Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples

Dandan Xu[1,2], Di Tang[3], Yi Chen[3]*, XiaoFeng Wang[3], Kai Chen[1,2]*, Haixu Tang[3], Longxing Li[1,2]

[1]*SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China*

[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*

[3]*Indiana University Bloomington*

{xudandan, chenkai, lilongxing}@iie.ac.cn, {tangd, chen481, xw7, hatang}@iu.edu

Abstract

Root-Cause Analysis (RCA) is crucial for discovering security vulnerabilities from fuzzing outcomes. Automating this process through triaging the crashes observed during the fuzzing process, however, is considered to be challenging. Particularly, today's statistical RCA approaches are known to be exceedingly slow, often taking tens of hours or even a week to analyze a crash. This problem comes from the biased sampling such approaches perform. More specifically, given an input inducing a crash in a program, these approaches sample around the input by mutating it to generate new test cases; these cases are used to fuzz the program, in a hope that a set of program elements (blocks, instructions or predicates) on the execution path of the original input can be adequately sampled so their correlations with the crash can be determined. This process, however, tends to generate the input samples more likely causing the crash, with their execution paths involving a similar set of elements, which become less distinguishable until a large number of samples have been made. We found that this problem can be effectively addressed by sampling around "counterexamples", the inputs causing a significant change to the current estimates of correlations. These inputs though still involving the elements often do not lead to the crash. They are found to be effective in differentiating program elements, thereby accelerating the RCA process. Based upon the understanding, we designed and implemented a reinforcement learning (RL) technique that rewards the operations involving counterexamples. By balancing random sampling with the exploitation on the counterexamples, our new approach, called RACING, is shown to substantially elevate the scalability and the accuracy of today's statistical RCA, outperforming the state-of-the-art by more than an order of magnitude.

*Corresponding authors

†State Key Laboratory of Information Security, IIE, CAS

1 Introduction

With the pervasiveness of software systems, their security quality needs to be ensured in a scalable way. For this purpose, *automated vulnerability discovery* has been intensively studied in past decades [30, 42, 48, 51]. Among all techniques being proposed, fuzzing is no doubt the most successful one, which has been credited with discovery of the most consequential security flaws, such as CVE-2017-5380 [2], CVE-2018-4145 [3] and CVE-2022-36320 [4]. A problem here, however, is that fuzzing alone does not find a vulnerability: it just induces crashes of the target program with test cases it produces, while leaving triage of the crashes, a critical step to identify the *root cause* – the vulnerability behind the crashes, to the human analyst. Manual *root-cause analysis* (RCA) is a slow and pains-taking process, particularly for today's complicated software system involving subtle connections across different components, so the program locations where the crashes occur may not have explicit data dependencies upon the true cause of the crashes. As a result, vulnerability discovery tends to be significantly delayed even after the flaws have been reached by test cases, raising a strong demand for the techniques that enable automated, highly scalable RCA.

Challenges in automated RCA. Today, most RCAs have been done manually with the help of some analysis tools. These tools bucket different crash inputs for identifying the underlying vulnerability [27], or perform reverse execution and backward taint analysis [16, 17, 45] to trace back to the program location where the vulnerability is introduced, from the location where the crash is observed. These approaches rely on a set of manually drafted rules to identify some specific types of vulnerabilities and become less effective in the absence of explicit data flows between a vulnerability and observed crashes, such as the case of type-confusion flaws [21].

A more generic alternative capable of handling different types of vulnerabilities, including those without explicit data dependency on the crash they induce, is *Spectrum-based Fault Localization* (*SFL*), a set of statistical, ruleless, fully automated RCA techniques known to outperform other automated

RCA supports [18]. These techniques aim at capturing the fundamental causal relations between a crash and its root cause (a vulnerable program entity like blocks, statements or the predicate of a statement) based upon their statistical correlation. More specifically, for each crash observed from a program, an SFL technique generates a set of test cases to fuzz the target program, and uses the fuzzing outcome (either leading to the crash or not) to estimate the statistical correlation between each program entity encountered during the fuzzing process and the crash; the entities found to be highly correlated are then ranked and output as suspicious root causes [12, 39, 52]. With its generality and effectiveness, the approach is known to be heavyweight, incurring a massive overhead, due to the requirement to fuzz the program on a large number of test cases for establishing the correlation for each entity and also differentiating their contributions to the crash. Indeed, prior research reports that analysis on each crash through SFL could take 12 hours [12] or even a week [52], rendering these techniques hard to scale in the presence of a large number of crashes induced by fuzzing.

RCA with counterexample based RL. To move SFL closer to practical use, its performance needs to be elevated. In our research, we found that a fundamental weakness of existing SFL approaches is their sole reliance on the guidance of known inputs leading to the crash: by mutating these crash inputs for new test cases, program entities potentially related to the crash can be repeatedly sampled; however, these new test cases tend to be biased toward the crash and cannot effectively differentiate these entities in terms of their contributions to the crash. As a result, a large number of random inputs need to be produced and tested before enough distinguishing outcomes (particularly those covering repeatedly sampled program entities related to the crash but leading to non-crashing results) are observed so the root-cause entity can stand out of the crowd. This weakness can be addressed by sampling around the differentiating test cases discovered, which we call *counterexamples* – the test cases that cause a significant change to an entity's putative correlation with the crash or the putative order of the entities (based upon their contributions) estimated on the outcomes of previous fuzzing rounds. These counterexamples balance the sampling strategy of the existing approaches (e.g., Aurora [12]), which is biased toward crash inputs, and therefore facilitate identification of the root cause.

Based upon the understanding, we designed and implemented a high-performance, scalable SFL-based RCA technique, using *reinforcement-learning* (*RL*) to dynamically adjust the sampling (fuzzing) strategy based upon the estimates of program predicates' relations with a given crash. Our approach, called RACING (Root-cAuse-analysis on Counterexamples based reinforcement-learnING), utilizes the outcome of each fuzzing round, including its impacts on each predicate's estimated contribution to the crash and the estimated rankings of these predicates based upon their contributions, to guide the next round of fuzzing. More specifically, given the test cases known to cause a crash, our approach first performs random mutations on them (as done in the prior work [51]) and fuzzes the target program using these mutations to generate a baseline estimate for each predicate's correlation with the crash and their relative order of correlation strengths. Then, our approach runs RL on the follow-up fuzzing process, rewarding the selection of the seeds (that are the inputs mutated by the fuzzer to generate new inputs) and the mutations that could lead to significant changes to the current estimates (the correlation of each predicate and their order). By balancing the exploitation of the selection strategy with a high expected reward and exploration of a random strategy, this learning process guides RACING toward a ranked list of the predicates, with the root cause being top on the list.

Evaluation and findings. We implemented RACING and evaluated it on 30 vulnerabilities found in 21 popular applications or libraries, including 19 also used in the prior research (Aurora [12]). Our study shows that RACING significantly outperforms the state-of-the-art, speeding up the RCA process by 13.22 times on average, compared with Aurora: typically an analysis that needs hours to complete by the prior approach can be done within minutes, with more than 2/3 of the delay being reduced even in the worst case. This performance gain does not come at any quality cost. Comparing with Aurora, RACING produces a high-quality ranked list of predicates, with the root cause keeping its rank in Aurora's report in 20.00% of the cases, slipping slightly below 33.33% of the time, while going above in 46.67% of all the results. Our findings provide strong evidence that indeed RACING enhances both the scalability and the effectiveness of SFL.

Contributions. We outline our contributions below:

• *New understanding*. We analyzed how counterexamples impact the discovery of the rank for a given crash's root cause, which has never been done before. Our analysis shows that by sampling these counterexamples, an enhanced SFL approach can quickly identify the program entities most related to the crash. This enables significant acceleration of the RCA process through statistical SFL.

• *New techniques*. Based upon the new understanding, we developed a novel RL-enhanced SFL technique that automatically learns from the outcome of each fuzzing round to adjust the seed selection and mutation strategy. This new technique is shown to improve the performance of the state-of-the-art SFL approach by over an order of magnitude and also elevate its effectiveness in finding the root-cause vulnerability. We have published the code of our approach online [5].

Roadmap. The rest of the paper is organized as follows: Section 2 provides the background of our research; Section 3 presents our analysis on the impact of counterexamples; Section 4 describes the design and implementation of RACING; Section 5 reports our experimental results; Section 6 discusses the limitations and potential future research; Section 7 surveys related prior work and Section 8 concludes the paper.

2 Background

Statistical RCA framework. As mentioned earlier, SFL is a statistical RCA, which is known to be the state-of-the-art in triaging the crashes of a program and identifying underlying vulnerabilities [12, 39, 52]. This approach is based upon the belief that the program entity fundamentally responsible for crashes (the root cause) should be encountered when the program is running on the inputs causing the crash (*crash inputs*), rather than those leading to a normal exit (*non-crash inputs*). Therefore, it requires a testing set with both crash inputs and non-crash inputs, and estimates the statistical correlation between the program entity and the crash on both crash and non-crash inputs. A higher estimated correlation indicates that the program entity is more likely to be the root cause. A prominent example of the estimation is *mutual information*, which is represented by a matrix for measuring the contributions of different program entities to the crashes [52]. Specifically, the mutual information I between X (program entity) and Y (crash or non-crash inputs) can be calculated as follows:

$$\begin{aligned} &\mathbf{I}(X;Y) = H(Y) - H(Y|X) \\ &H(Y|X) = -\sum_{x\in\{0,1\}} Pr(x) \sum_{y\in\{0,1\}} Pr(y|x) \log_2 Pr(y|x) \end{aligned} \quad (1)$$

where $H(Y)$ is the marginal entropy and $H(Y|X)$ is the conditional entropy. Mutual information is also used in RACING, to measure the correlation of a program entity to the crash.

Predicate. A root cause is a program entity serving as a *necessary condition* for the crash, which depending on the granularity of RCA, could be a predicate, statement, or block. Following Aurora [12], our research focuses on the predicate-level root causes including three types: (1) register and memory predicates, (2) flag predicates, and (3) control-flow predicates. Specifically, predicates (1) and (2) are assertions on an instruction, predicting the occurrence of the crash when some conditions on the instruction's destination operand and the flag register are satisfied. For example, a predicate could be `eax<=z` for the destination operand `eax` of an instruction `mov eax, ebx`, indicating that an execution will crash *when* `eax<=z` *and exit normally otherwise*. And, `flag.ZF>0` is a flag predicate that predicts a crash *only when the zero flag is set to 1*. Predicate (3) evaluates the difference between executions on the control-flow graph (CFG), including the execution of certain edges and the number of successors for nodes. For example, the predicate $\texttt{count}_{\texttt{x->y}}\texttt{>0}$ is used to predict a crash *when the edge* `x->y` *has been executed for at least once*, while the predicate $\texttt{successor}_{\texttt{x}}\texttt{>2}$ predicts a crash *when the number of executed successor nodes for x is more than 2*. All three types of predicates are expressed in a union format `Var op Thr`, where `Var` represents the variable of interest, `op` denotes an operator (<= or >), and `Thr` denotes a threshold calculated during the fuzzing process. We follow Aurora's method to construct the predicates (see details at Appendix A.1). In particular, we use mutual information (Eq 1) to estimate the correlation of the predicates with the crash.

Note that a predicate might change during a fuzz test, when new observations have been made to affect the estimate of the condition that leads to the crash. However, at any fuzz round, a variable of interest is always associated with a predicate based upon the best estimate so far. Therefore, throughout the paper, we consider any execution of the instruction as a sample on its predicate, which both affects the estimate of the predicate's correlation with the crash and could lead to the adjustment of the predicate itself, and use the term "sample value" to refer to the value assigned to the variable of a predicate during an execution.

AFL in RCA. American Fuzzy Lop (AFL) [51] is a code-coverage-based brute-force fuzzer, which is among the most popular fuzzing tools. Given a set of seed inputs, AFL repeatedly selects and mutates seeds to generate new inputs to test a target program and detect its exceptions (e.g., crashes). Any mutation improving code coverage is added to a queue as a new seed candidate. During this fuzzing process, AFL tends to generate an enormous number of inputs, including both crash and non-crash inputs. Existing statistical RCA techniques run AFL for a predetermined period of time to produce a large number of fuzzing outcomes on a test set, which serve as samples for estimating the correlations between program elements and observed crashes [12, 52]. However, we found that this testing set generation process is optimized for code coverage, as AFL is supposed to do, rather than for efficient identification of the root causes. Also it is biased towards crashes, since all these RCA approaches enable the AFL's *crash exploration mode*, only selecting seeds from crash inputs (see Section 3). Therefore, in our research, we modified AFL's seed selection and mutation strategies to incorporate our counterexample-based RL, so as to make the fuzzing process more suitable for efficient discovery of program vulnerabilities (Section 4).

Reinforcement learning. In this paper, we will utilize the algorithm for a reinforcement learning problem, the Adversarial Multi Armed Bandits (AMAB) problem. This problem describes a game of T rounds between a player and an adversary. In each round $t = 1, ..., T$, the player selects a distribution d_t over the k-arms and the adversary selects a reward vector $g_t \in \mathbb{R}^k$. An action a_t is sampled from d_t and the player observes the reward $g_t(a_t)$. The player's expected regret is

$$G_T = \mathbb{E}[\max_{i\in[k]} \sum_{i=1}^{T} g_t(i) - \sum_{i=1}^{T} g_t(a_t)]. \quad (2)$$

We assume that the reward vector g_t depends only on the player's actions. So, the expectation above is calculated on the distribution of the player's actions. The goal of the player is to select the distributions $d_1, ..., d_T$ such that the regret G_T across the T rounds is minimized.

During RL, the player typically employs two important strategies, *exploitation* and *exploration*. Exploitation refers to the player using the knowledge it has already gained to make the best decision in a given state. Exploration refers to the

player trying new actions that may have not been experienced before (often via random sampling) to gather more information and potentially uncover better strategies. In RACING, for RL's exploration mode, we adopt a random strategy.

3 Counterexamples for Root Cause Analysis

At the center of a statistical RCA is the estimate of accurate predicate rankings according to these predicates' contributions to a given crash. For this purpose, existing approaches, such as Aurora [12], assess each predicate's correlation with the crash using a sampling strategy biased toward the crash, which as mentioned earlier, is less differentiating and therefore requires a lot of samples (outcomes of fuzz tests). We believe that by balancing the sampling around known crash inputs (as the prior approaches do) and the sampling around the test cases with the most differentiating power, the root cause for the crash can be discovered much more efficiently, on a small set of samples. In our research, we refer to such testing cases, which are characterized by their outcomes' disagreement with the current estimates, as *counterexamples*.

In our research, we consider two kinds of counterexamples: those correcting the current estimates of rankings (counterexamples for rankings, or CoRs) and those correcting the estimate of each predicate's correlation (counterexamples for predicates, CoPs). Our research shows that CoRs alone can help determine accurate rankings of predicates on a small number of inputs, while CoRs and CoPs together can further cut down the required sample size for accurately estimating the ranking of the root cause. Following we present our analysis on the efficacy of these two types of counterexamples.

3.1 Counterexample for Ranking

To understand the impacts of CoRs on learning of predicate rankings, we first formalize the ranking problem, whose solution depends upon effectively distinguishing predicates from each other based upon their contributions to a given crash. Then, we show that CoRs are exactly such differentiating inputs and a sampling strategy leveraging them can speed up the process to move the root-cause predicate up on the list, due to the small sample size it requires.

Problem modeling. We model RCA as a ranking problem, ranking predicates according to their associated statistics that measure how likely each of them causes a given crash, based upon its correlation with the crash. Formally, consider N predicates $\{p_1, p_2, ..., p_N\}$ to be ranked and their associated statistics $\{s_1, s_2, ..., s_N\}$. We use s_i^* to denote the true value of the statistic s_i for the predicate p_i and $\tilde{s}_i$ an estimate of s_i^*. An order of these N predicates is denoted by r, which is a permutation of the first N positive integers, i.e., $r \in \Pi([N])$. We denote the i-th element in r as r_i, and the ranking of predicate p_i as $r(i)$. Our purpose is to determine the true order (rankings) r^* of these N predicates according to the true values of their statistics (in a descending order, i.e., $s^*_{r_1^*} > ... > s^*_{r_N^*}$). In practice, however, we only know the estimates of these statistics $\{\tilde{s}_i\}$, which are calculated on n_i samples $e_i = \{e_{i1}, e_{i2}, ..., e_{in_i}\}$ for the predicate p_i. To highlight the importance of the sample size, we use $\tilde{s}_{i,n_i}$ to describe the estimate made on n_i samples. Note that each sample e_{ij} is produced by running the program under the RCA analysis on an input sample – a testing case and each execution of the program can generate multiple samples for different predicates. Since this fuzz process is time-consuming, our objective is to minimize the number of testing cases for determining the true order r^*.

Sampling strategies for ranking estimates. A direct way to estimate the order r is to generate unbiased random inputs so each predicate is adequately sampled, leading to the convergence of their statistics ($\tilde{s}_i \to s_i^*$), according to the Law of Large Numbers [36]; in this case, the estimated order r will converge to r^*. This simple solution, however, is less practical since it is hard to ensure that randomly generated inputs can reach the predicates, so a large number of the input samples need to be produced and tested, particularly for a complicated program, to provide each predicate with adequate coverage.

To address this problem, prior SFL approaches, such as Aurora [12], take advantage of a known input leading to the crash, called *Proof of Concept* (PoC), to guide the generation of other input samples. By mutating the PoC input and other crash inputs mutated from the PoC, these approaches create random samples around such an input, which are likely to go through a similar instruction set as the crash input, to sample the predicates on these instructions. This strategy enables the generation of the test inputs that more effectively cover the predicates than the unbiased random samples. However, these input samples are biased toward the crash and their related execution paths are less effective in distinguishing different predicates' contributions to the crash. Oftentimes, a subset of the predicates tends to appear *together* on the execution paths of the inputs leading to the crash, and are also less likely to show up on a non-crash path. As a result, these predicates become less distinguishable, rendering the root cause hard to stand out. Only after a large number of random trials around the crash inputs, would enough non-crash inputs be identified, which carry some of these predicates (so their contributions to the crash are sufficiently undermined), and to a lesser extent, new crash inputs discovered involving a small subset of the predicates (so their contributions to the crash become sufficiently prominent). These input samples could eventually move the root cause up the ladder, towards its true rank. We consider some of these inputs CoRs when they cause a significant change to the estimated order.

To speed up the convergence of the root-cause predicate's rank, we need to more efficiently differentiate different predicates' contributions to the crash. This purpose can be served by sampling around the discovered CoRs. Specifically, these inputs generated by mutating the CoR are likely to be differentiating, since a CoR is characterized by its capability to distin-

guish among predicates, mostly through causing a normal-exit result that is less observable under the PoC-guided approaches like Aurora. By incorporating this approach, our sampling strategy becomes more balanced between using the inputs that likely go through predicate-related instructions and leveraging those that differentiate the predicates' impacts on the crash, potentially leading to quick discovery of a root cause.

Counterexamples and convergence. As mentioned earlier, we consider a CoR to be an input sample that causes a big change to the estimated order. Specifically, a program's execution on a given input sample can hit multiple predicates to generate either a crashing or a normal-exit sample for each predicate, thereby inducing changes in ranks for some of them. To measure the impact of such input, we simply sum up the rank changes across all predicates encountered on its execution path in their absolute values, compared with the estimated order before the execution: that is, $\sum_{i=1}^{N} |r_t(i) - r_{t-1}(i)|$, where r_t is the estimated order after t fuzzing rounds. This measurement is found to work well in our experiments (Section 5), which is essentially twice of the Kendall Tau distance [19] (KTD) between two orders.

By balancing between the sampling around crash inputs and the sampling around CoRs, the estimate of the predicate order will converge and tends to converge quickly (Section 5). To avoid fuzzing the program even after the order estimate is stabilized, we can monitor the variation of the rank change measured between two consecutive fuzzing rounds, until the measurement approaches zero (below a threshold used in our research), indicating the convergence of the ranking list, according to the Cauchy's convergence theorem [13].

3.2 Counterexample for Predicate

As described earlier, the root-cause predicates can be more efficiently identified with the help of CoRs. This process can actually be further accelerated by seeking the counterexamples for estimating individual predicates' correlations with a given crash, as discovered in our research. These CoPs serve to amplify the distances between different predicates' estimated statistics through their oversized impact on each predicate's *error rate* (for predicting the crash). As a result, the number of samples required for ranking these predicates (based on their contributions to the crash) can also be reduced.

Distance amplification. For a predicate, CoPs are testing cases on which the execution disagrees with the predicate (e.g., the normal-exit execution, however, makes the condition associated with the predicate satisfied). Hence, if we increase the sampling rate of CoPs for a predicate, the error rate of the predicate should increase. We developed a method that increases the sampling rate of CoPs for each predicate (see Section 4.2) and, at the meantime, does not promote those non-root-cause predicates be ahead of root-cause predicates.

Formally, for the predicate p_i, $\tilde{s}_i$ could be seen as an estimate of how accurate this predicate can predict the exit status (crash or exit normally) of an execution of the program, and thus the error rate of this predicate is $1-\tilde{s}_i$. If the error rates of all predicates are amplified by a factor $\alpha > 1$, the distance between the estimated error rate of any two predicates increases accordingly: for two predicates p_1 and p_2, if $\tilde{s}_1$ and $\tilde{s}_2$ are both amplified by α, their distance increases from $(\tilde{s}_1 - \tilde{s}_2)$ to $\alpha(\tilde{s}_1 - \tilde{s}_2)$.

Sample size reduction. Consider a simple task where we want to rank two statistics according to their estimates. Intuitively, the larger the distance between true values of them, the fewer samples are required to rank them. Formally, we assume that there are two predicates p_1 and p_2 with the estimates $\tilde{s}_{1,n_1}$ and $\tilde{s}_{2,n_2}$ respectively. The true value of $\tilde{s}_{i,n_i}$ is s_i^* that represents how often the predicate p_i correctly predicts whether an execution triggers the given crash or not (i.e., the error rate of the predicate is $1-s_i^*$). Thus, among the n_i samples, the number of correct prediction based on the predicate p_i, referring as $c_{i,ct}$, follows a binomial distribution with the parameters n_i and s_i^*, a.k.a, $c_{i,ct} \sim \mathbf{B}(n_i, s_i^*)$, and an unbiased estimate of s_i^* on n_i samples is $\tilde{s}_i = \frac{c_{i,ct}}{n_i}$. According to the Wald method [7], we obtain that $\tilde{s}_i$ approximately follows a normal distribution with the mean of s_i^* and the variance of $\frac{s_i^*(1-s_i^*)}{n_i}$, a.k.a., $\tilde{s}_i \approx \mathbf{N}(s_i^*, \frac{s_i^*(1-s_i^*)}{n_i})$. If we further assume that the predicate p_1 and p_2 were sampled the same number of times, and their true values are ranked as $s_1^* > s_2^*$, then we can get $\tilde{s}_1 - \tilde{s}_2 \approx \mathbf{N}(s_1^* - s_2^*, \frac{1}{n_1}[s_1^*(1-s_1^*) + s_2^*(1-s_2^*)])$. Therefore, for a fixed distance between the two true values $(s_1^* - s_2^*)$, the greater number of samples (n_i) are used, the less likely we estimate $\tilde{s}_1 - \tilde{s}_2 < 0$. Equivalently, at a given confidence level, when the distance becomes larger, fewer number of samples are required to confidently rank them.

The same argument applies to the scenario of more than two predicates. If we increase the distance among true values of statistics (without changing the rank), the number of samples required to confidently rank the predicates decreases, and thus the number of testing cases required for producing these samples will also decrease. As shown in the beginning of this section, using more CoPs in fuzzing will effectively increase the distances of error rates between every pair of predicates. Therefore, in general, oversampling CoPs can accelerate the process of finding the correct rank r^*.

4 RACING: Design and Implementation

Our study (Section 3) shows that by leveraging counterexamples (CoRs and CoPs), statistical RCA can effectively determine an accurate ranking list just based on a small set of testing cases. This finding has been incorporated into RACING, which runs a reinforcement learning algorithm to guide a fuzzer to generate the testing cases highly likely to be counterexamples, which are more useful for identifying the root cause of a given crash than the others, thereby accelerating the RCA process. More specifically, RACING learns information

from prior fuzzing rounds, particularly the impacts of the test cases on the order of predicates and their statistics, to iteratively approximate an optimal seed-selection and mutation strategy, following the rewards for the strategies that make a big difference to the rankings and the statistics estimated so far (which are mostly counterexamples); these selected strategies are then applied to the next fuzzing round.

4.1 RL on Counterexamples

Our solution to the RCA problem is based upon a reinforcement learning technique designed to address the Adversarial Multi Armed Bandits (AMAB) problem. AMAB is a T-round game where the player intends to maximize her expected reward. In each round, the player selects an action to perform: in the context of RCA, such an action includes the selection of a seed or a method to mutate the seed to produce input samples. After running the target program on these inputs, we use the outcomes of the execution to determine the reward for the current round, and further estimate an expectation for the action based upon all the rewards it has received so far. In the rest of the section, we report our design of the reward mechanism and actions for selecting seeds and mutation methods.

Reward. The classical AMAB game (Eq 2) generates a reward for each round in the form of regret increment, for the purpose of minimizing the accumulated regret. For RACING, the reward is designed to guide the selection of the action that maximizes the differentiating power on predetermined predicates: that is, the seed selection or the mutation method that maximizes the difference among these predicates' estimated contributions to the crash. To this end, our RL technique rewards the action that maximizes the combined effect of CoR that changes the estimated order of the predicates, and CoP that causes the violation of the predicates on the instructions encountered, based upon our analysis in Section 3. Specifically, our reward G_t for the round t is defined as follows:

$$g_t = g_t^{order} + g_t^{count} \tag{3}$$

where g_t^{order} is the order change measured by the normalized KTD [28] between the estimated order before and after the round t, and g_t^{count} is the ratio of the predicates being violated during the program's executions in that round.

Action for seed selection. An action of our RL technique is selection of a seed for fuzzing the target program. The seed here is an input to the program, which is used to produce other input samples through mutating the input. To choose the input that stands the best chance to help maximize the reward for the next fuzzing round, we first organize all the input samples that have been evaluated so far into groups, according to the predicate-carrying instructions the execution on each input has encountered: each group corresponds to one such instruction, including all the inputs whose executions go through the instruction; every input tends to be affiliated with multiple such groups. Each group is associated with a *value* that is the average of all the rewards generated by its members. To select a seed, our approach first chooses the group with the highest value, and then picks from the group an input most likely to produce a CoP violating the predicate of the instruction associated with the group (see Section 4.2). This input is then used as the seed for generating other input samples.

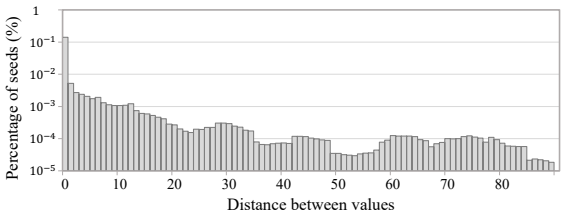


Figure 1: Distribution of distance.

Action for mutation selection. After selecting a seed, our approach takes further actions to choose a location (which byte on the input) and an operator for mutation, which are then applied to the seed to generate other input samples. Again, both the location and the operator are selected based upon their values – the average reward they have generated so far. For locations, we only gather rewards gained by mutating the top 2,000 bytes of an input and use RL to determine the probability of choosing them. For the bytes after the first 2,000, they might be selected only when the RL is in the exploration mode (random sampling). Here, 2,000 is selected experimentally (see details in Appendix A.2). For operators, RACING utilizes 16 different operators in 3 categories: insertion, modification, and deletion of some bytes on the seed. For instance, we may flip one bit on the seed, insert bytes with pre-defined tokens into the seed, etc. The full list of these operators can be found in Table 6 (see Appendix A.3).

4.2 Optimizing CoP Generation

Based on our analysis in Section 3.2, RCA could be accelerated by increasing the probability of generating CoPs that amplify the distance between the true values of the statistic for a root-cause predicate and that for a non-root-cause predicate. For a predicate `eax<=z`, CoPs are testing inputs on which non-crashing executions would assign `eax` values no more than z while crashing executions would assign `eax` values larger than z. Thus, to find CoPs, we need to generate the input sample whose execution causes the value of `eax` to fall within or beyond a boundary. To study how to generate such inputs with a high probability, we look into the relationship between a seed and the inputs generated from it.

The nearer the better. Comparing the value generated by the execution on a seed for a predicate and that produced on the input mutated from the seed, we found that they tend to be similar. Figure 1 shows the distribution of the distance

Table 1: Transferabilities between crash and non-crash inputs.

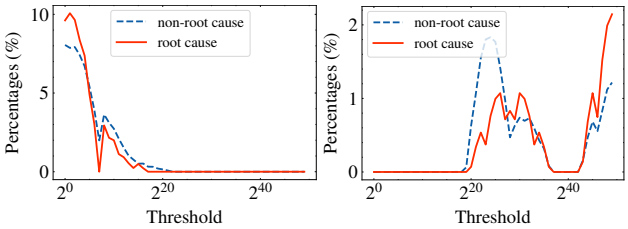
	Crash (to)	Non-Crash (to)
Crash (from)	54.73%	45.27%
Non-Crash (from)	0.01%	99.99%

between such sample values (defined in Section 2) of predicate, across all predicates within all programs (Table 2) we analyzed. As we can see, such a distance is very likely to be small: actually the probability becomes higher when the values become more similar. As a result, to generate the test cases more likely to produce the target value for a predicate, we should use the seed that leads to the predicate value as close to the target as possible. Based on the analysis, we can obtain the CoPs effectively from those "in-range" seeds: taking the predicate `eax<=z` as the example, those non-crash inputs but making `eax<=z` satisfied are more likely to generate CoPs of this predicate.

Transferability between crash and non-crash input. Besides the distance, we also look into the relationship between crash inputs and non-crash inputs. Table 1 illustrates the transferabilities between these two kinds, which are averaged among values gained from all programs we studied. From the table, we can see that, compared with crash inputs, the non-crash inputs, once used as seeds, are more likely to generate non-crash inputs with almost negligible probability to produce crash inputs. Besides from non-crash inputs, there are about half-to-half probabilities to produce crash and non-crash inputs respectively from crash inputs.

Optimizing seed selection. Considering both the distance and transferability, to generate CoPs for the predicate `eax<=z`, a good choice of seeds is the non-crash inputs making `eax<=z` satisfied. If no such non-crash inputs, a crash input making `eax<=z` satisfied would be also a good choice. In practice, to generate CoPs of a "`<=z`" predicate, we utilized the above strategy to set the seed as an available (not being chosen before) non-crash input, or a crash input if no such non-crash inputs, on which the execution will assign the smallest value to the operand of this predicate. Similar strategy is used to choose the seed for generating `>z` predicates' CoPs.

Root cause towards end. Our strategy increases the probability of generating CoPs for predicates. However, it is unclear whether our strategy would promote those non-root-cause predicates wrongly be ahead of root-cause predicates. To understand that, we investigate the relationship between thresholds of root-cause predicates and that of non-root-cause predicates. Respectively for `<=z` and `>z` predicates (see definitions in Section 2), we draw their thresholds' distributions on Figure 2a and Figure 2b. Figure 2a demonstrates that the thresholds of those `<=z` predicates crowd on small values, and the smaller the value, the greater the likelihood that it is the threshold of `<=z` predicates, regardless of root-cause or non-root-cause. Specially, the thresholds of root-cause predicates are more likely to be small values compared with that of non-root-cause predicates. Figure 2b demonstrates that there are two crowds (peaks) of threshold's values: one nears 2^{32} and another is bigger than 2^{50}, in the distribution of `>z` predicates' thresholds. We attribute the present of the first crowd (nears 2^{32}) to 32-bit operands used in a part of predicates, and similarly attribute the present of the second crowd ($> 2^{50}$) to 64-bit operands used in the rest predicates. In both two crowds, root-cause predicates exhibit a greater preference for using a large value as their threshold than non-root-cause predicates (where the red solid line is above the blue dash line). Based on the above results, we conclude that the thresholds of root-cause predicates are more likely extreme values (the smallest or the largest values) compared with that of non-root-cause predicates. Therefore, for `<=z` predicates, the thresholds of the root-cause predicates are, with high probability, smaller than that of non-root-cause predicates, and the probability of generating a CoP, on which executions would assign the operand with values smaller than the threshold, for the root-cause predicate is smaller than the probability of generating such CoPs for non-root-cause predicates, even though these probabilities all has been amplified by our strategy. And a consistent conclusion could be drawn for `>z` predicates.



(a) $<= z$ predicates. (b) $> z$ predicates.

Figure 2: Distribution of predicates' threshold.

4.3 RACING Algorithm

Algorithm 1 illustrates RACING, which takes a program *Prog* and a *PoC* leading to a crash as inputs to generate top-50 predicates mostly related to the crash and the order of their contributions r^*. Here we describe each RACING function:

Select_Action. This function returns a distribution d_t over all possible actions. Since there are two types of actions, selection of seed and selection of mutation, as mentioned earlier, this function outputs a pair of distributions $d_t = (d_{t,seed}, d_{t,mutate})$ over $N_a = (N_{a,seed}, N_{a,mutate})$ number of actions. Specifically, the distribution for seed selection $d_{t,seed}$ is built upon the average rewards for all $N_{a,seed}$ groups:

$$d_{t,seed}(i) = \frac{\exp(avg(i))}{\sum_{j=1}^{N_{a,seed}} \exp(avg(j))}, \tag{4}$$

where $d_{t,seed}(i)$ is the probability of choosing the group i, and $avg(j)$ is the average reward for the group j. This distribution

Algorithm 1: RACING algorithm

Input: Program *Prog*, and a Proof of Concept *PoC*.
Result: Top-50 predicates and their true order r^*.

```
begin
    S_0 = [], γ_0 = 0.5, t = 1
    do
        d_t ← Select_Action(S_{t-1})
        a_t ← Sample_Action(d_t, γ_t)
        {input} ← Generate_Input(a_t)
        {(e_i, p_i)} ← Run(Prog, {input})
        r_t ← Rank({(e_i, p_i)})
        g_t ← Compute_Reward({(e_i, p_i)}, S_{t-1})
        S_t ← Update_State(S_{t-1}, r_t, g_t, {(e_i, p_i)})
        γ_t ← Update_Gamma({g_i}, {d_i}, {a_i})
    while (! Converge(S_t))
    {p_i}_50, r* ← select top-50 p_i according to r_t
    return {p_i}_50, r*
end
```

is biased toward the groups with high average rewards. The distributions for mutation selection $d_{t,mutate}$ are constructed in a similar way, based upon the average rewards of the mutation locations and 16 mutation operations.

Sample_Action. This function randomly draws an action a_t from the distribution d_t, according to an exploration probability γ_t. The γ_t is used to balance exploitation and exploration, with a probability of 1-γ_t for exploitation and γ_t for exploration. Similar to d_t, $\gamma_t = (\gamma_{t,seed}, \gamma_{t,mutate})$. Specifically, for selecting a seed in each round, the probability of choosing the i-th input group as the seed is $d'_{t,seed}$(i), formally, it is:

$$d'_{t,seed}(i) = (1-\gamma_{t,seed})d_{t,seed}(i) + \frac{\gamma_{t,seed}}{N_{a,seed}}, \tag{5}$$

and the probability of choosing the mutation i is $d'_{t,mutate}(i)$, which is adjusted under the exploration probability $\gamma_{t,mutate}$ using the similar equation to Eq 5. Within the chosen input group (seed), we further identify the input sample most likely to produce a CoP, as mentioned earlier (Section 4.2). In this way, RACING identifies for the round t the joint action $a_t = (a_{t,seed}, \{a_{t,mutate}\})$, where $a_{t,seed}$ is the seed and $\{a_{t,mutate}\}$ is a set of the mutations sampled according to $d'_{t,mutate}$.

Generate_Input. This function returns a set of input samples $\{input\}$ by applying sampled mutations $\{a_{t,mutate}\}$ on the seed $a_{t,seed}$.

Run. This function executes the program on the selected test-case inputs and updates the statistics for each predicate encountered by the execution. Also based upon the statistics, our approach further adjusts the predicate as Aurora does [12] (also see Algorithm 2 in Appendix A.1). This update process has a time complexity $O(n_i)$ if there are n_i samples in total for the predicate p_i. In general, this function updates each predicate p_i and gathers all samples on p_i throughout all fuzzing rounds to form the set e_i.

Rank. This function returns the order r_t of all current estimates $\{\tilde{s}_i\}$. Each $\tilde{s}_i$ is updated concurrently with the predicate p_i based on the sample set e_i obtained in the round t.

Compute_Reward. This function generates the reward for this round according to Eq 3. Particularly, our approach utilizes the normalized KTD to calculate g_t^{order}, which is also normalized to a value in $[0,1]$ (simply dividing it by 2).

Update_State. This function updates the information for action selection, particularly the average reward $avg(i)$ for each seed group and the average award for each mutation, based upon the new award received.

Update_Gamma. This function adjusts $\gamma_t = (\gamma_{t,seed}, \gamma_{t,mutate})$ to balance exploration and exploitation efforts. Specifically, we update $\gamma_{t,seed}$ as follows ($\gamma_{t,mutate}$ is updated similarly):

$$\gamma_{t,seed} = \frac{N_{a,seed}}{2N_{a,seed} + \sum_{i=1}^{t} \Gamma_i(\gamma_{i-1,seed})}, \quad \text{where} \quad \Gamma_i(\gamma) = \frac{\gamma * g_i}{(1-\gamma) * d_{i,seed}(a_i) + \gamma / N_{a,seed}}. \tag{6}$$

Here Γ_i function computes the contribution of the current exploration strategy to the reward received in the round i. In our research, we utilized a monotonously decreasing function to let γ_t decline with the growth of the information gathered from prior rounds: the higher the rewards in the prior rounds can be attributed to exploitation, the faster the γ_t decreases.

Converge. This function checks whether the ranking list $[r_i]$ estimated in each round converges. Specifically, our approach calculates the KTD between the orders in two consecutive rounds to get a distance list $[KTD(r_1,r_2), ..., KTD(r_{t-1},r_t)]$. Then, we measure the variance of the last several $(\min(10,t))$ distances on the list. If this variance is smaller than a threshold (0.01), we conclude that the order list has converged and exit the loop. Here, 10 is a value found in our experiments that can improve analysis performance. Additionally, we only calculate the distance for the top-k ($k = 100$) predicates in the list. As Table 3 column T_{top100} shows, all root causes we studied have been elevated to the top-100, and the vast majority of them are moved up within a short period of time (in 3 minutes for 26 out 30 root causes).

4.4 Implementation

To implement RACING, we combine our reinforcement learning algorithm with AFL [51] algorithm that is customized accordingly, as well as cooperate with the code instrumentation techniques to record values (samples) produced for each instruction (predicate) during fuzzing the given program.

AFL customization. In RACING, the fuzzer is modified from AFL through replacing its strategies for seed selection and mutation with our own (Section 4.3).

Code instrumentation. Since the root cause of a vulnerability is inherently linked to the instructions executed by the Proof of Concept (PoC), to reduce the unnecessary cost, we only instrumented a code fragment following each instruction in the PoC's execution trace to record the assigned values and the control-flow transitions during the execution of test cases.

Table 2: Vulnerability information for evaluating RACING.

ID	Program	Version	Vulnerability ID	Vulnerability Type	Sanitizer	# lines of source code	# lines of PoC execution trace	Root Cause
V_1	readelf	binutils-2.32	CVE-2019-9077	heap buffer overflow	ASAN	62,754	2,625	readelf.c:16197
V_2	nm	binutils-2.28	Bugzilla-21670	stack buffer overflow	ASAN	10,882	2,781	tekhex.c:276
V_3	lua	lua-5.0	bug #5.0-2	heap buffer overflow	ASAN	11,260	5,767	ldo.c:325
V_4	nasm	nasm-2.14rc15	bugzilla-3392556	use after free	-	37,527	3,235	preproc.c:3821
V_5	tcpdump	tcpdump-4.9.2	CVE-2017-16808	heap buffer overflow	ASAN	77,256	2,056	print-aoe.c:328
V_6	sleuthkit	commit 3e2332a	issue-905	double free	-	215,149	25,272	ext2fs.c:807
V_7	patch	commit dce4683	bug #54558	heap buffer overflow	ASAN	509,743	2,255	pch.c:1332
V_8	bash	commit 6444760	msg00042	integer overflow	-	113,807	6,341	braces.c:421
V_9	nasm	nasm-2.14rc15	CVE-2018-16517	nullptr dereference	-	37,527	4,274	nasm.c:1477
V_{10}	libzip	libzip-1.2.0	CVE-2017-12858	use after free	-	14,184	1,727	zip_dirent.c:580
V_{11}	perl	commit dca9f61	issue-17384	heap buffer overflow	ASAN	980,019	46,755	regcomp.c:23690
V_{12}	objdump	commit 561bf3e	CVE-2017-9746	heap buffer overflow	ASAN	33,499	5,268	objdump.c:1932
V_{13}	mruby	commit 7483753	issue-3947	uninitialized variable	MSAN	117,108	15,649	pack.c:802
V_{14}	mruby	commit aa5c5de	CVE-2018-12248	heap buffer overflow	ASAN	117,248	19,093	fiber.c:208
V_{15}	mruby	commit 88604e3	hackerone-185041	type confusion	-	109,979	14,867	error.c:277
V_{16}	mruby	commit fabc460	CVE-2018-10199	use after free	ASAN	117,118	20,747	object.c:401
V_{17}	python	python-2.7.11	CVE-2016-5636	heap buffer overflow	-	443,641	49,689	zipimport.c:898
V_{18}	screen	screen-4.7.0	oss-sec-2020/q1/65	heap buffer overflow	-	42,466	3,840	ansi.c:1578
V_{19}	mruby	commit e9ddb59	CVE-2018-10191	integer overflow	ASAN	117,128	21,190	vm.c:1200
V_{20}	matio	commit bcf0447	CVE-2020-19497	integer overflow	-	21,032	692	mat5.c:4975
V_{21}	ezXML	ezxml-0.8.6	CVE-2021-30485	nullptr dereference	-	857	453	ezxml.c:362
V_{22}	libpng	libpng-1.6.34	CVE-2018-13785	divide-by-zero	-	56,923	1,714	pngrutil.c:3152
V_{23}	libjpeg	2.0.90 (2.1 beta1)	CVE-2021-20205	divide-by-zero	-	56,840	3,023	rdgif.c:447
V_{24}	libtiff	tiff-4.0.7	CVE-2017-7595	divide-by-zero	-	65,104	5,160	tif_jpeg.c:1628
V_{25}	libjpeg	commit f4b8a5c	CVE-2018-14498	heap buffer overflow	ASAN	48,682	4,812	rdbmp.c:209
V_{26}	libxml2	commit 362b322	CVE-2017-5969	nullptr dereference	-	231,069	10,237	valid.c:1181
V_{27}	libtiff	tiff-4.0.6	CVE-2016-5321	heap buffer overflow	ASAN	67,869	4,278	tiffcrop.c:992
V_{28}	libsixel	sixel-1.8.4	CVE-2021-45340	nullptr dereference	-	22,643	1,402	stb_image.h:6110
V_{29}	nm	binutils-2.32	CVE-2019-17451	integer overflow	-	11,340	8,241	dwarf2.c:4429
V_{30}	objdump	binutils-2.32	CVE-2019-9074	heap buffer overflow	ASAN	36,271	15,001	pei-x86_64.c:727

Existing binary instrumentation tools have limited capabilities, such as instrumenting limited lines of code or taking a considerable amount of time. Hence, we instrumented code using the following way: we execute the PoC with the help of IntelPIN [24] to obtain the binary code addresses of the PoC's execution trace at first; then, we use add2line [1] to locate source code corresponding to those addresses; finally, we create a custom LLVM Pass, an LLVM plugin, to insert a logging handler (triggering the recording of the assigned values to the target operand in an instruction as well as control-flow transitions) after each LLVM instruction in the Intermediate Representation of those located source code. In this way, the instrumented binary code has been included in the executable program compiled by the LLVM.

5 Evaluation

5.1 Setting

Environment. All experiments were conducted on a 64-bit Ubuntu 20.04 server with 32 Intel Xeon(R) Silver 4110@2.10 GHz CPU cores, 128 GB memory, and a 22 TB hard drive.

Vulnerability. We use 30 vulnerabilities to evaluate RACING, as illustrated in Table 2, including 19[1] from Aurora [12] (V_1-V_{19}) and 11 more collected ourselves in the real world (V_{20}-V_{30}). The 30 vulnerabilities were found in 21 programs varying in volumes, ranging from 857 to 980,019 lines of source code. These vulnerabilities cover 9 distinct types of common software vulnerabilities, including integer overflow, heap buffer overflow, stack buffer overflow, use after free, double free, divided-by-zero, nullptr dereference, type confusion, and uninitialized variable. Moreover, the level of complexity for each vulnerability is also diverse, which can be approximated by its number of distinct instructions of each PoC's execution trace, as illustrated in Table 2. The simplest vulnerability V_{21} in an XML parsing library, named ezXML, contains 453 distinct instructions in its PoC's execution trace, while the most complex one V_{17} of Python contains 49,689. Notably, when a program cannot be directly crashed by the PoC, we leverage the Address Sanitizer (ASAN) [38] or Memory Sanitizer (MSAN) [40] to compile the program and capture the exceptions to trigger a crash.

[1]The remaining 6 vulnerabilities in Aurora data set cannot be reproduced, so we only consider the 19.

Table 3: Evaluation result of RACING

Note: The time duration is given in the format `hours:minutes:seconds`.
blue: better result, red: worse result, green: equal result.

ID	RACING				Aurora						Speedup
	Ranking	T_{all}	$T_{ranking}$	T_{top100}	Ranking	T_{all}	$T_{ranking}$	$T_{fuzzing}$	$T_{analysis}$	Ranking*	
V_1	10	00:00:40	00:00:14	00:00:12	6	02:06:52	00:08:00	2:00:00	0:06:52	14	190.30×
V_2	1	00:01:36	00:00:51	00:00:01	1	12:00:50	00:15:00	12:00:00	0:00:50	8	450.52×
V_3	28	00:04:38	00:01:41	00:00:03	21	12:14:49	08:00:00	12:00:00	0:14:49	84	158.59×
V_4	8	00:03:06	00:02:44	00:00:01	14	02:24:13	00:20:00	2:00:00	0:24:13	37	46.52×
V_5	4	00:02:12	00:00:21	00:00:10	4	02:02:04	00:09:00	2:00:00	0:02:04	9	55.48×
V_6	9	00:01:57	00:01:24	00:00:18	10	02:02:05	00:06:00	2:00:00	0:02:05	24	62.61×
V_7	1	00:02:38	00:00:03	00:00:01	1	02:04:17	00:00:20	2:00:00	0:04:17	1	47.20×
V_8	20	00:03:25	00:03:19	00:00:02	29	03:10:46	02:00:00	2:00:00	1:10:46	1720	55.83×
V_9	20	00:12:07	00:02:44	00:00:03	58	02:39:57	01:20:00	2:00:00	0:39:57	45	13.20×
V_{10}	2	00:14:33	00:10:15	00:10:15	4	12:01:13	06:00:00	12:00:00	0:01:13	6	49.57×
V_{11}	16	00:16:46	00:13:01	00:00:30	9	06:15:05	01:30:00	2:00:00	4:15:05	138	22.37×
V_{12}	4	00:23:17	00:05:04	00:02:26	6	02:12:15	02:00:00	2:00:00	0:12:15	84	5.68×
V_{13}	3	00:19:22	00:01:45	00:00:27	1	12:57:01	00:09:00	12:00:00	0:57:01	1	40.12×
V_{14}	10	00:24:12	00:02:05	00:01:12	1	12:42:09	00:03:00	12:00:00	0:42:09	1	31.49×
V_{15}	39	00:46:52	00:46:52	00:00:13	47	14:00:14	08:00:00	12:00:00	2:00:14	74	17.93×
V_{16}	3	01:05:21	00:01:33	00:00:36	3	15:03:16	04:00:00	12:00:00	3:03:16	7	13.82×
V_{17}	33	01:32:38	00:34:00	00:08:10	28	18:08:53	10:00:00	12:00:00	6:08:53	510	11.75×
V_{18}	9	01:55:19	00:27:50	00:01:44	41	02:12:10	01:10:00	2:00:00	0:12:10	41	1.15×
V_{19}	18	04:29:23	03:46:20	03:46:20	9	14:05:54	10:00:00	12:00:00	2:05:54	16	3.14×
V_{20}	3	00:00:15	00:00:07	00:00:00	1	02:00:04	00:30:00	2:00:00	0:00:04	18	480.27×
V_{21}	3	00:00:12	00:00:01	00:00:00	3	02:00:29	00:20:00	2:00:00	0:00:29	31	602.42×
V_{22}	1	00:00:40	00:00:30	00:00:11	9	02:00:43	00:20:00	2:00:00	0:00:43	4	181.08×
V_{23}	9	00:01:56	00:00:07	00:00:00	14	02:11:49	00:10:00	2:00:00	0:11:49	19	68.18×
V_{24}	3	00:02:20	00:00:06	00:00:03	10	02:07:27	00:04:00	2:00:00	0:07:27	7	54.62×
V_{25}	1	00:02:32	00:00:12	00:00:12	1	02:01:43	00:00:20	2:00:00	0:01:43	1	48.05×
V_{26}	6	00:05:04	00:03:16	00:00:03	11	02:07:15	01:00:00	2:00:00	0:07:15	15	25.12×
V_{27}	14	00:05:37	00:01:41	00:00:07	7	02:07:06	00:10:00	2:00:00	0:07:06	18	22.63×
V_{28}	8	00:09:01	00:01:31	00:01:17	2	12:05:43	07:00:00	12:00:00	0:05:43	12	80.49×
V_{29}	7	00:25:58	00:14:57	00:00:40	47	02:03:17	00:20:00	2:00:00	0:03:17	47	4.75×
V_{30}	4	00:53:21	00:15:08	00:01:51	13	02:01:48	00:04:00	2:00:00	0:01:48	13	2.28×
Avg.	9.90	00:27:34	00:13:59	00:08:34	13.70	06:04:26	02:10:17	5:20:00	0:44:26	100.17	13.22×

Ground truth. In order to obtain the ground truth, we found it from the vulnerability's patch. Specifically, we manually studied the 30 vulnerabilities' patches. The vast majority of patches (76.67%) consist of no more than ten lines of modified code, and even 11 patches are within three lines of modified code, making it easy to determine the root cause. For the remaining complex patches, which may be related to multiple files and contain a large number of modified lines, we referred to the vulnerability report, if any, and manually inspected the vulnerability with a debugger to determine a reasonable ground truth. Table 2 illustrates the file name and the number of line in which the root cause is located. Additionally, we provide an example in the appendix to illustrate how the ground truth is determined (see details at Appendix A.5).

5.2 Results

RACING is designed based on the classic statistics-based RCA framework, but uses counterexamples for RCA acceleration. In the evaluation, we answer the following research questions:

- **RQ1:** How does the effectiveness of RACING compare to current works for locating root causes?
- **RQ2:** How much does RACING improve efficiency?

To answer these questions, we compared our result to Aurora [12], the state-of-the-art work for identifying predicate-level root causes. Table 3 presents the evaluation result. As illustrated in Table 3, all 30 root causes were identified within top-50 by RACING, including 73.33% (22) in the top 10, 33.33% (10) in the top 3, and 4 even at the top spot. Comparatively to Aurora, we have 6 root causes of vulnerabilities (colored in green in the *Ranking* column of Table 3) receiving the same rankings as Aurora, yet, 14 (46.67%) root causes are ranked even higher by RACING than Aurora (colored in blue in Table 3). For instance, V_9 was identified by RACING with a ranking of 20, whereas Aurora gave it a ranking of 58, which is not only significantly lower but also outside the top 50, meaning an expert has to put in much more effort to check the actual root cause. On average, the rankings obtained by RACING for these vulnerabilities' root causes are 3.80 positions higher than those obtained by Aurora. This result is reasonable. Even though Aurora spends a large amount of time (e.g., 12 hours) to generate testing set, nobody knows whether the randomly generated data set is statistically adequate. Our approach, RACING, not only attempts to generate counterexamples to aid in the accurate ranking of predicates, but also uses a termination strategy to determine whether the testing set is sufficient to identify the root cause. Therefore, we have better rankings than Aurora for some vulnerabilities' root causes. Of course, due to the random nature of fuzzing in practice, 10 root causes are ranked lower than Aurora by

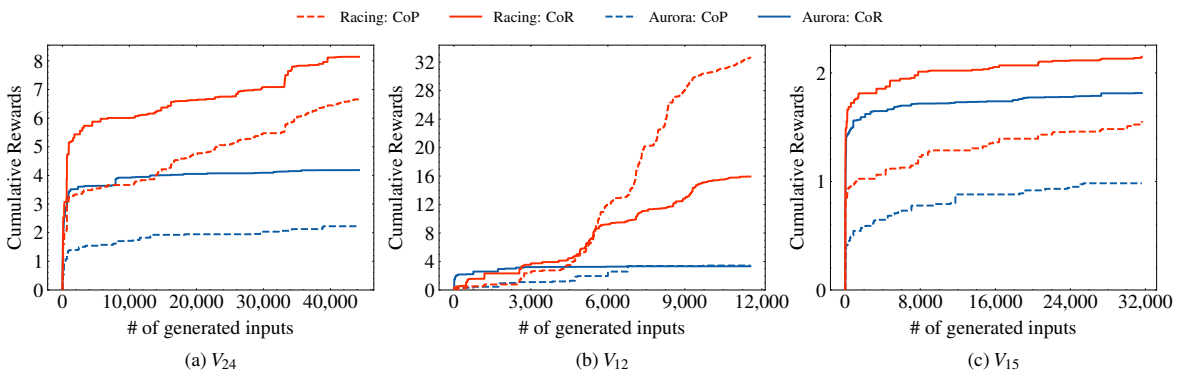


Figure 3: Comparison between RACING and Aurora on CoP and CoR generation.

RACING (colored in red in Table 3). However, the maximum ranking difference for the same root cause is 9 positions, with an average difference of 5.80 positions. Considering, except V_3 with a ranking of 28 and V_{17} with a ranking of 33, all of them are in the top 20, we deem these lower rankings acceptable. Overall, RACING possesses the same efficacy as existing works for locating root causes, which concludes **RQ1**.

To evaluate the efficiency improvement of RACING, we first compare the total time cost (T_{all}) by RACING and Aurora to perform RCA. Notably, Aurora includes two separate steps to analyze root causes. It uses AFL to generate the testing set with $T_{fuzzing}$ time and then $T_{analysis}$ time to determine each predicate's ranking. Particularly, when Aurora generates less than 100 test cases in 2 hours, it will take an additional 10 hours to collect a larger testing set. Unlike Aurora, in RACING, the test case generation and the ranking computation are performed successively in each round (see Algorithm 1). As shown in the *Speedup* column in Table 3, RACING identified each root cause significantly faster than Aurora, with a minimum increase of 1.15x for V_{18} and a maximum increase of 602.42x for V_{21}. Specifically, RACING identified 86.67% (26) root causes in 1 hour, 80.00% (24) in 30 minutes, 56.67% (17) in 10 minutes, and 13.33% (4) even in 1 minute (e.g., 15 seconds for V_{20} and 12 seconds for V_{21}). On average, RACING required approximately 27 minutes to identify the root cause, whereas Aurora required approximately 6 hours, a 13.22x reduction. Figure 4 illustrates the detailed distribution of RACING over Aurora based on speed multiples. One may question whether Aurora wastes time in the later period generating test cases. To figure it out, we examined the earliest time the root cause received the same ranking as its final ranking (denoted by $T_{ranking}$) during the test case generation in Aurora. The results indicate that Aurora did waste a great deal of time generating more test cases even though the root causes could be identified with a relatively high ranking. For instance, V_2 ranked first in the data set generated in 15 minutes by Aurora, but Aurora used 12 hours to collect more test cases to analyze it. Nevertheless, RACING performs better. It takes only 51 seconds to rank the root cause of V_2 number one. On average, the root causes obtained the same ranking as their final rankings in about 2 hours in the Aurora-generated testing set, whereas RACING could allow the root causes to receive the same ranking as their final rankings in around 14 minutes. This comparative study demonstrates that RACING can expedite the process of predicates obtaining their final rankings. Moreover, we checked the rankings of root causes in Aurora at the end time of RACING (see *Ranking** in Table 3). 83.33% (25) root causes are ranked lower than those computed by RACING, which once again demonstrates the efficiency of RACING. Therefore, we conclude **RQ2** that RACING has significantly improved efficiency compared with existing works on root cause analysis.

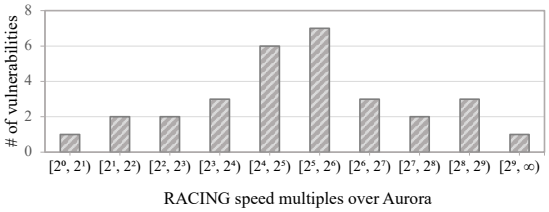


Figure 4: RACING speed multiples' distribution over Aurora.

In addition, to evaluate whether RACING actually generates counterexamples, i.e., whether a counterexample affects more predicates and whether an input results in a larger change in order, we compare RACING to Aurora using three selected vulnerabilities whose root cause was identified within 10 minutes (V_{24} in 2 minutes and 20 seconds), 30 minutes (V_{12} in 23 minutes and 17 seconds), and more than 30 minutes (V_{15} in 46 minutes and 52 seconds). Specifically, for each generated input, we counted what is the percentage of predicates this input is the counterexample (CoP) for and how much KTD (see Section 4) this input causes, which indicates the change in the order (CoR). As shown in Figure 3, the lines for CoPs and CoRs in the Aurora testing set rise very slowly, except at the beginning, demonstrating that fewer CoPs and CoRs have been generated by Aurora due to their blind test case generation method as we analyzed in Section 3. How-

Table 4: Result of ablation studies.

Note: The time duration is given in the format `hours:minutes:seconds`

	V_{24}		V_{12}		V_{15}	
	Time	**Ranking**	**Time**	**Ranking**	**Time**	**Ranking**
RACING (reward: CoR + CoP)	00:02:20	3	00:23:17	4	00:46:52	39
RACING (reward: CoR)	00:14:14	3	00:33:43	4	01:51:03	39
Aurora	02:07:27	10	02:12:15	6	14:00:14	47
RL + top 100	00:09:02	3	00:40:35	4	01:10:21	39
RL + optimizing + top 100	00:02:20	3	00:23:17	4	00:46:52	39
RL + optimizing + top 300	00:45:47	3	01:07:48	4	01:23:25	39
RL + optimizing + top 500	01:57:15	4	02:07:39	5	04:57:22	39

ever, the lines for CoPs and CoRs in the testing set generated by RACING rise rapidly and continuously, indicating that the inputs are successfully generated by RACING, resulting in more predicates affected and greater order changes than Aurora. Especially, for V_{15}, RACING generates significantly more CoPs and CoRs from the beginning to the end. Combining the *Ranking** of V_{24}, V_{12}, and V_{15} in Table 3, which are all lower than the rankings computed by RACING at the end of RACING's analysis time, this result again demonstrates the effectiveness of counterexamples (CoPs and CoRs) for estimating the accurate rankings of predicates.

5.3 Ablation Study

As stated in Section 4.1, the reward is comprised of the combined effect of CoRs and CoPs. To evaluate their utilities in the reward, we conducted an ablation study by removing the CoP from the reward and setting $g_t = g_t^{order}$ in RL, based on three selected vulnerabilities in the evaluation (V_{24}, V_{12}, V_{15}). As shown in Table 4, the result demonstrates that CoRs alone can assist in identifying the root cause with a higher ranking in a shorter time, increasing 8.95x for V_{24}, 3.92x for V_{12}, and 7.57x for V_{15} in comparison to Aurora. Furthermore, CoRs and CoPs together can further cut down the analysis time.

Besides using reinforcement learning to guide the fuzzer, RACING also includes an optimizing method to generate CoPs (Section 4.2), and always focuses on top 100 predicates to avoid wasting time on those ranked below 100. Therefore, we conducted additional ablation studies to evaluate their utilities in RACING, again based on V_{24}, V_{12}, V_{15}. The result is presented in Table 4. The 5th row labeled "RL+optimizing+top 100" indicates the result of RACING.

To evaluate the optimizing method, we replace it with a random method selecting an input on which execution goes through a given instruction at random. The result demonstrates that the optimizing method enables the order to be converged more rapidly to locate the root cause. Specifically, the optimizing method is around 17 and 23 minutes faster than the random selection when analyzing V_{12} and V_{15}, respectively. And for analyzing V_{24}, the optimizing method takes only 2 minutes and 20 seconds, whereas the random selection takes 9 minutes and 2 seconds, which is nearly 4 times longer. Moreover, to figure out whether the optimizing method generates the CoPs as anticipated, with a higher probability of generating counterexamples for a given predicate than random selection, we compared the average percentage of successfully generated CoPs for both methods. We found that with the help of the optimizing method, we generated CoPs with an average probability of 0.35% (=158/44,266) on V_{24}, 3.99% (=461/11,554) on V_{12}, and 0.18% (=58/31,614) on V_{15} while generating inputs, whereas using the random selection, the probability reduces to 0.09% (=227/253,062) on V_{24}, 1.37% (=388/28,364) on V_{12}, and 0.09% (=84/89,837) on V_{15}. This result indicates that our optimizing method is more likely to produce counterexamples for given predicates.

To evaluate how well the k is configured, we set two more values 300 and 500 to k. As illustrated in Table 4, as the value of k increases, the ranking of the root cause almost does not change, but the termination time increases significantly. Taking V_{24} as an example, RACING ($k = 100$) has spent 2 minutes and 20 seconds locating the root cause, whereas the time cost multiplies 20fold (45 minutes and 47 seconds) when $k = 300$ and even 50fold (1 hour and 57 minutes) when $k = 500$. This result is reasonable, because a larger k indicates that more time is needed to generate counterexamples for more predicates. Also, this result demonstrates that setting $k = 100$ is sufficient, since RACING can locate the root cause with the same ranking as when k is set with a larger value.

6 Discussion

Limitation. Below, we illustrate the limitations of RACING that managed to rank the root cause of vulnerabilities, as demonstrated in our experiments, at the very top in a short time. There are still some root causes failed to be ranked at the first by our RACING due to two reasons: 1) RACING failed to generate counterexamples that can lower the ranks of those predicates falsely ranked at the top. 2) some executions (coincidentally correctness) making the root-cause predicate satisfied however exit normally. These two reasons reveal the limitations of RACING in discovering a comprehensive set of counterexamples and dealing with coincidentally correctness.

Also, our current implementation of RACING relies on the program's source code for code instrumentation, limiting the application of our tool on those programs without the source code. However, this implementation (our code) is readily expandable if a more comprehensive and potent binary code instrumentation tool becomes available in the future.

Additionally, predicates could be classified into atomic predicates (e.g., `x<=10`) and compound predicates (e.g., `x<10 AND y>=8`). Currently, RACING only takes the first step to deal with atomic predicates. It lacks the support for compound predicates, which account for *less than half* of the vulnerabilities[2]. However, the idea of RACING can be straightforwardly extended to handle compound predicates because they still have counterexamples like `x>=10 OR y<8` for `x<10 AND y>=8`, and RL can be used to generate such counterexamples. Specifically, the design of RACING's reward mechanism (Eq 3) as well as the actions for seed and mutation selection (Eqs 4,5,6) can be directly used, with non-trivial adjustments to the construction of (compound) predicates and customization on the optimization strategy for CoP generation. To estimate the likely cost of RACING to support compound predicates, we developed a prototype called RACING$^+$ to support a simple type of compound predicate by combining two atomic predicates using the logical operator "AND" (`A AND B`) and tested it on two vulnerabilities (see details at Appendix A.4). In order to understand RACING$^+$'s ability, we additionally modified the computational component of Aurora (called Aurora$^+$) to handle the same type of compound predicate. As shown in Table 7 in Appendix A.4, RACING$^+$ accelerates over Aurora$^+$ by an average of 6.58x, while still identifying the compound root causes within its top-50, surpassing Aurora$^+$ in terms of ranking. These preliminary results indicate the potential of RACING for effectively supporting compound predicates in future endeavors.

Future work. We attribute the sluggishness of current RCA techniques to their employed biased sampling method raised from the crash centered fuzzing. Our research steps forward to overcoming this bias through incorporating with counterexamples. Along this path, further efficient RCA methods working with balanced sampling methods are anticipated in the future. Especially, working with an unbiased fuzzer is promisingly effective for RCA.

7 Related Work

Spectrum-based fault localization. Spectrum-based fault localization (SFL) technique is a statistically-based automated cause analysis method. Based on a testing set containing both crash and non-crash inputs, the SFL technique estimates the statistical correlation between each program entity (such as a statement, block, or predicate) and crash, and outputs the highly correlated entities in rank order as potential root causes. The majority of SFL research focused on designing statistical quantities, including ranking metrics [6, 15, 25, 34, 37, 43] and distribution statistics [8, 10, 31, 32]. Different from them, our work is not focused on the improvement of statistics metrics. We are concentrating on the generation of the testing set. The metric we used in RACING is mutual information, which is proposed by DeFault [52], one of the most recent SFL studies. There are some SFLs that have provided strategies to improve the testing suite [6, 11, 22, 49]. To evaluate a test case, Baudry *et al.* proposed a testing criterion called Dynamic Basic Block [11]. Hao *et al.* proposed three strategies for reducing the number of test cases based on their capability [22]. Abreu *et al.* contend, based on an experimental study, that a greater number of crash inputs results in greater effectiveness [6]. Yu *et al.* eliminates the test cases that have no impact on the ranking performance [49]. All of them, however, improve the testing suite on an existing data set. In contrast to them, we generate test cases dynamically based on our strategies to construct the testing set, and our evaluation demonstrates the efficacy and efficiency of the test generation method.

Non-statistical RCA. Non-statistical RCA strives to identify the root cause of certain vulnerability types through detailed and precisely formulated rules. Typical non-statistical RCA solutions [14, 23, 46, 47] employ program analysis techniques (e.g., data flow analysis, symbolic execution) to verify the program entities against manually crafted rules and uncover potential root causes. However, such analysis rules are often specific to certain vulnerability types, limiting their broader use. Among these solutions, FreeWill [23] can diagnose only 1 vulnerability type (use after free), while Bunkerbuster [46] supports the most with 6 vulnerability types, but it still cannot handle types such as uninitialized variables, type confusion and nullptr dereference in our dataset. Moreover, most non-statistical RCA faces inherent limitations associated with program analysis techniques, such as the high computational overhead of data flow analysis and program slicing [14], as well as scalability issues in symbolic execution [46,47]. Given that Bunkerbuster [46] supports the most vulnerability types, we ran it on 21 vulnerabilities it covers in our dataset. We found that 20 out of 21 aborted early due to problems like memory exhaustion, with only 1 successful case taking nearly 13 hours, which is almost 300 times longer than the execution time of RACING or 6 times longer than that of Aurora. Overall, compared to non-statistical RCA methods, statistical RCA methods have broader applicability and better efficiency.

Fuzzing. Fuzzing techniques have led to a tsunami of security vulnerability discovery and elimination through automatically and quickly funding abundant crashes. In the early days of fuzzing, random selection and mutation of seed were used to generate new inputs for testing programs [20, 35]. In order

[2]Since no previous studies have investigated this specific proportion, we randomly selected 20 vulnerabilities from Defects4J [26], a real-world fault dataset widely used for benchmarking fault localization, and found that 45% of the root causes were compound predicates.

to conduct more exhaustive testing, modern fuzzing typically employs heuristic strategies to guide exploration [29, 33, 51]. The coverage-guided fuzzing is one of the most popular techniques, such as AFL [51], which increases the likelihood of discovering more vulnerabilities by increasing code coverage. Current statistics-based RCA works, such as Aurora [12], DeFault [52], directly employed AFL to generate a testing set. However, as we analyzed in Section 3, such a generation process is time-consuming. Our approach, RACING, modified AFL's seed selection and mutation strategies to make the fuzzing process more suitable for efficient root cause analysis. In addition, recently, many researchers use reinforcement learning to enhance the fuzzing process. For example, Woo et al. [44] use the number of crashes as a reward function to find the greatest number of unique bugs in a fuzzing campaign. Yue et al. [50] proposed EcoFuzz, which uses the adversarial MAB algorithm to solve the seed energy allocation problem. AFL-HIER [41] chooses UCB1 [9], one of the MAB algorithms, to solve the seed scheduling problem under fine-grained coverage metrics. These works have demonstrated the effectiveness of reinforcement learning applied in fuzzing. However, the purpose of these works is to find more vulnerabilities, whereas the reinforcement learning in RACING is to learn how to generate more CoPs and CoRs for efficient root cause analysis.

8 Conclusion

In this paper, we introduced our reinforcement learning (RL) based approach, RACING, to elevate the scalability and accuracy of today's statistical RCA. Compared to previous works that generate the input samples more likely causing the crash, RACING harnesses our new observation that by sampling around "counterexamples" causing significant changes to the current estimates of correlations, suspicious program elements can be more effectively differentiated, thereby accelerating the RCA performance. Our evaluation on 30 real-world vulnerabilities demonstrates the efficiency and effectiveness of RACING: it successfully elevated all 30 root causes to the top (within the top-50) within half an hour in average, which is significantly faster than the state-of-the-art method of SFL that could take more than twelve hours for RCA, thereby speeding up the RCA process by more than an order of magnitude and elevating its effectiveness in locating the root-cause of vulnerabilities.

9 Acknowledgments

We thank the shepherd and all the anonymous reviewers for their constructive feedback. The IIE authors are supported in part by NSFC (92270204), Youth Innovation Promotion Association CAS.

References

[1] addr2line(1) - Linux manual page. https://man7.org/linux/man-pages/man1/addr2line.1.html.

[2] CVE-2017-5380. https://nvd.nist.gov/vuln/detail/CVE-2017-5380.

[3] CVE-2018-4145. https://nvd.nist.gov/vuln/detail/CVE-2018-4145.

[4] CVE-2022-36320. https://nvd.nist.gov/vuln/detail/CVE-2022-36320.

[5] RACING's source code. https://github.com/RacingN4th/Racing.git, 2023.

[6] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[7] Alan Agresti and Brent A Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[8] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 5–15, 2007.

[9] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[10] George K Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 73–84, 2010.

[11] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, pages 82–91, 2006.

[12] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. Aurora: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 235–252, 2020.

[13] Arlen Brown and Carl Pearcy. *An introduction to analysis*, volume 154. Springer Science & Business Media, 2012.

[14] Yue Chen, Mustakimur Khandaker, and Zhi Wang. Pinpointing vulnerabilities. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 334–345, 2017.

[15] Yu-Min Chung, Chin-Yu Huang, and Yu-Chi Huang. A study of modified testing-based fault localization method. In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 168–175. IEEE, 2008.

[16] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. {REPT}: Reverse debugging of failures in deployed software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 17–32, 2018.

[17] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, pages 820–831, 2016.

[18] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[19] Ronald Fagin, Ravi Kumar, and Dakshinamurthi Sivakumar. Comparing top k lists. *SIAM Journal on discrete mathematics*, 17(1):134–160, 2003.

[20] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, page 6. USENIX Association, 2000.

[21] Hackerone. Type confusion in mrb_exc_set leading to memory corruption. https://hackerone.com/reports/185041.

[22] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. Test input reduction for result inspection to facilitate fault localization. *Automated software engineering*, 17:5–31, 2010.

[23] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. {FreeWill}: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2497–2512, 2022.

[24] Intel. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[25] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[26] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[28] Ravi Kumar and Sergei Vassilvitskii. Generalized distances between rankings. In *Proceedings of the 19th international conference on World wide web*, pages 571–580, 2010.

[29] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[31] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *Acm Sigplan Notices*, 40(6):15–26, 2005.

[32] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[33] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[34] Wes Masri. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, 20(2):121–147, 2010.

[35] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[36] Pál Révész. *The laws of large numbers*, volume 4. Academic Press, 2014.

[37] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *2009 IEEE 31st International Conference on Software Engineering*, pages 56–66. IEEE, 2009.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. 2012.

[39] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 537–549, 2021.

[40] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[41] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. 2021.

[42] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[43] Xinping Wang, Qing Gu, Xin Zhang, Xiang Chen, and Daoxu Chen. Fault localization based on multi-level similarity of execution traces. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 399–405. IEEE, 2009.

[44] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[45] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security Symposium*, pages 17–32, 2017.

[46] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 320–336, 2021.

[47] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. {ARCUS}: Symbolic root cause analysis of exploits in production systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1989–2006, 2021.

[48] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[49] Yanbing Yu, James A Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th international conference on Software engineering*, pages 201–210, 2008.

[50] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2307–2324, 2020.

[51] Michał Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2017.

[52] Xing Zhang, Jiongyi Chen, Chao Feng, Ruilin Li, Wenrui Diao, Kehuan Zhang, Jing Lei, and Chaojing Tang. Default: mutual information-based crash triage for massive crashes. In *Proceedings of the 44th International Conference on Software Engineering*, pages 635–646, 2022.

A APPENDIX

A.1 Predicate Construction

Following Aurora [12], our research focuses on three types of predicates. Below we elaborate on the details of their union form transformation and threshold selection.

Union form transformation. All three predicate types follow a union form `Var op Thr`. For register and memory predicates, `Var` is the destination operand of an instruction, `op` could be <= or > and `Thr` is an integer chosen dynamically (see below). For flag predicates, `Var` is the flag bit of interest, `op` is > and `Thr` is 0. For control-flow predicates, there are two primary types: the execution of an edge and the number of executed successors for nodes. For each edge `x->y` connecting the node `x` and `y` in the CFG, an additional variable $\texttt{count}_{\texttt{x->y}}$ is maintained to track the number of times the edge has been executed during one execution of the program. In this scenario, `Var` is the variable $\texttt{count}_{\texttt{x->y}}$, `op` is > and `Thr` is 0. Similarly, for each node `x`, an additional variable $\texttt{successor}_{\texttt{x}}$ is created to store the number of its successors that have been executed during one execution of the program. In this scenario, `Var` is the variable $\texttt{successor}_{\texttt{x}}$, `op` is > and `Thr` is either 0, 1 or 2.

Threshold selection algorithm. Both flag predicates and control-flow predicates have a limited number of predefined thresholds. Conversely, the threshold of register and memory predicate can encompass tens of thousands of possible values. Evaluating each potential value is computationally impractical in real-world scenarios. To address this challenge, Algorithm 2 is employed to dynamically update the threshold, ensuring it is linked to the best estimate of the mutual information so far.

Algorithm 2: Threshold selection

Input: m_i sample values $\{v_1, v_2, \ldots, v_{m_i}\}$ in ascending order for a predicate p_i on n_i inputs $\{e_{i1}, e_{i2} \ldots, e_{in_i}\}$.
Result: The updated predicate p_i' and its statistic $\tilde{s}_i$.

```
1  begin
2      s̃_i = 0, C_t = 0, N_f = 0
3      for j ← 1 to m_i do
4          C_t = C_t + CrashSamples(v_j)
5          N_f = N_f + NonCrashSamples(v_j)
6          s̃_i' = Compute_MutInf(C_t, N_f, C_a, N_a)
7          if s̃_i' > s̃_i then
8              s̃_i = s̃_i'
9              if C_t > C_a − C_t then
10                 p_i' ← Var<=v_j
11             else
12                 p_i' ← Var>v_j
13     return (p_i', s̃_i)
```

Concretely, given m_i sample values $\{v_1, v_2, \ldots, v_{m_i}\}$ for a predicate, it tries to use each v_j in ascending order (line 3-12) as the threshold, computes its mutual information with the crash (line 6), and updates the predicate's threshold `Thr` if a new value v_j yields higher statistic than previous ones (line 7-12). To evaluate the predicate's mutual information on the fly, we maintain two counters, C_t and N_f, which represent the number of crash inputs that satisfy the current predicate at v_j, and the number of non-crash inputs that violate the predicate at v_j, respectively. In each step, we increment C_t with the number of crash samples for v_j, and N_f with the number of non-crash samples for v_j (lines 4-5). With C_t, N_f, C_a (the total number of crash inputs), and N_a (the total number of non-crash inputs), we can obtain the number of crash inputs that violate the predicate via $C_f = C_a - C_t$, and the number of non-crash inputs that satisfy the predicate via $N_t = N_a - N_f$. We then evaluate the mutual information I (Eq 7) between X (predicates) and Y (crash or non-crash) using Eq 8.

$$\begin{aligned} I(X;Y) =& H(Y) - H(Y|X) \\ =& -\sum_{y\in\{0,1\}} Pr(y)\log_2 Pr(y) \\ &+ \sum_{x\in\{0,1\}} Pr(x) \sum_{y\in\{0,1\}} Pr(y|x)\log_2 Pr(y|x) \end{aligned} \tag{7}$$

$$\begin{aligned} Pr(x=1) &= \frac{C_t+N_f}{C_a+N_a},\ Pr(x=0) = \frac{C_f+N_t}{C_a+N_a} \\ Pr(y=1) &= \frac{C_a}{C_a+N_a},\ Pr(y=0) = \frac{N_a}{C_a+N_a} \\ Pr(y=0|x=0) &= \frac{N_t}{C_f+N_t},\ Pr(y=1|x=0) = \frac{C_f}{C_f+N_t} \\ Pr(y=0|x=1) &= \frac{N_f}{C_t+N_f},\ Pr(y=1|x=1) = \frac{C_t}{C_t+N_f} \end{aligned} \tag{8}$$

Ultimately, the value v_j with the highest statistic is chosen as the predicate's threshold `Thr` (line 13). Note that when `<=` covers more crash inputs than `>`, the predicate's `op` is `<=` (line 9-12). Otherwise, it is `>`.

A.2 Determination of exploitation set of location for mutation

In our dataset, the inputs of several vulnerabilities are extremely long. Such inputs require a more extensive payoff table for determining the optimal locations in RL, thereby increasing computational costs. To minimize the cost to obtain higher rankings for the root causes, we only exploit a certain number of head bytes of an input with RL with the remaining bytes randomly selected. Testing values from 500 to 3,000 on three vulnerabilities with long inputs (V_{22} with 2,036 bytes, V_{25} with 4,170 bytes, V_{29} with 31,248 bytes), we found that 2,000 bytes allow RACING to achieve the highest rankings with the lowest time costs, as shown in Table 5. Therefore, we only gather rewards for the top 2,000 bytes of an input (see Section 4.1).

Table 5: Evaluation of varying byte numbers.

Note: The time duration is given in the format `hours:minutes:seconds`

#Bytes	V_{22}		V_{25}		V_{29}	
	Ranking	**Time**	**Ranking**	**Time**	**Ranking**	**Time**
500	3	00:00:30	1	00:03:54	9	01:08:18
1,000	2	00:00:33	1	00:03:16	8	00:43:36
2,000	1	00:00:40	1	00:02:32	7	00:25:58
3,000	1	00:00:45	1	00:02:59	7	00:36:36

A.3 Mutation operators

RACING uses the mutation operators defined by AFL, as shown in Table 6.

A.4 RACING's extension for compound predicates

Below we detail our adjustments to RACING$^+$ to support compound predicate of type `A AND B` (Section 6). Concretely, we make the following adjustments:

The construction of compound predicates: We constructed `A AND B` by joining two atomic predicates like `x<10` and `y>=8`, each of them associated with a variable of interest (e.g., `x`, `y`). To achieve this, we modified our implementation to enumerate all combinations of any two variables associated with atomic predicates. Meanwhile, based on the same threshold adjustment method, we slightly modified the implementation to iterate over all possible combinations of sample values for the two variables and select the pair that maximizes the mutual information between the compound predicate and the crash as the thresholds.

Table 6: Mutation operators defined by AFL.

ID	Operators Name	Meaning
1	bitflip 1/1	Randomly flip 1 bit.
2	interest 8/8	Randomly replace byte to hard-coded interesting values.
3	interest 16/8	Randomly replace word to hard-coded interesting value.
4	interest 32/8	Randomly replace dword to hard-coded interesting value.
5	arith-sub 8/8	Randomly subtract from byte.
6	arith-add 8/8	Randomly add to byte.
7	arith-sub 16/8	Randomly subtract from word.
8	arith-add 16/8	Randomly add to word.
9	arith-sub 32/8	Randomly subtract from dword, random endian.
10	arith-add 32/8	Randomly add to dword, random endian.
11	random bytes	Randomly select one byte and set the byte to a random value.
12	delete bytes	Randomly select several consecutive bytes and delete them.
13	insert bytes	Randomly copy some bytes from a test case and insert them to another location in this test case.
14	overwrite bytes	Randomly overwrite several consecutive bytes.
15	user extras (over)	Randomly overwrite bytes with user-provided tokens.
16	user extras (insert)	Randomly insert bytes with user-provided tokens.

Table 7: Evaluation on RACING+ for compound predicates.

Note: The time duration is given in the format `hours:minutes:seconds`

Vulnerability	RACING+		Aurora+	
	Ranking	Time	Ranking	Time
CVE-2017-7962	40	00:11:50	116	02:04:29
CVE-2018-19209	47	01:07:55	261	02:58:47

Optimization strategy for CoP generation: We also customized the optimization strategy for CoP generation to choose the inputs most likely to generate new inputs that violate the compound predicate. For example, for a predicate `x<10 AND y>=8`, we prioritize the selection of non-crash inputs that satisfy `x<10 AND y>=8`.

A.5 Ground truth analysis

We use V_1 as an example to illustrate how to determine the ground truth for a vulnerability. Listing 1 displays the patched code of V_1, where the developer adds a check at lines 16,177–16,181 to fix a heap buffer overflow vulnerability at line 16,204. This patch explicitly reveals that the vulnerability is related to the relative sizes of `sect->sh_size` and `sizeof(*eopt)`. To further investigate the root cause, we dynamically analyze the unpatched code (line 16,182–16,384) with a Proof of Concept (PoC) and find that during execution, the value of `sect->sh_size` is 1, and the value of `sizeof(eopt)` is 8, which results in the value of `sect->sh_size / sizeof(eopt)` being 0. In this case, the code attempts to request a buffer of size 0 through `cmalloc` (line 16,187). However, the allocator actually returns a buffer of size one, which is stored in `iopt`, and later assigned to `option` (line 16,195). Since `offset` is 0, `sizeof(*eopt)` is 8, and `sect->sh_size - sizeof(*eopt)` equals `0xfffffffffffffff9`, the program satisfies the loop condition (the unsigned comparison at line 16,197) and enters the loop, attempting to write to `option` (line 16,203–16,204). Here a buffer overflow occurs because the program tries to write to `option->size` (line 16,204), which is the second byte of `option`, exceeding the boundary of the buffer. Therefore, the fundamental reason behind V_1 is the entering of the loop when `sect->sh_size < sizeof(*eopt)`. Since the result of `sizeof(*eopt)` is fixed at 8, we identify the root cause as `sect->sh_size < 8` at line 16,197.

```
16177 + if (sect->sh_size < sizeof (* eopt))
16178 + {
16179 +  error ("The MIPS options section is too small.\n");
16180 +  return FALSE;
16181 + }
16182 eopt = get_data (NULL, filedata, options_offset, 1,
16183                  sect->sh_size, _("options"));
16184 if (eopt)
16185 {
16186   iopt = (Elf_Internal_Options *)
16187         cmalloc ((sect->sh_size / sizeof (eopt)),
        sizeof (* iopt));
16188
      ...
16194   offset = 0;
16195   option = iopt;
16196
16197   while (offset <= sect->sh_size - sizeof (* eopt))
16198   {
16199     Elf_External_Options * option;
16200
16201     eoption = (Elf_External_Options *)((char *) eopt +
          offset);
16202
16203     option->kind = BYTE_GET (eoption->kind);
16204     option->size = BYTE_GET (eoption->size);
16205
      ...
16216     offset += option->size;
16217     ++option;
16218   }
16219
      ...
16384 }
```

Listing 1: Patched code snippet of V_1

To map the root cause from the source code to the instruction level, we further analyze the assembly code of line 16,197 and find that the subtraction expression `sect->sh_size - sizeof(*eopt)` corresponds to two instructions, (1) `mov rax, qword ptr [rax + 0x20]` and (2) `sub rax, 8`. The first instruction loads the variable `sect->sh_size` (stored at address `rax+0x20`) into the register `rax`, while the second instruction subtracts `rax` by 8. Since we have identified the root cause as `sect->sh_size < 8`, we use the predicate `rax<=7` on the destination operand of the first instruction `mov rax, qword ptr [rax + 0x20]` as the ground truth of V_1.