	0.0058	0.0169	0.4642	0.2900	0.3845
#16 objdump	0.0044	0.0026	0.0036	0.3149	0.1047	0.2761
#17 objdump	0.0198	0.0005	0.0144	0.3074	0.0160	0.1812
#18 nm-new	0.0381	0.0023	0.0253	0.2448	0.0615	0.1812
#19 readelf	0.0398	0.0031	0.0149	0.2693	0.0963	0.1409
#20 nm-new	0.0447	0.0027	0.0121	0.3265	0.1027	0.2452

The *Mann-Whitney U Test* is used to test whether two sets of independent samples come from the same population distribution. Because it does not require the data to obey a normal distribution, it is suitable for non-normally distributed data. The *p-value* of the Mann-Whitney U Test is one of its core outputs and is used to

Just Accepted

determine whether there is a significant difference between two sets of data. The *p-value* represents the likelihood of observing more extreme data, assuming that the two sets of data samples come from the same distribution. If the *p-value* is less than 0.05, we have reason to believe that the distributions of the two sets of data samples are significantly different. If the *p-value* is greater than 0.05, we consider that there is no significant difference in the distribution of the two sets of data samples. *Vargha-Delaney's A Measure* is a non-parametric measure of effect size that is used to compare the magnitude and direction of differences between two independent samples. When the A value of *Vargha-Delaney's A Measure* is equal to 0.5, it indicates that there is no difference between the two sets of sample data (the distribution is the same). When the value of A is greater than 0.5, it indicates that the value of the previous sample tends to be greater than the latter sample. When the value of A is less than 0.5, it indicates that the value of the previous sample tends to be smaller than the latter sample. The smaller the A value, the greater the probability that the value of the previous sample is smaller than the value of the latter sample. *Vargha-Delaney's A Measure* is a flexible and intuitive effect size measurement method, which is particularly suitable for non-normally distributed data scenarios. More importantly, *Vargha-Delaney's A Measure* is particularly suitable for use with the *Mann-Whitney U Test* to quantify the actual meaning of the difference, rather than just its significance.

The statistical results of the *Mann-Whitney U test* and *Vargha Delaney's A Measure* between OptRCA and RACING, BENZENE and AURORA are shown in Table 7. The collected data are all crash exploration curves. The time and number of collections are the same as Figure 5. As can be seen from Table 7, the *p-value* of the *Mann-Whitney U test* between OptRCA and the other three schemes are all less than 0.05. In other words, the distributions of OptRCA and the other three schemes are significantly different. A more precise comparison can be found that the *p-value* between OptRCA and BENZENE is the smallest, followed by the *p-value* between OptRCA and AURORA. The *p-value* between OptRCA and RACING is the largest. In other words, the difference between OptRCA and BENZENE is the most significant, followed by OptRCA and AURORA. The difference between OptRCA and RACING is the least significant. On the other hand, the A values of *Vargha Delaney's A Measure* in Table 7 are all less than 0.5. This shows that the data distribution of OptRCA tends to be lower than that of the other three schemes. In other words, the data distribution of OptRCA is the lowest among the four schemes. Through a more detailed comparison, it can be found that the A value between OptRCA and RACING is the largest, followed by OptRCA and AURORA. The smallest A value is between OptRCA and BENZENE. In other words, the data distribution of OptRCA is the lowest, followed by RACING. The highest data distribution is BENZENE, and the second highest is AURORA.

Just Accepted

5.8 Ablation Study

In order to understand the impact of each module of OptRCA on efficiency and accuracy, we conducted a detailed ablation study on OptRCA. We selected three vulnerabilities as research samples for experiments. For each vulnerability, we tested it with 5 different implementation strategies. The first strategy is the complete version of OptRCA, which includes all improved modules. The second strategy is *OptRCA without Maximizing Correlation Fuzzing*, which replaces the fuzzing module of OptRCA with the original AURORA fuzzing module. The third strategy is *OptRCA without Hill-Climbing Retention*, which replaces the non-crash retention module of OptRCA with the non-crash retention module of the original AURORA. The fourth strategy is *OptRCA without Optimized Calculation Formula*, which replaces the root cause analysis calculation formula of OptRCA with the original AURORA analysis calculation formula. The fifth strategy is the original AURORA solution. Table 8 shows the experimental results of our ablation study.

As can be seen from Table 8, in general, the three modules have an impact on accuracy and efficiency. Among them, the *Maximizing Correlation Fuzzing* module can greatly improve accuracy, the *Hill-Climbing Retention* module can greatly improve efficiency and accuracy, and the *Optimized Calculation Formula* module can slightly

Table 8. Result of OptRCA's ablation studies. In the table, *#Predicates to BugFix* represents the number of predicates that a human analyst needs to check until the location is reached where the developers applied the bug fix. *Time* indicates the total time taken for the analysis process. *OptRCA without Maximizing Correlation Fuzzing* means replacing the optimized fuzzing strategy of OptRCA with the original AURORA fuzzing strategy. *OptRCA without Hill-Climbing Retention* means replacing the optimized non-crash retention strategy of OptRCA with the original non-crash retention strategy of AURORA. *OptRCA without Optimized Calculation Formula* means replacing the optimized analysis calculation formula of OptRCA with the original AURORA calculation formula.

Testing Strategy	#4 mruby		#7 lua		#9 php	
	Time	#Predicates to BugFix	Time	#Predicates to BugFix	Time	#Predicates to BugFix
OptRCA	443	38	120	34	468	57
OptRCA without Maximizing Correlation Fuzzing	420	62	114	131	389	396
OptRCA without Hill-Climbing Retention	997	49	189	55	856	80
OptRCA without Optimized Calculation Formula	445	39	121	38	487	62
AURORA	980	67	167	157	1037	463

improve efficiency and accuracy. After quantitative analysis of the three modules, more detailed results can be obtained. First, by comparing the results of *OptRCA* and *OptRCA without Maximizing Correlation Fuzzing*, we can see the impact of OptRCA's *Maximizing Correlation Fuzzing* module on efficiency and accuracy. After averaging the results of the three vulnerabilities, it can be seen that the *Maximizing Correlation Fuzzing* module improves accuracy by 78% and reduces efficiency by 12% on average. Second, by comparing the results of *OptRCA* and *OptRCA without Hill-Climbing Retention*, we can see the impact of OptRCA's *Hill-Climbing Retention* module on efficiency and accuracy. After averaging the results of the three vulnerabilities, it can be seen that the *Hill-Climbing Retention* module improves accuracy by 30% and efficiency by 49% on average. Third, by comparing the results of *OptRCA* and *OptRCA without Optimized Calculation Formula*, we can see the impact of OptRCA's *Optimized Calculation Formula* module on efficiency and accuracy. After averaging the results of the three vulnerabilities, it can be seen that the *Optimized Calculation Formula* module improves accuracy by 7% and efficiency by 2% on average.

6 RELATED WORK

Statistical root cause analysis. Many researchers focus on using statistical methods to locate faults or root causes. Santelices *et al.* [45] designed a coverage-based fault localization approach that can reduce the analysis overhead by combining the advantages of each coverage type. Baudry *et al.* [23] proposed a new test-for-diagnosis criterion based on Dynamic Basic Block, which solved the contradiction between the number of test cases and the accuracy of fault location. Hao *et al.* [37] designed three strategies that can ensure that the accuracy of fault location is close to the original results while reducing 90% of test cases. Based on experimental studies, Abreu *et al.* [21] found that more crash test cases can bring higher efficiency of fault location analysis. Research by Yu *et al.* [60] shows that different test case reduction strategies have different effects on the effectiveness of fault location. Blazytko *et al.* [24] proposed AURORA, which is a tool based on statistical methods to analyze root causes. Different from other technologies, AURORA can not only locate the root cause, but also provide explanation information of the root cause. Xu *et al.* [53] proposed RACING, which combines reinforcement learning with AURORA technology to greatly improve the efficiency of root cause analysis.

Non-statistical root cause analysis. Researchers have also proposed some techniques that use non-statistical methods (data flow and control flow analysis, core dump analysis, symbolic execution, etc.) to analyze the root cause. Xu *et al.* [55] proposed the CREDAL tool, which combines the source code of crashing programs with core dump analysis to automatically locate memory corruption vulnerabilities. The DEEPVSA approach designed by Guo *et al.* [36] uses deep learning to expand the ability of value set analysis to better locate memory alias bugs.

Just Accepted

ARCUS is an automated root cause analysis tool designed by Yagemann *et al.* [59] based on symbolic execution. It can automatically locate bug root cause in executions flagged by the end-host monitor and demonstrably block them at the binary level. In addition to ARCUS, Yagemann *et al.* also proposed another tool called Bunkerbuster [57]. It utilizes symbolically reconstructed states based on execution traces to better conduct bug location and root cause analysis. Its analysis target mainly focuses on overflow, use-after-free, double free, and format string bugs of user programs and their imported libraries. He *et al.* [38] proposed FreeWill, a root cause analysis tool specifically for UAF bugs. FreeWill detects reference miscounting by identifying UAF objects and related references from execution traces and comparing them with their proposed omission-aware counting model.

7 DISCUSSION

Although OptRCA has significantly improved compared to AURORA, it still has two defects that needs to be improved. First, OptRCA can only improve the accuracy of *#Predicates to BugFix*, but the improvement of *#Source Lines to BugFix* is very limited. In fact, as can be seen from Table 4, for *#Source Lines to BugFix*, the number of vulnerabilities in which OptRCA is better than AURORA is only 1. In other words, the results of OptRCA can only save developers time when looking for the correct predicate. After confirming the correct predicate, it is still difficult to find the source code line that needs fixing. This defect seriously affects the optimization effect of OptRCA. Through deeper analysis, we found that the root cause that affects the accuracy of *#Source Lines to BugFix* is the collection of control flow and data flow. The analysis framework used by OptRCA is based on an improvement of AURORA, so the data flow and control flow it collects are the same as AURORA. In order to balance efficiency, OptRCA, like AURORA, does not collect enough control flow and data flow information. For example, in the memory-related predicate, only the memory values of a very small number of load and store instructions are selected, while other memory value changes are ignored. It is especially worth mentioning that the stack operation instructions related to memory are not within the information collection scope of OptRCA and AURORA. This makes OptRCA, like AURORA, collect very limited control flow and data flow information. In future work, we plan to improve OptRCA's control flow and data flow collection methods so that it can track more data dependencies and control flow information, thereby improving the accuracy of *#Source Lines to BugFix*.

Second, compared with AURORA, OptRCA greatly improves the problem of imbalanced number of crash and non-crash test cases, but it is still not perfect. From the results of the ablation study, it can be seen that the optimization formula of OptRCA can indeed improve a certain degree of accuracy, but the improvement is very limited. From Table 3, we can see that for nearly half of the vulnerability examples, the number gap between crash and non-crash is close to one order of magnitude. The current work does not evaluate whether this gap close to one order of magnitude will seriously affect the accuracy. In future work, we hope to improve the design and evaluation of OptRCA from two aspects. On the one hand, for the accuracy difference between Formula 1 and Formula 2, we hope to find a more intuitive evaluation method. On the other hand, for the imbalanced test cases close to one order of magnitude, we hope to find a more accurate calculation formula.

8 CONCLUSION

In this paper, we introduce OptRCA, a more efficient and accurate root cause analysis and explanation tool. Based on the seed energy scheduling for maximizing correlation, OptRCA can obtain crash test cases caused by the same root cause with a greater probability. Therefore, subsequent root cause analysis can get more accurate results. The hill-climbing retention method greatly reduces the number of non-crash test cases, thus making the subsequent root cause analysis process faster and more efficient. Not only that, the hill-climbing retention method also balances the quantitative difference between crash and non-crash, allowing OptRCA to use a more realistic analysis formula, thereby further improving the accuracy of the analysis results. Experimental evaluation results show that OptRCA is not only faster than AURORA, but also has more accurate analysis results.

Just Accepted

REFERENCES

[1] 2018. *Use after free caused by integer overflow in environment stack #3995*. https://github.com/mruby/mruby/issues/3995
[2] 2020. *Bug 3392707 - heap-use-after-free in saa_wbytes nasmlib/saa.c:132*. https://bugzilla.nasm.us/show_bug.cgi?id=3392707
[3] 2020. *Bug 3392712 - double-free in pp_tokline asm/preproc.c:6750*. https://bugzilla.nasm.us/show_bug.cgi?id=3392712
[4] 2020. *Heap buffer overflow in mruby interpreter #5042*. https://github.com/mruby/mruby/issues/5042
[5] 2020. *Sec Bug #79037 global buffer-overflow in 'mbfl_filt_conv_big5_wchar'*. https://bugs.php.net/bug.php?id=79037
[6] 2020. *Sec Bug #79099 OOB read in php_strip_tags_ex*. https://bugs.php.net/bug.php?id=79099
[7] 2020. *Sec Bug #79371 mb_strtolower (UTF-32LE): stack-buffer-overflow at php_unicode_tolower_full*. https://bugs.php.net/bug.php?id=79371
[8] 2020. *Segmentation fault at mrb_vm_exec #4973*. https://github.com/mruby/mruby/issues/4973
[9] 2020. *Segmentation fault in Mruby interpreter making it crash #5046*. https://github.com/mruby/mruby/issues/5046
[10] 2021. *Bug 26929 - [readelf] crash with ASAN in print_dynamic_symbol*. https://sourceware.org/bugzilla/show_bug.cgi?id=26929
[11] 2021. *Bug 26931 - [nm] crash with ASAN in display_rel_file*. https://sourceware.org/bugzilla/show_bug.cgi?id=26931
[12] 2021. *Heap Overflow in mruby #5316*. https://github.com/mruby/mruby/issues/5316
[13] 2021. *Vulnerability Details : CVE-2021-44647*. https://www.cvedetails.com/cve/CVE-2021-44647/
[14] 2022. *Bug 29699 - Segmentation fault caused by null pointer dereference in nm-new, _bfd_elf_get_symbol_version_string, elf.c:1969*. https://sourceware.org/bugzilla/show_bug.cgi?id=29699
[15] 2022. *Bug 3392815 - heap-buffer-overflow asm/nasm.c:856 in quote_for_pmake*. https://bugzilla.nasm.us/show_bug.cgi?id=3392815
[16] 2022. *Vulnerability Details : CVE-2022-33099*. https://www.cvedetails.com/cve/CVE-2022-33099/
[17] 2023. *Bug 29988 - AddressSanitizer: heap-buffer-overflow /binutils-gdb/bfd/libbfd.c:784 in bfd_getl64*. https://sourceware.org/bugzilla/show_bug.cgi?id=29988
[18] 2023. *Bug 30285 - objdump heap-buffer-overflow in _bfd_elf_print_private_bfd_data() at /binutils-gdb/bfd/elf.c:1844 (SIGSEGV)*. https://sourceware.org/bugzilla/show_bug.cgi?id=30285
[19] 2023. *Bug 3392857 - Heap-Buffer-Overflow in NASM(asm/preproc.c:6863 in expand_mmacro)*. https://bugzilla.nasm.us/show_bug.cgi?id=3392857
[20] 2023. *Bug: Loading a corrupted binary file can segfault*. https://github.com/lua/lua/commit/ab859fe59b464a038a45552921cb2b23892343af
[21] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/J.JSS.2009.06.035
[22] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/
[23] Benoit Baudry, Franck Fleurey, and Yves Le Traon. 2006. Improving test suites for efficient fault localization. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 82–91. https://doi.org/10.1145/1134285.1134299
[24] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 235–252. https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko
[25] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2022. *Aurora: Statistical Crash Analysis for Automated Root Cause Explanation*. https://github.com/RUB-SysSec/aurora
[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020
[27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1032–1043. https://doi.org/10.1145/2976749.2978428
[28] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 725–741. https://doi.org/10.1109/SP.2015.50
[29] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725. https://doi.org/10.1109/SP.2018.00046
[30] Intel Corporation. 2023. *Intel 64 and IA-32 Architectures Software Developer Manuals*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[31] Intel Corporation. 2023. *Pin - A Dynamic Binary Instrumentation Tool*. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

Just Accepted

[32] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

[33] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 820–831. https://doi.org/10.1145/2884781.2884844

[34] Free Software Foundation. 2023. *GNU Binutils*. https://www.gnu.org/software/binutils/

[35] Google. 2016. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[36] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1787–1804. https://www.usenix.org/conference/usenixsecurity19/presentation/guo

[37] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. 2010. Test input reduction for result inspection to facilitate fault localization. *Autom. Softw. Eng.* 17, 1 (2010), 5–31. https://doi.org/10.1007/S10515-009-0056-X

[38] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. 2022. FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2497–2512. https://www.usenix.org/conference/usenixsecurity22/presentation/he-liang

[39] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 782–792. https://doi.org/10.1145/2786805.2786844

[40] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[41] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 627–637. https://doi.org/10.1145/3106237.3106295

[42] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. 2024. Benzene: A Practical Root Cause Analysis System with an Under-Constrained State Mutation. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 1865–1883. https://doi.org/10.1109/SP54263.2024.00074

[43] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 697–710. https://doi.org/10.1109/SP.2018.00056

[44] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[45] Raúl A. Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 56–66. https://doi.org/10.1109/ICSE.2009.5070508

[46] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 157. https://doi.org/10.1109/SECDEV.2016.043

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[48] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, 46–55. https://doi.org/10.1109/CGO.2015.7054186

Just Accepted

[49] 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 *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[50] Robert Swiecki. 2023. *Honggfuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based)*. https://github.com/google/honggfuzz

[51] Wikipedia. 2023. *Core dump*. https://en.wikipedia.org/wiki/Core_dump

[52] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 511–522. https://doi.org/10.1145/2508859.2516736

[53] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. [n. d.]. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples. ([n. d.]).

[54] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. 2024. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4229–4246. https://www.usenix.org/conference/usenixsecurity24/presentation/xu-dandan

[55] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 529–540. https://doi.org/10.1145/2976749.2978340

[56] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 17–32. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-jun

[57] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 320–336. https://doi.org/10.1145/3460120.3485363

[58] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1989–2006. https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann

[59] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1989–2006. https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann

[60] Yanbing Yu, James A. Jones, and Mary Jean Harrold. 2008. An empirical study of the effects of test-suite reduction on fault localization. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 201–210. https://doi.org/10.1145/1368088.1368116

[61] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[62] Michael Zalewski. 2014. *afl-fuzz: crash exploration mode*. https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html

[63] Michał Zalewski. 2015. *American Fuzzy Lop (AFL)*. https://lcamtuf.coredump.cx/afl/

[64] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[65] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 2022. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 417–428. https://doi.org/10.1145/3533767.3534414

Received 15 May 2024; revised 2 April 2025; accepted 6 May 2025

Just Accepted