BENZENE: A Practical Root Cause Analysis System with an Under-Constrained State Mutation

Younggi Park*† Hwiwon Lee*† Jinho Jung† Hyungjoon Koo‡ Huy Kang Kim*
*Korea University †Ministry of National Defense ‡Sungkyunkwan University

***Abstract*—Fuzzing has demonstrated great success in bug discovery, and plays a crucial role in software testing today. Despite the increasing popularity of fuzzing, automated root cause analysis (RCA) has drawn less attention. One of the recent advances in RCA is crash-based statistical debugging, which leverages the behavioral differences in program execution between crash-triggered and non-crashing inputs. Hence, obtaining non-crashing behaviors close to the original crash is crucial but challenging with previous approaches (*e.g.*, fuzzing). In this paper, we present BENZENE, a practical end-to-end RCA system that facilitates an automated crash diagnosis. To this end, we introduce a novel technique, called *under-constrained state mutation*, that generates both crashing and non-crashing behaviors for effective and efficient RCA. We design and implement the BENZENE prototype, and evaluate it with 60 vulnerabilities in the wild. Our empirical results demonstrate that BENZENE not only surpasses in performance (*i.e.*, root cause ranking), but also achieves superior results in both speed (4.6 times faster) and memory footprint (31.4 times less) on average than prior approaches.**

1. Introduction

Software fuzz testing (*i.e.*, fuzzing) has demonstrated great success in the discovery of unknown software bugs [99]. Fuzzing plays a vital role in testing commercial off-the-shelf software prior to its release. For example, Microsoft Windows 10 has been hardened using the open-source fuzzing tool, OneFuzz [112]. Google's OSS-Fuzz has reported over 8,900 vulnerabilities and 28,000 bugs across 850 open-source projects [87]. In particular, Mozilla Security has provided browser-fuzzing resources in the form of open-source [103]–[105]. The wide adoption of fuzzing has driven an increasing number of fuzzing studies [71], [84], [102], [116], [126] as well as the introduction of various fuzzing techniques [70], [76], [77], [82], [93], [119], [124].

As a product of fuzzing, a crashing input (*i.e.*, a crash) results in abnormal behavior of a program, which entails a system crash, abrupt termination, or unwanted execution. A crash indicates a failure to handle code or data in memory (*e.g.*, invalid memory access), which is often regarded as a (exploitable) vulnerability. Hence, a software developer should be able to provide a patch that fixes vulnerable code in a timely manner. However, manual debugging of each crash is typically a tedious and time-consuming task, even for highly experienced practitioners. Compared to recent advances in fuzzing techniques, the study of automated root cause analysis (RCA) against crashes has drawn relatively less attention, despite its importance.

Statistical debugging is a widely adopted debugging technique that employs a statistical approach to tracking down bugs. AURORA [69] proposes a promising direction of study with crash-based statistical debugging. Statistical debugging focuses on investigating the behavioral gaps between crashing and non-crashing program execution, for which a collection of varying program behaviors is required. Thus, given a crashing sample, AURORA leverages fuzzing to gather multiple samples and then computes a crash condition (*i.e.*, with predicates) per instruction by tracing the program. During the tracing, AURORA records every program's status, and feeds it to synthesize predicates to infer the root cause of the crash. Another approach to RCA uses a symbolic execution [68], [74], [123] technique. Recently, ARCUS [123] has leveraged both execution flow and memory snapshots to replay a crash using a symbolic execution engine. While replaying the crash, it confirms whether any state violates the pre-defined rule for a certain vulnerability type, such as a heap overflow or use-after-free.

The approach of AURORA [69] with crash-based statistical debugging demonstrates satisfactory performance. However, we identify two limitations in AURORA for scalable RCA: ① fuzzing techniques for bug discovery are inadequate for efficiently collecting the appropriate non-crashing behaviors for RCA; and ② tracing a program to capture all the memory and register values used in every exercised instruction can be time-consuming. Furthermore, ARCUS [123] adopts both a symbolic execution and rule-based approach, but faces the following barriers: ① symbolic execution engines are inherently not scalable, especially with respect to path explosion and constraint solving; and ② vulnerability types that are challenging to predefine for RCA, such as type confusion, null pointer dereference, and uninitialized variables, can be difficult to analyze. In summary, our in-depth observation indicates that, in practice, to successfully reveal a root cause (*e.g.*, even in a complex structure), an automated RCA system requires ① a suitable means for generating non-crashing behaviors that are similar to the execution paths with a crash and ② efficient tracing methods that reduce tracing overheads during RCA.

Corresponding author: Hyungjoon Koo (kevin.koo@skku.edu)

In this paper, we present BENZENE, a practical end-to-end RCA system capable of automating a diagnosis with a crash-inducing input. Our major intuition is that a (deterministic) crash must satisfy *every crash-triggering condition*. That is, the presence of a crash must satisfy a set of conditions (*i.e.*, predicates) to yield a given crash, which implies that ① a crash would not occur if any predicate(s) in the set has been negated and ② a non-crashing behavior contradicts one or more predicates in a crashing condition. Based on this intuition, we introduce an *under-constrained state mutation* scheme for direct state mutations at runtime, which assists in collecting varying non-crashing behaviors for RCA. BENZENE adopts a few mutation strategies to efficiently generate a valid non-crashing behavior that is close to an initial crash. Next, BENZENE computes a score for the behavioral similarity using a code coverage map (often adopted as a metric of the performance of a fuzzing technique). Finally, BENZENE synthesizes predicates that describe crash-inducing behavior and produces the rank of plausible locations for a root cause.

We design and implement a prototype of BENZENE that consists of three main components: ① dynamic binary analysis, ② program behavior explorations, and ③ RCA. We evaluate BENZENE using 60 real-world vulnerabilities, including bugs with complex structures, thus demonstrating its practicality in terms of both effectiveness and efficiency. BENZENE can accurately identify the location of the root cause (with a rank) for most samples (93.3%). A comparison of the performance of BENZENE to those of the state-of-the-art approaches highlights the potential of BENZENE's for realizing a practical RCA system (*i.e.*, $4.6\times$ in speed and $31.4\times$ in memory footprints on average). Notably, the speed of BENZENE achieves $8.1\times$ faster than AURORA [69] while incurring $9.1\times$ lower memory consumption on average. Meanwhile, BENZENE exhibits $1.1\times$ faster than ARCUS [123] even with $53.7\times$ lower memory on average.

The following summarizes our contributions.

- We present BENZENE, an end-to-end practical RCA system that can diagnose a problem triggered by a crash-inducing input.
- We introduce a novel technique, under-constrained state mutation with the aim of generating both crashing and non-crashing behaviors for the RCA. Besides, our approach adopts three mutation strategies that significantly improve the collection of RCA-needed program behaviors.
- We design and implement the BENZENE prototype comprising three components: dynamic binary analysis, program behavior exploration, and RCA.
- We empirically evaluated BENZENE with 60 vulnerabilities in the wild, demonstrating not only its practicality but also its superior performance compared to previous state-of-the-art approaches [69], [123].

We plan to make BENZENE an open-source system [1] to promote the field of automated RCA in the future.

1. https://github.com/zer0fall/BENZENE

2. Background

This section provides the definition of an RCA problem and previous approaches [69], [123] to RCA.

Problem Definition. Debugging is part of the software development life cycle and encompasses the overall activities that locate and correct an error (*i.e.*, bug). A crucial phase of software debugging is the RCA (also referred to as fault localization [118] in the literature) for deducing a site(s) that triggers a bug, which helps a developer to find the fault location(s). Although ordinary software testing involves importing a number of test cases with accessible source code, one may often encounter an RCA with limited information (*e.g.*, crash without source). In this work, we mainly focus on the root cause of a security-relevant bug (*i.e.*, vulnerability) when a certain crash (*i.e.*, initial crash) occurs. However, this technique can be applied to a generic RCA (§10).

Crash-based Statistical Debugging. Statistical debugging is a powerful automated technique for revealing the root cause by tracing bugs based on the success or failure of a program. In essence, it generates pre-defined predicates in a target program and collects useful information such as a predicate value or a pass-fail result. A scoring system with these predicates enumerates plausible sites that trigger a bug (*e.g.*, pinpointing candidate predicates with a rank for their root cause) such that a practitioner can quickly investigate them. In particular, *crash-based* statistical debugging requires a *crash-inducing sample* for the RCA, which is useful when only a program binary and crash are available without source code. Diagnosing the root cause under such circumstances is challenging owing to insufficient information (*e.g.*, debugging symbols) for further investigation.

AURORA. A recent advancement, AURORA [69], demonstrates a promising solution with a crash-based statistical debugging technique. First, AURORA generates multiple crashing and non-crashing samples via fuzzing, based on a given crash-inducing input. Second, it computes the crashing condition (*i.e.*, a predicate) for each instruction. For example, a predicate `[0x40010a:rax>3]` represents a condition in which a program crashes when `rax` holds any value greater than three at the address of `0x40010a`. AURORA traces a program by collecting all the values in a register and memory for every instruction. AURORA synthesizes predicates per instruction and computes the score of each candidate predicate. To compute the score, AURORA first calculates θ as defined as in Equation 1 where C and N denote the number of inputs that trigger a crashing and a non-crashing behavior, respectively, and their subscript t represents the case where the prediction by the given predicate is successful, and f otherwise.

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

Then, AURORA computes a predicate score with $2*|\hat{\theta}-0.5|$ in the range of [0,1]. As the score is closer to 1, its corresponding predicate better describes the crashing condition. Finally, AURORA ranks the predicates of each instruction based on its

```
1  int gdImageColorMatch (gdImagePtr im1, gdImagePtr im2) {
2      ...
3      buf = emalloc(0x28 * im2->colorsTotal); // 0x28 size buffer
4      for (x=0; x < im1->sx; x++) {
5          for( y=0; y < im1->sy; y++ ) {
6              color = im2->pixels[y][x]; // color set to 0x80
7              bp = buf + (color * 5);    // out-of-bound access
8              (*(bp++))++;               // crash
9              ...
```

Figure 1: Code snippet for CVE-2019-6977. A heap overflow vulnerability in Line 3 incurs an out-of-bound write due to missing a boundary check on the user-controlled variable (`im2->colorsTotal`). Note that the code has been modified for brevity.

```
1  <?php
2        $img1 = imagecreatetruecolor(0xfff, 0xfff);
3        $img2 = imagecreate(0xfff, 0xfff);
4  <     imagecolorallocate($img2, 0, 0, 0);   // crash
5  ---
6  >     for($i = 0; $i < 255; $i += 1) {
7  >       imagecolorallocate($img2, 0, 0, 0);
8  >     }                                     // non-crash
9        imagesetpixel($img2, 0, 0, 0x80);
10       imagecolormatch($img1, $img2);
11 ?>
```

Figure 2: Input mutation example for CVE-2019-6977. A fuzzer should be able to generate a non-crashing input from a given crash (*e.g.*, Line 4 replaced to Lines 6-8) for successful RCA, which requires a challenging grammar-aware mutation.

score, followed by reporting root cause candidates with their rankings. Note that the predicate synthesis metric proposed by AURORA [69] has been adopted in our study.

ARCUS. ARCUS [123] introduces another state-of-the-art direction for RCA by leveraging symbolic execution in a given crash. ARCUS reconstructs a program state on top of a series of exercised blocks and an initial memory snapshot and inspects any intermediate state using the pre-defined rules for each vulnerability type (*e.g.*, heap overflow and use-after-free). It is noted that ARCUS requires an Intel PT [89] for recording blocks. Furthermore, in contrast to an AURORA's ranking report, ARCUS makes a binary decision (*i.e.*, whether a vulnerable state has been detected) and pinpoints the location(s) that induces a bug if detected. As a final note, ARCUS does not require the disabling of address space layout randomization (ASLR), enhancing its usability.

3. Motivation and Approaches for RCA

This section demonstrates a motivating example, introduces notations, and outlines challenges and key approaches take both effectiveness and efficiency into consideration.

3.1. Motivating Example

CVE-2019-6977. Figure 1 presents a code snippet that induces CVE-2019-6977 [41], a real-world vulnerability in PHP v7.2.13 (`gd_color_match.c`). The code contains a heap overflow vulnerability in `gdImageColorMatch()` owing to insufficient memory allocation: *e.g.*, the member variable `im2->colorsTotal` is set to `0x1`, thus allocating `0x28` bytes for `buf` (Line 3). This causes `bp` (*e.g.*, `0x280`) to point to an out-of-bound address (Line 7) when the `color` is set to `0x80` (Line 6), resulting in an undesirable crash that attempts to write data in the unallocated space (Line 8).

RCA with AURORA. AURORA seeks a failure-inducing location by examining crash-explanatory predicate(s) based on a collected dataset through fuzzing. In Figure 1, finding non-crashing inputs that control `im2->colorsTotal` (Line 3) is required to infer predicates pertaining to a crash. Note that a fuzzer (*i.e.*, the AFL's crash exploration mode) offers a means of producing such a non-crashing input. Figure 2 presents a successfully mutated non-crashing input; *e.g.*, by carefully adjusting the number of `imagecolorallocate()` call invocations (Lines 6-8 in Figure 2) from a given crashing input. However, generating such grammar-aware mutations is almost infeasible [64] within an acceptable time period. Even with a successful mutation, AURORA suffers from a significant overhead of collecting all the stored values in both registers and memory because synthesizing predicates per (exercised) instruction is a requisite for RCA.

RCA with ARCUS. In contrast to AURORA, ARCUS is a rules-based RCA tool equipped with a symbolic execution engine and inspect whether every symbolic state violates pre-defined rules based on the construction of a crash execution. However, ARCUS fails to determine the root cause in our example (Figure 1) because its current rules do not cover a heap overflow case without hijacking the control flow. In general, manually registering rules that take every bug case into account (*e.g.*, heap spraying [80], [110]) is impractical.

3.2. Challenges and Our Approaches

Preliminary Definitions. We assume that the *behavior* of a target program (P) has a *distinct execution path* (*i.e.*, deterministic) with a given input I. We denote a vulnerability as V in P. Then there may be multiple Is that trigger a crash because of V. Each crash determines a unique execution path that results in a program crash (*i.e.*, crashing behavior). On borrowing symbols from propositional logic, a crash can be represented by a series of functions (*i.e.*, predicates), each of which holds either a true or false value.

Crashing Conditions. A *crashing condition* (C) can be expressed as a *compound predicate* that describes the execution with a crash-triggering input $i \in I$; *i.e.*, $C_i = p_1 \wedge p_2 \wedge \cdots \wedge p_n$. Because a distinct crash-inducing input constructs a unique crashing condition, we define a *common crashing condition* (C^*) by extracting common predicates that can be discovered across all crashing behaviors (triggered by V). Now, k different crashes form a set of crashing conditions $\{C_1, C_2, \ldots, C_k\}$ where the number of predicates may vary depending on the crashing condition. Suppose that every predicate can be mapped with a unique bit vector as $p_i \rightarrow B_{p_i}$ where the ith digit is 1 and all others are 0s (*e.g.*, $p_2 = 010_2$ and $p_3 = 001_2$). C with bit vectors can be expressed as $B_C = B_{p_1} | B_{p_2} | \ldots | B_{p_k}$, where $|$ denotes the bitwise OR operator. We can then obtain

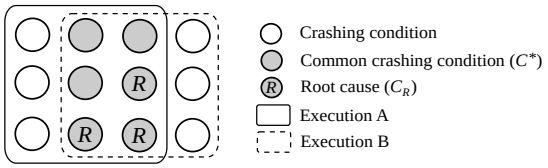

Figure 3: A high-level idea of our root cause lemma (§3.2). A common crashing condition (C^*) is derived from the collected program behaviors (*i.e.*, crashing executions A and B). The root cause predicate (C_R) must be within C^*.

$B_{C^*} = B_{C_1} \& B_{C_2} \& \ldots \& B_{C_k}$, where & represents the bitwise AND operator. The number of bits in B_{C_i} must be identical to the number of distinct predicates collected from all C_is. As an example, the common crashing condition from the following two crashing inputs would be $C_1 = p_1 \wedge p_2 \wedge p_3 \rightarrow B_{C_1} = 1110_2, C_2 = p_1 \wedge p_3 \wedge p_4 \rightarrow B_{C_2} = 1011_2 \implies B_{C^*} = 1010_2 \rightarrow C^* = p_1 \wedge p_3$.

Root Cause. A root cause (C_R) can be represented as an essential (set of) predicate(s) that contributes to a crash. As an example, for a typical buffer overflow, the missing range check that represents a predicate "buffer size > 0x400" would be the root cause. In general, a root cause predicate incorporates both explicit (*e.g.*, taking branches) and implicit (*e.g.*, missing range checks) cases.

Root Cause Lemma. Based on the aforementioned notations, we state the fundamental lemma: *C_R due to V must be within C^**. The lemma can be justified as follows: if the root cause predicate(s) do(es) not exist in C^*, then it implies that there exists a crashing input (I) that triggers V without satisfying C_R of V. This is a contradiction because we assume that C_R is the root cause to be triggered. Figure 3 depicts a common crashing condition and the root cause candidates. To obtain C^*, the example utilizes two crash-triggering inputs from two executions, A and B, and ensure that C_R is present within C^*.

Crux of BENZENE. We designed BENZENE for an RCA process at a high level with the following two major phases: (P1) extracting common crashing conditions is required, and (P2) the root cause can be derived from those conditions.

Challenges. (P1) To achieve the first phase, we adopt AURORA's approach to identify C^* from the difference in behavior between crashing and non-crashing runs of a program. However, the main challenge in investigating such behaviors arises from the *rareness* of these events because the root cause often resides in a hardly reaching location. One possible solution is to use a fuzzer to generate both (a few) crash-triggering inputs and (a vast number of) non-crashing inputs. However, the number of useful inputs for RCA with fuzzing is far from sufficient. To address this issue, (A1) we introduce state mutation to efficiently collect varying behaviors pertaining to a given crash (Approach #1). (P2) To accomplish the second phase, an efficient means of pinpointing C_R is required for practical RCA systems. However, it is non-trivial to deal with a large volume of C^* for RCA. (A2) To overcome this challenge, we propose an approach (Approach #2), wherein the root cause is inferred based on behavioral similarity.

PB	colorsTotal (line 3)		sx (line 4)		sy (line 5)		Execution Path	CS
	TV	p_1	TV	p_2	TV	p_3		
IC	0x1	T	0xfff	T	0xfff	T	3-4-5-6-7-8	✓
#1	0x400	F	0xfff	T	0xfff	T	3-4-5-6-7-8	✗
#2	0x1	T	0xfff	T	0x0	F	3-4-5-9	✗
#3	0x1	T	0x0	F	-	-	3-4-9	✗
#4	0x80	F	0x80	T	0x0	F	3-4-5-9	✗
#5	0x20	T	0xfff	T	0x100	T	3-4-5-6-7-8	✓

TABLE 1: A demonstration on how to determine the candidate(s) of root cause predicates. One can choose the first input (*e.g.*, contradicting p_1) as the best candidate to reveal a root cause because its execution path (*e.g.*, 3-4-5-6-7-8 with the line numbers in Figure 1) is the closest to the initial crash's without crashing (*i.e.*, crash-relevant-but-non-crashing input). PB, IC, TV, and CS denote a program behavior, initial crash, traced value, and crashing status, respectively. Each predicate represents p_1:`[colorsTotal < 0x80]`, p_2:`[sx > 0]`, and p_3:`[sy > 0]`.

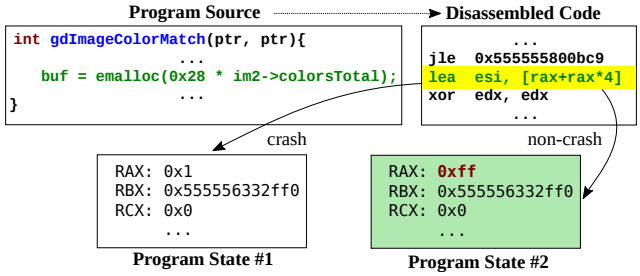

Figure 4: Example of a state mutation, diverting a program behavior for CVE-2019-6977. The source line for memory allocation corresponds to one of the disassembled codes, `lea esi, [rax+rax*4]` instruction. Our state mutation can directly update the value in the `rax` register, `0x1` → `0xff`, before fetching `lea` at runtime. The mutation facilitates the diversion of a program execution from a crashing to a non-crashing state (*i.e.*, seamless termination).

(Approach #1) A state mutation at runtime can assist in collecting varying non-crashing behaviors for the RCA. Discovering an input that induces a crash-related-but-non-crashing behavior is essential for identifying the common crashing condition, because it allows one to observe the behavioral difference that assists in uncovering the root cause (*e.g.*, #1 in Table 1). A certain non-crashing behavior is *desirable* when it contradicts a root cause (*i.e.*, behavioral difference) but results in a non-crash. However, a naïve mutation is unlikely to identify a desirable behavior because of the immense search space. To this end, we introduce *state mutation*, a mutation technique tailored to RCA that mutates a target program state at runtime. The key idea behind this technique is that part of a crash-inducing input is eventually stored in a processor register or on process memory. In this manner, obtaining non-crashing behavior is possible without considering a complex syntax (*i.e.*, grammar). Figure 4 illustrates a part of the execution flow at `gdImageColorMatch()` with a given crash. The program state is captured immediately before executing the instruction `lea esi, [rax+rax*4]`. If one could intercept the state,

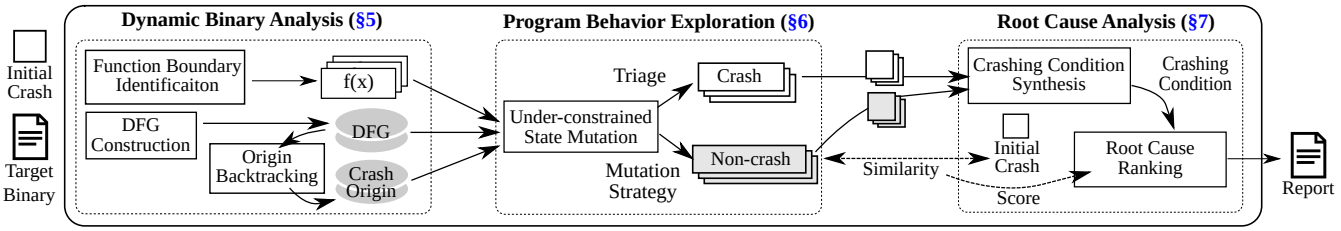

Figure 5: The architecture of BENZENE, an end-to-end system for automated root cause analysis (RCA), which comprises three components: dynamic binary analysis (§5), program behavior exploration (§6), and RCA (§7). We describe the overview of BENZENE in §4.

mutate the value of `0x1` (*i.e.*, `im2->colorsTotal`; one of the root-cause-related variables), and continue the execution, the program would terminate seamlessly without a crash.

(Approach #2) Behavioral similarity between a given crash and non-crashes helps to quickly deduce a root cause. Using our state mutation technique, it is possible to collect abundant crash-related-but-non-crashing behaviors, identifying C^*. To effectively infer the root cause, we adopt a *behavior-similarity-based* approach. The key insight of this approach is that non-crashing behavior must contradict one or more predicates under a common crashing condition. Our approach is based on the fact that, *the closer the code coverage* between a given crash and a non-crash, *the more likely* it is to reveal the root cause of V by contradicting the predicate(s) in the crash. For instance, Table 1 lists four non-crashing behaviors (#1-4; last column), and each behavior contradicts at least one predicate (*i.e.*, gray cell) in C^*. In this case, the predicate (p_1) contradicted by the first non-crashing input (#1) is most likely to become the root cause of CVE-2019-6977 because #1 is the closest behavior to the initial crash (*i.e.*, IC). We leverage Principal Component Analysis (PCA) [92] to compute a similarity score for non-crashing behaviors, as in Manes et al. [100].

Our Approach over ARCUS and AURORA. ARCUS is based on a rule and symbolic execution engine, but has two significant limitations: ① the symbolic execution engine is not scalable for complex, highly-structured programs, and ② composing a single rule can be impractical for certain bug types such as type confusion, owing to custom types (*e.g.*, structures). Meanwhile, AURORA adopts a statistical approach by collecting various behaviors pertaining to a given crash. While this idea is promising, it has two limitations: ① collecting such behaviors through fuzzing in a reasonable time may be infeasible, and ② the tracing overhead is non-negligible as it requires exhaustive tracing of all behaviors. Adopting the AURORA's statistical approach with behavior differences between crashes and non-crashes, BENZENE is designed to tackle the above shortcomings with the focus on practicality. The gist of BENZENE lies in *state mutations* for quickly identifying crash-related behaviors, which can be regarded as fuzzing tailored to RCA. Furthermore, BENZENE reduces the tracing overheads by *selectively* tracing behaviors based on their similarity and mutation strategies.

4. BENZENE Overview

Figure 5 sketches the overview of BENZENE, which consists of three main components.

Dynamic Binary Analysis. As a preparation component, BENZENE conducts dynamic binary analysis to extract essential information for the RCA (§5), including ① function boundary identification, ② data flow graph (DFG) construction, and ③ crash origin backtracking. BENZENE utilizes this information to implement various strategies for (efficient) state mutations. Note that no static analysis is required.

Program Behavior Exploration. BENZENE synthesizes a crashing condition and determines a root cause by examining which predicates in the crashing condition are contradicted by desirable non-crashing behaviors. To effectively obtain such a behavior dataset, we introduce a novel technique, called under-constrained state mutation (§6.1), which directly mutates a program state (*e.g.*, a value in a register or memory) at runtime. We adopt three mutation strategies to make our technique practical and robust (§6.2). Moreover, we introduce a conservative crash triage to gather a minimum set of crashes relevant to a given bug, which is required for synthesizing the crashing condition (§6.3).

Root Cause Analysis. In the final phase, BENZENE infers a set of possible locations (for further investigation) that trigger a given crash based on non-crashing behaviors. Thus, BENZENE first constructs a matrix that portrays the edge coverage of each collected behavior and, then computes a similarity score between the coverage of an initial crash and that of a non-crashing behavior (§7.1). Next, BENZENE synthesizes predicates that describe a crashing condition based on a non-crashing behavior (§7.2). Finally, BENZENE ranks all the predicates that contribute to a given crash using both the synthesis of the crashing conditions and the similarity score of each non-crashing behavior (§7.3).

5. Dynamic Binary Analysis

BENZENE performs a dynamic binary analysis to provide crucial information for further RCA.

Function Boundary Identification. BENZENE begins with a binary analysis by recognizing ① the location of each function, and ② the instructions within. Based on the execution with a given crashing input, BENZENE dynamically extracts function information by monitoring a call stack with instrumented instructions (*e.g.*, a `call`/`ret` pair).

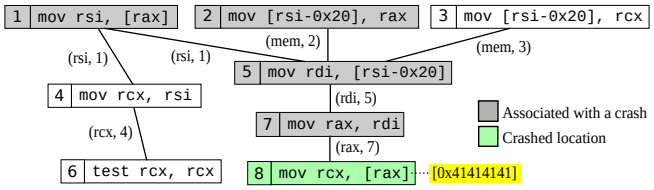

Figure 6: Example of building a data flow graph for backtracking a crash origin. A square box represents an instruction (*i.e.*, node) with its identifier, and a line between boxes represents a data flow (*i.e.*, edge). Note that a gray box depicts a crash-relevant node. A node may have multiple edges because data in a source operand can be associated with multiple origins (*e.g.*, $inst_5$).

Data Flow Graph Construction. BENZENE internally builds a DFG with a given crash for two reasons: ① to identify the entry node of a function, and ② to infer the values that a certain operand can hold, which assists further mutations (§6.2). By running a program with an initial crash, we build a dynamic DFG ($G(V, E)$) as follows. First, each instruction (node) is labeled with a unique identifier for tagging the registers and memory regions. Second, we represent a data flow (edge) with a tuple of $(source, inst\#)$ because ① data propagation can be tracked via a register or memory (*i.e.*, source operand in an instruction), and ② a single instruction could be reached from multiple instructions. Third, for each exercised instruction, we examine which $inst$'s identifier is tagged on the source operand, and then, append the incoming edge ($source$, $inst\#$) based on the identifier. Next, we tag the destination operand using the current instruction's identifier. Figure 6 illustrates a DFG example with an instruction (*i.e.*, square box) and data flow (*i.e.*, line). In the case of $inst_7$, `rdi` (*i.e.*, source operand) is tagged by $inst_5$'s identifier (*i.e.*, `rdi` originates from $inst_5$); thus, the incoming edge (`rdi`, 5) is added to $inst_7$. Note that the source operand `[rsi-0x20]` at $inst_5$ has three incoming edges, $(\texttt{rsi}, inst_1)$, $(\texttt{mem}, inst_2)$, and $(\texttt{mem}, inst_3)$, because data dependencies can be associated with multiple origins.

Crash Origin Backtracking. Based on the DFG, BENZENE traces backward to obtain the *exact origin*s of a crash site (*e.g.*, invalid memory access), leveraging a debugger's reverse execution [107]. We use this information to extract target functions for our state mutation. Namely, starting from a crashing location, BENZENE follows back to the incoming edges that are relevant to a crash-inducing value. In Figure 6, suppose that a crash has been triggered at $inst_8$ because of the invalidity of a dereferencing value (*e.g.*, `0x41414141`) of the `rax` register. Following relevant data reversely, the `rax` value originates from `rdi` at $inst_7$, and so does `rdi` from `[rsi-0x20]` at $inst_5$. Similary, the operand `[rsi-0x20]` at $inst_5$ can be affected by two instructions: $inst_1$ and $inst_2$. This example entails four instructions ($inst_{[1|2|5|7]}$) as the origin of a crash. Based on the observation that an invalid value (`0x41414141`) arises from $inst_2$, $inst_3$ has been excluded. A set of instructions associated with the crash origin is provided at the mutation phase (§6.2).

6. Program Behavior Exploration

This section presents our mutation scheme (§6.1) and strategy (§6.2) for non-crashing behavior and crash triage (§6.3) for crashing behavior.

6.1. Under-constrained State Mutation

To obtain crash-related non-crashing behaviors (Ⓟ1 in §3.2), we adopt an approach that mutates a program state at runtime. At this point, one may raise the question of ensuring the validity of the behavior from such a state mutation because forceful modification of a program's intermediate state often renders a subsequent execution invalid (*e.g.*, an unreachable state). However, in contrast to fuzzing, our state mutation does not need to reason the validation of the input, but *harnesses* the mutation itself to obtain a non-crashing behavior. In this regard, our mutation technique is called *under-constrained state mutation*, which has been borrowed from under-constrained symbolic execution [109]. Namely, a non-crashing behavior is a (by-)product of an unsatisfied crashing condition, even in the presence of an infeasible state, which can aid in seeking the candidate predicates of the root cause (*i.e.*, crashing condition). To exemplify, the `im1->sx`'s mutation from `0xfff` to `0x0` in Line 4 in Figure 1 results in a non-crash because it does not hold one of the predicates in the crashing condition, [`im1->sx` > `0x0`]. Although this mutation leads to an infeasible execution by guiding an unreachable branch (*i.e.*, escape without a loop), its behavior is useful for obtaining a predicate for the RCA. We elaborate on the crashing behaviors' validity with under-constrained state mutations in §6.3.

Notation. Let an exercised instruction be $i \in \Omega$ where Ω represents all instructions in a certain program. Now we define a state as $S_i(\mathcal{R}, \mathcal{M})$ at the very moment of executing instruction (i), where the current status of the registers and memory are denoted as $\mathcal{R}$ and $\mathcal{M}$, respectively. To describe a state mutation (T), we define two attributes: ① source type: $t \in \{opr, mem\}$ where opr and mem denote a source operand or memory to mutate from, and ② value: $v \in Z_n = \{x | x \in$ all representable integers with n bits$\}$. Then, a state mutation can be formally written as $T_i(t, v, v') : S_i(\mathcal{R}, \mathcal{M}) \rightarrow S'_i(\mathcal{R}', \mathcal{M}')$ where a state transition is made with a value (v to v') of type (t). For example, the state mutation on $i :=$ `lea esi, [rax+rax*4]` in Figure 4 can be written as $T_i(\texttt{rax}, 0, 255) : S_i(\mathcal{R}, \mathcal{M}) \rightarrow S'_i(\mathcal{R}', \mathcal{M}')$. Note that BENZENE focuses solely on a value mutation in a source operand because the value in a destination operand is automatically determined by its sources.

Challenge. Despite the effectiveness of an under-constrained state mutation in diversifying a program behavior, a naïve mutation often encounters a state explosion owing to too many possible cases. For instance, the above $T_x(rax, 0, 255)$ is hardly reached with a random mutation strategy because a 64-bit register (rax) could hold $2^{64} - 1$ possible values. Moreover, as a state mutation requires forceful execution by setting up an arbitrary value in $S_i(\mathcal{R}, \mathcal{M})$ on the fly, a

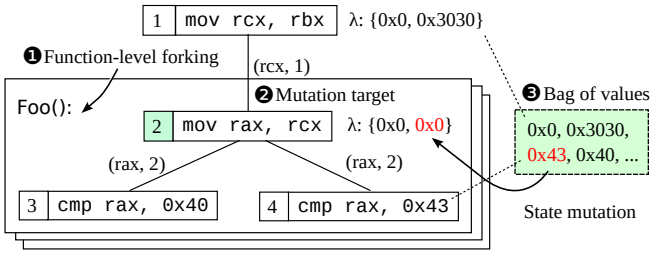

Figure 7: Illustration of our mutation strategy for an effective and efficient under-constrained state mutation. ❶ BENZENE adopts a function-level process `forking` to narrow down the scope of a mutation. Besides, BENZENE drastically reduces ❷ the number of target instructions with a DFG (*e.g.*, instruction that refers an external value), and ❸ the number of observed values with a bag (*i.e.*, BoV; Bag of Values) for guiding further mutation. λ represents trace values observed in a source operand of each instruction.

subsequent execution would be highly unstable or impractical (*e.g.*, meaningless crashes). The following section describes our mutation strategies for efficiency and practicality.

6.2. Mutation Strategy

Mutation at a Glance. At a high level, similar to LibFuzzer [113] and in-memory fuzzing, we explore the behaviors of a program at the function level by creating a new process with `fork`-ing. Once these target functions are extracted, we repeat the state mutations with a fixed number of iterations per function. At each iteration, we choose a *mutation target* $S_i(\mathcal{R}, \mathcal{M})$. When the mutation target (S_i) has been reached, we perform a state mutation (T_i), modifying the current state of the program accordingly. After the mutation, we continue the process until termination and monitor a (non-)crash. Note that mutating the program state makes it susceptible to an undesirable crash; thus, we devise strategic mutations.

Function-level Forking. As there are a large number of instructions in a program, we need to narrow the location scope for a state mutation. To address this problem, we adopt the granularity of a function for the mutation process (*e.g.*, `Foo()` in Figure 7). To be specific, we extract a list of functions pertaining to invalid memory access at a crash site. This can be performed using function information and crash origins during a dynamic binary analysis (§5). Once the target functions are identified (*e.g.*, 10 minutes), we iterate a state mutation for each target function (F); *i.e.*, ① choose a state mutation $T_i : S \rightarrow S'$ within a target function (*i.e.*, $i \in F$), ② execute an instruction, and ③ monitor a mutation result (*i.e.*, crash or non-crash). For each iteration, the process is forked at the entry of a target function.

Mutation Target Selection. For each forked process at a target function, we must choose a state $S_i(\mathcal{R}, \mathcal{M})$, as a mutation candidate. Although we restrict our mutation scope to the target function, there are still a large number of instructions within the function. To address this issue, we narrow the instructions to a selection pool of instructions (*i.e.*, $i \in P, P \subset F$) that refers to an external value (coming from outside a function boundary). The intuition behind is that the behavior of a certain function can be determined by external values (*e.g.*, arguments or global variables); thus, mutating them can aid in deriving desirable behaviors. We leverage the DFG (§5) to identify instructions that use such external values at the binary level. Considering $inst_2$ in Figure 7 where the incoming edge of $inst_2$ is $(\texttt{rcx}, inst_1)$. The affecting value (*i.e.*, value in `rcx`) originates from an external function (*e.g.*, caller). As $inst_2$ takes an externally referencing value, we select $S_{inst_2}(\mathcal{R}, \mathcal{M})$ as a mutation target in P. By contrast, $inst_3$ whose data flow comes from $inst_2$ can be excluded from P because `rax` is dependent on `rcx`. It is noteworthy that our approach is agnostic to a calling convention (*i.e.*, a scheme for how a subroutine takes an argument from a caller, such as `stdcall`, `cdecl`, or `fastcall`) because we inspect a value if it comes from either a register or memory with the boundary of a function.

Type	Description
`geq`$(\texttt{op}, \alpha)$	all values in `op` are greater than or equal to α
`leq`$(\texttt{op}, \alpha)$	all values in `op` are less than or equal to α
`exist`$(\texttt{op}, \alpha)$	α exists in `op`
`strlen_geq`$(\texttt{op}, \alpha)$	string length pointed by `op` is greater than or equal to α
`strlen_leq`$(\texttt{op}, \alpha)$	string length pointed by `op` is less than or equal to α

TABLE 2: Predicate types for BENZENE. We adopt the first two types from AURORA, and introduce three additional types.

Bag of Values. An under-constrained state mutation is prone to leading to an undesirably invalid crash due to its inherent instability. Our careful observation heuristically reveals two major cases that frequently trigger such an invalid crash: ① mutating a pointer that refers to a user-defined data type (*e.g.*, structure) and ② mutating a constrained variable with a distinct range (*e.g.*, `enum` type, array index). To handle both cases, we maintain a *bag of values (BoV)*, λ, to guide further mutations. The bag is collected internally with a set of observed values (from a source in accordance with an instruction) per $S_i(\mathcal{R}, \mathcal{M})$. Thus, a mutation candidate (v') can be selected from (*i.e.*, $v' \in \lambda$) for $T_i(t, v, v')$. The intuition behind this strategy is that a value that shares the same data flow may have an identical data type with a high probability [128]. This aids in observing acceptable behavior without worrying about complicated type inferences on a binary. However, it provides an opportunity to cover a large mutation space within an acceptable time constraint. We cultivate a BoV per S from the stored values of an exercised instruction by backtracking the incoming edges (*i.e.*, up to the point with a single source edge in DFG) in a DFG (*e.g.*, ❸ in Figure 7). A special value is of interest for the BoV, which can enter a branch (*e.g.*, `0x40` from `cmp rax, 0x40`), enabling us to efficiently collect non-crashing behaviors such as prior works in fuzzing [76], [111], [124]. In case that BoV exceeds a pre-defined size (*e.g.*, 1024), BENZENE conducts random sampling. With BoV, BENZENE generates random values (RVs), probabilistically choosing a value from either BoVs or RVs (*e.g.*, 50%) for each mutation cycle. Note that we disabled ASLR because the memory layout for each mutation must be consistent during our mutation.

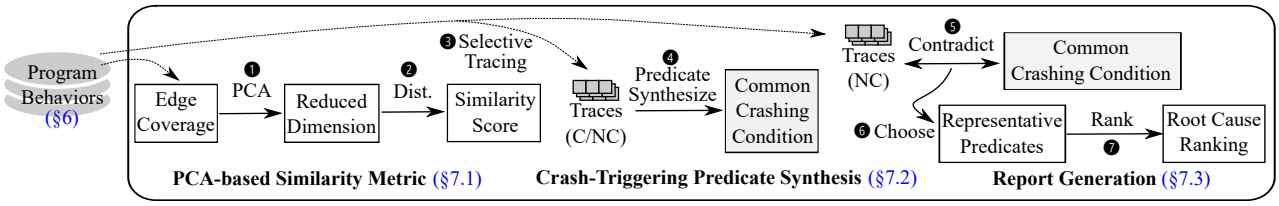

Figure 8: Root cause analysis overview in BENZENE. Each program trace contains an edge coverage map, forming a matrix of all program behaviors. Next, we apply PCA to reduce a dimension of the matrix (❶), followed by computing a similarity score (❷) between a non-crashing and a crash-inducing execution. To obtain a crashing condition with multiple predicates, we group the traces (❸) and use a predicate synthesis approach proposed by AURORA [69] (❹). By inspecting concrete values from traces (with non-crashing behaviors), we identify a contradictory predicate against the condition (❺). BENZENE elects a single representative predicate (❻) from all contradictory predicates for each non-crashing behavior (*i.e.*, earlier execution is more probable to be a root cause). Finally, we rank all representative predicates based on their similarity scores (❼). C and NC denote traces for crashing and non-crashing behaviors, respectively.

6.3. Crash Triage

Although our under-constrained state mutation technique is mainly focused on collecting non-crashing behaviors, it is necessary to collect a *minimum* set of crashing behaviors because we utilize a means of AURORA's predicate synthesis (Equation 1). However, in contrast to a non-crashing behavior, the validity of which can be confirmed, it is difficult to determine whether a crash caused by mutating a state has been associated with an original crash. For example, if a target that originally represents a particular pointer is mutated to an invalid integer (*e.g.*, `0x1`), a crash (irrelevant to a bug) would be pointless. Such *invalid* crashes significantly hinder the synthesis of accurate crashing conditions by introducing substantial noise into Equation 1. Based on a previous reverse execution triage technique [78], [79], [121], we classify a certain crash into the same category if it follows identical data flows. However, constructing a DFG for every crashing execution is costly, with a significant overhead. To tackle this problem, we devise a *crash triage* technique that distinguishes an execution with a given crash (*i.e.*, root cause) from that with an invalid state mutation while incurring a low overhead. The technique entails the following two conditions: ① a crashing site (*i.e.*, address) triggered by a state mutation is identical to that of an initial crash, and ② an instruction associated with a crash origin is exercised. If a crash with a state mutation satisfies the above conditions, we record this behavior; otherwise, we discard it. Apparently, our method may result in the synthesis of an over-approximated crashing condition by considering only a subset of all possible crashes due to V; however, C_R would still be preserved with the subset. In other words, C_R must satisfy (a subset of) crashes that non-crashes must not.

7. Root Cause Analysis

BENZENE extracts a common crashing condition from the collected behaviors and derives a root cause by ranking the predicates corresponding to each non-crashing behavior (*i.e.*, contradicted predicates). This section describes the RCA component of BENZENE (Figure 8).

7.1. PCA-based Similarity Metric

Edge Coverage Matrix. Similar to AFL [125], we utilize an edge coverage map that is widely adopted to measure the performance of a fuzzing technique. BENZENE records edge coverage information (*i.e.*, the occurrence of transitions between basic blocks) per collected behavior on a 64 KB-size map. The map keeps being updated along with an iterative mutation. With n different behaviors, we can obtain an edge coverage matrix of n (row) $\times$ 64 KB (column) in size. One notable side effect is that the coverage map of an execution with a non-crashing behavior would keep updated until the termination of a program, whereas that of an execution with a crash would not. This inevitably brings about a significant discrepancy between the coverage maps for comparison, which hinders our major objective of finding non-crashing behaviors similar to an original crash. To mitigate this issue, we harness a watchpoint that informs us when to stop updating the coverage map. To be precise, the watchpoint monitors one of the crash origins (§6.3); a coverage map update would be suspended for a subsequent execution once a crash-triggered instruction (from a given crash) has been exercised.

PCA-based Similarity. To reduce the dimensions of the edge coverage matrix, we apply PCA [92] to obtain a compact coverage map for each collected behavior (❶ in Figure 8). Then, we compute the Euclidean distance (*i.e.*, L2 norm) between the coverage of a non-crashing behavior and the initial crash. BENZENE determines how close an execution with a certain non-crashing behavior is to that with the original crash via a similarity score (❷ in Figure 8): *i.e.*, the smaller distance, the closer score. This similarity scoring allows for prioritizing all the collected non-crashing behaviors in proximity to an initial crashing execution.

7.2. Crash-triggering Predicate Synthesis

Selective Tracing. Mutating a target function at each fuzzing cycle results in either a crashing or non-crashing behavior. However, tracing the entire program behavior with such mutations is computationally expensive [69]. Furthermore, because the traces prior to a state mutation are always the

same, they cannot be used to the AURORA metric Equation 1. To address both the tracing overheads and unchanging traces owing to discrepant mutation points, we introduce the concept of a *tracing group* that consists of behaviors obtained by state mutations with the same target function. Behaviors that belong to the same tracing group can be utilized to calculate Equation 1 because they share an adjacent mutation point (*i.e.*, target function). We then prioritize the tracing groups with a similarity metric. Specifically, BENZENE considers the Top-N non-crashing (program) behaviors (*e.g.*, $N = 50$), and *selectively* traces the tracing group whose non-crashing behaviors are shown in the Top-N instead of performing exhaustive tracing (❸ in Figure 8). As our RCA process employs the contradictory predicate for each non-crashing behavior (Ⓐ2 in §3.2), we only need to consider the tracing groups that incorporate the non-crashing behaviors in Top-N.

Predicate Synthesis. Based on the results obtained from selective tracing (*i.e.*, various program behaviors per tracing group), BENZENE performs *predicate synthesis* as proposed by AURORA [69]. The objective of predicate synthesis is to identify a *yet unknown* crashing condition (*i.e.*, a compound predicate) using traces from each exercised instruction (❹ in Figure 8). Table 2 enumerates the predicate types for BENZENE. Unlike AURORA, we exclude the two predicate types related to control-flow and flag predicate and introduce three new predicate types, namely, `exist`, `strlen_geq` and `strlen_leq`. As discussed in §3.2, BENZENE can derive a common crash condition by extracting common predicates from all crash-triggering behaviors. It is noteworthy mentioning that our approach can handle a missing condition because a synthesis is based on the behavioral differences. For example, BENZENE can synthesize p_1 (Table 1) with the observation of `colorsTotal` despite the check on `colorsTotal` has been missing (Figure 1).

7.3. Report Generation

Root Cause Inference. At this point, we extract a common crash condition (*e.g.*, the 12 circles in Figure 3). For each non-crashing execution, it is necessary to discover a predicate contradictory to the crashing condition (❺ in Figure 8). For example, the three predicates outside of Execution A in Figure 3 are a set of contradictory predicates. We choose a single representative predicate for each non-crashing behavior by inspecting the order of the exercised instructions (❻ in Figure 8). The intuition behind this is that an early-exercised instruction is more likely to become a *root cause* because its subsequent behaviors are more likely to propagate a bug (*e.g.*, crash).

Root Cause Ranking. Now, a representative predicate is selected from all the contradictory predicates per non-crashing behavior. In the final phase of RCA, BENZENE sorts out all representative predicates according to similarity scores (between crashing and non-crashing behaviors) from §7.1 (❼ in Figure 8). Next, we rank the representative predicates based on the location of the possible root cause. Finally, BENZENE produces a ranking report for users. Note that we only consider ranks up to 50, as in AURORA [69].

8. Implementation

We implement the prototype of BENZENE with three components on x64. In summary, BENZENE consists of 13K line of C/C++ and Python.

Dynamic Binary Analysis. For the dynamic DFG construction, we utilize the enhanced `libdft` [76], [94], which is a taint analysis library based on Intel's PIN [98]. To obtain a function boundary, we implement a built-in callstack monitor for BENZENE, which instruments pairs of `call` and ret with PIN. Additionally, we leverage the Mozilla RR's reverse execution feature [107] to traverse back to the origin of the data flow from a crashed site (*i.e.*, invalid memory access).

Program Behavior Exploration. We build the BENZENE mutation engine using DynamoRIO [72], a dynamic binary instrumentation (DBI) framework, for our under-constrained state mutation. Note that we harness DynamoRIO because of its speed (*e.g.*, slower) and stability (*e.g.*, failure to catch a segmentation fault when the `fork` is called) of PIN. The mutation engine incorporates the feature of tracing a program because it holds a common code base. We implement a mutation server for efficient management of mutation processes such as monitoring and control. We use the SQLite3 database [62] to store program traces including register and memory values, during execution.

Root Cause Analysis. The RCA component for computing coverage similarity and predicate synthesis is written in Python. We adopt `sklearn` [61] for computing PCA [92].

User-configurable Mutation Parameters. BENZENE allows one to configure the following parameters: time to discover target functions, the number of state mutations per function, and BoV size where BENZENE pre-define them as 10 minutes, 10, and 1,024, respectively.

9. Evaluation

We evaluate BENZENE using Ubuntu 20.04 equipped with Intel(R) Xeon(R) Silver 4210 (20 cores at 2.20 GHz) and 128 GB RAM. As ARCUS's tracing requires Intel-PT support, we ran the evaluation in a different environment: Ubuntu 20.04 with Intel(R) Core(TM) i7-8700 and 64 GB RAM. The resource consumption during ARCUS's tracing is negligible. We conduct all other experiments, including ARCUS's analyses, in the same environment. Finally, we disable the ASLR because BENZENE requires that RCA be performed in a deterministic environment. We assess BENZENE with the following three research questions.

- **RQ 1:** How well can BENZENE identify the root cause of a crash in a real-world program that supports both complex and highly structured inputs, and various bug types? (§9.2)
- **RQ 2:** How closely can BENZENE predict a root cause (*i.e.*, predicate ranking) compared to an actual patch? (§9.3)
- **RQ 3:** How efficient is BENZENE in RCA? (§9.4)

9.1. Dataset and Experimental Setup

Dataset. We choose 60 vulnerabilities from real-world applications that contain 11 bug types: stack overflow, heap overflow, integer overflow, use after free, double free, type confusion, uninitialized variable, null dereference, format string, global overflow, and division by zero. To be precise, we employ 21 solid samples (*i.e.*, PoCs; proof-of-concepts) provided by AURORA [69], and another 22 samples from ARCUS [123] for direct comparison. We exclude a vulnerability that requires a multi-threading or multiprocessing environment because BENZENE supports RCA on a single CPU core. To highlight the performance of BENZENE with various complex programs, we choose 17 additional samples from real-world projects including PDF, multimedia, database engine, interpreters, and libraries. We have listed all the sample information in Table 5 of the Appendix including the CVE or Issue number, bug type, and total number of instructions to estimate the RCA complexity of each case.

Evaluation with AURORA. We choose 17 samples (01–17) from BENZENE, 19 samples from AURORA (18–27, 29–35, 37–38), 10 samples from ARCUS (39, 41, 42, 44, 49, 50, 55–58), evaluating 46 samples in total. We exclude 12 samples that cannot be processed by AURORA, such as those requiring unsupported fuzzing methods that handle argument strings or environment variables. Furthermore, 28 and 36 are also excluded because AFL was unable to process those cases properly. Because AURORA is dependent on AFL 2.52b [125], we build fuzzing-target binaries (from AURORA) using `afl-gcc`. It should be noted that when any sample requires a sanitizer to yield a crash, we use a specific sanitizer module such as an Address SANitizer (ASAN) or Memory SANitizer (MSAN).

Evaluation with ARCUS. We choose 4 samples from BENZENE (12–14, 16), 8 samples from AURORA (31–38), 22 samples from ARCUS (39–60), evaluating 34 samples in total. Note that ARCUS cannot run 26 samples because ① it requires an exploitation that involves with a control flow hijacking (*e.g.*, stack overflow, heap overflow), and ② certain bug types (*e.g.*, type confusion) are missing in a pre-defined rule.

Evaluation Criteria. In our evaluation, we mainly compare BENZENE with AURORA [69] and ARCUS [123]. For the AURORA evaluation, we run fuzzing on each sample for an hour. If the root cause for the given sample was not found in a report, we perform fuzzing for another 5 hours (*i.e.*, 6 hours in total). For the ARCUS evaluation, we set up a 20-hour timeout because ARCUS requires sufficient time for specific samples (Figure 9). To identify the root cause of a given crashing input, both BENZENE and AURORA generate a report that includes an inference of a buggy location with a predicate. Both approaches predict a list of predicates with ranks (*i.e.*, root cause candidates). Hence, we regard a root cause predicate within the 50th rank as a success, and otherwise, a failure. By contrast, ARCUS produces a (possible) buggy location directly with a binary judgment of "`Matched`" or "`Not-matched`". Therefore, a ranking comparison between BENZENE and ARCUS is infeasible.

Comparison Methodology. For a fair comparison of the RCA results, we consider a patch for each bug as the ground truth. Our success criteria for the RCA is satisfied if any predicate within the ranked list correctly identifies the exact location (*e.g.*, a variable, branch, or statement) patched by a developer. However, multiple repairs that are semantically equivalent can exist for a given bug. Therefore, we manually inspect the root cause predicate by verifying that it points to the patched location at the instruction level. As mentioned in [69], we assume the RCA is successful for BENZENE and AURORA when the root cause is located in the result report within the 50th rank.

9.2. Effectiveness of BENZENE

Summary. Table 3 summarizes the effectiveness of BENZENE compared to AURORA and ARCUS. Δ`Root` counts the number of exercised instructions between the location of the root cause and that of a crash. Interested readers may refer to the "Total" number of all instructions to reach that crash in Table 5 of the Appendix, which estimates an approximate complexity of each RCA process. The efficacy of our BoV technique is evidenced in Table 3. The samples with ✱ indicate the contributions of the BoV in discovering desirable behaviors during a mutation process: eight vulnerabilities among the 60 samples, and their RCAs were unsuccessful otherwise. For example, 08 reveals that a valid pointer (*e.g.*, `0x7ffff6e9ec18`) is required to discover a meaningful input during fuzzing. Generating a valid pointer from purely random mutations is not viable without a BoV. Interestingly, BENZENE demonstrates effectiveness regardless of the complexity of a program (*e.g.*, an input that follows a complex structure) or the bug type of the crash-inducing input, which highlights the strength of our state mutation approach. Interested readers refer to our case studies in Appendix A.4.

Comparison with AURORA. With AURORA, BENZENE reveals root cause locations with a success rate of 95.6% (44 out of 46 cases), whereas AURORA has a success rate of 63.0% (29 cases). AURORA has 17 RCA-failing cases with seven different bug types. We reveal several primary factors that hinder AURORA's RCA including: ① fuzzing failure (4 cases) ② insufficient collection of non-crashing behaviors (12 cases), and ③ unsupportive predicate type (1 case). We provide more information in Appendix A.1.

Comparison with ARCUS. With ARCUS, BENZENE reveals root cause locations with a success rate of 94.1% (32 out of 34 cases), whereas ARCUS has a success rate of 35.2% (12 cases). BENZENE outperforms ARCUS in terms of effectiveness and applicability. We identify several primary factors that impede ARCUS's RCA including: ① symbolic execution failure, ② inconsistent executions between a symbolic engine and tracer, ③ constraint-solving failure, and ④ inadequate rules to describe a bug type. Further information is provided in Appendix A.2.

✓: Root Cause Found, ✗: Root Cause Not Found, ✱: Bag of Values Contributed, ⇏: Unable to Proceed, ⊘: Out of Scope

Target (BoV)	Sanitizer	ΔRoot (%)	RCA BEN	RCA AUR	RCA ARC	RCA M‡	Target (BoV)	Sanitizer	ΔRoot (%)	RCA BEN	RCA AUR	RCA ARC	RCA M‡
01 PHP-8865 [39]	-	3,325 (<0.1%)	1	✗	⊘	✓	31 mruby-10199 [30]	ASAN	112,609 (0.7%)	5	25	✗	✓
02 PHP-6977 [41]	ASAN	132 (<0.1%)	1	✗	⊘	✓	32 Lua-6706 (✱) [21]	-	147 (<0.1%)	1	12	⇏	✓
03 mruby-0525 (✱) [26]	-	-	2	✗	⊘	✗	33 nasm-8343 [32]	-	457,901 (47.5%)	2	2	⇏	✓
04 Poppler-12293 [45]	-	1,287,909 (<0.1%)	9	✗	⊘	✓	34 Sleuthkit [51]	-	483 (<0.1%)	1	5	✗	✓
05 SoX-11358 [52]	ASAN	79,470 (11.6%)	40	✗	⊘	✓	35 libzip-12858 [20]	-	46,218 (13.0%)	9	1	⇏	✓
06 TinyCC-20375 [58]	-	57 (<0.1%)	1	✗	⊘	-	36 Python-5636 [47]	-	828,077 (0.8%)	5	⊘	✗	✓
07 mruby-46020 (✱) [25]	-	7,249 (0.3%)	16	✗	⊘	✓	37 bash [6]	-	75,418 (5.3%)	2	5	⇏	✓
08 PHP-7226 (✱) [42]	-	5,134 (<0.1%)	1	✗	⊘	✓	38 mruby-10191 [24]	-	3,251 (<0.1%)	3	35	✗	✓
09 PHP-0273 (✱) [43]	-	266 (<0.1%)	2	✗	⊘	✓	39 libpng-0597 [18]	-	4,471 (1.7%)	2	3	✓	✓
10 libical-11706 [16]	-	25,028 (0.6%)	19	✗	⊘	✓	40 jpegtoavi-1279 [13]	-	125,152 (37.6%)	1	⊘	✓	-
11 SoX-8356 [53]	-	4,194,431 (<0.1%)	2	✗	⊘	-	41 o3read-1288 (✱) [35]	-	10,859 (4.2%)	1	1	✓	✗
12 mruby-181321 [29]	-	2,578 (<0.1%)	11	29	✗	✓	42 autotrace-9167 [3]	ASAN	7,507 (0.6%)	5	(24)	✗	-
13 PHP-2386 [40]	-	736 (<0.1%)	3	✗	✗	✓	43 Redis-12326 [50]	-	8,514 (0.7%)	1	⊘	⇏	✓
14 Poppler-7310 [46]	ASAN	0 (0.0%)	2	8	✗	✓	44 ftp-15705 [9]	-	19,094 (13.1%)	3	11	⇏	-
15 SQLite-16168 [54]	-	15,526 (1.1%)	6	✗	⊘	✓	45 gif2png-5018 [8]	-	76,428 (24.7%)	1	⊘	✓	✓
16 SQLite-13434 [55]	-	79,416 (5.8%)	2	10	⇏	✓	46 dmitry-7938 [7]	-	56,330 (10.4%)	1	⊘	⇏	-
17 libbfd-8393 [14]	ASAN	7 (<0.1%)	1	19	⊘	✓	47 ntpq-12327 [34]	-	349,775 (29.2%)	1	⊘	✓	✓
18 readelf-9077 [49]	ASAN	917 (0.1%)	2	1	⊘	✓	48 libiec-18957 [17]	-	167 (<0.1%)	1	⊘	✓	✓
19 objdump-9746 [36]	ASAN	1,778,797 (27.7%)	3	3	⊘	✓	49 pdf-re-14267 (✱) [37]	-	181,576 (9.6%)	2	1	⇏	✓
20 tcpdump-16808 [57]	ASAN	5,341 (0.5%)	2	3	⊘	✓	50 abc2mtex-1257 [2]	-	28,688 (10.7%)	1	✗	✓	✓
21 perl-17384 [38]	ASAN	30,139 (2.3%)	9	36	⊘	✓	51 abc2mtex-47254 [1]	-	2,944 (1.3%)	1	⊘	✓	✓
22 patch-54558 [10]	ASAN	18,955 (6.0%)	4	3	⊘	✓	52 MiniFtp-46807 [22]	-	778 (0.3%)	1	⊘	✓	-
23 mruby-12248 [23]	ASAN	1,249 (<0.1%)	6	1	⊘	✓	53 GM-11403 [12]	ASAN	0 (0.0%)	✗	⊘	⇏	✗
24 nasm-16517 [31]	-	471,868 (49.4%)	1	15	⊘	✓	54 GM-14103 [11]	ASAN	70,009 (0.9%)	✗	⊘	⇏	✗
25 mruby-185041 [27]	-	1,322 (<0.1%)	✗	29	⊘	✗	55 autotrace-9182 [5]	-	1,512,761 (0.8%)	1	(4)	⇏	-
26 Python-116286 [48]	-	1,108 (<0.1%)	✗	✗	⊘	✗	56 libtiff-2025 [19]	-	61 (<0.1%)	3	3	⇏	✓
27 mruby-3947 [28]	MSAN	0 (0.0%)	8	14	⊘	✓	57 libexif-2645 [15]	-	400 (0.1%)	1	7	⇏	✓
28 PHP-11038 [44]	MSAN	1,766,918 (15.1%)	8	⊘	⊘	✓	58 autotrace-9186 [4]	-	794 (<0.1%)	1	(16)	✓	-
29 Xpdf (✱) [60]	ASAN	0 (0.0%)	1	✗	⊘	✓	59 typespeed-0105 [59]	-	752 (0.2%)	1	⊘	✓	✓
30 nm-21670 [33]	ASAN	0 (0.0%)	2	✗	⊘	✓	60 sudo-0809 [56]	-	339,050 (33.7%)	1	⊘	✓	✓

TABLE 3: Comparison of RCA results between BENZENE (BEN), AURORA (AUR), and ARCUS (ARC). ΔRoot denotes a distance with the number of exercised instructions between a root cause and a crashing point, which represents the complexity of an RCA task. We compare a root cause predicate of BENZENE with that of AURORA (ranking) and ARCUS (presence). The symbols in the table include: [✓] when a root cause has been successfully discovered, [✗] when no root cause has been found, [✱] when the BoV contributes to RCA, [⇏] when an RCA is unable to proceed, and [⊘] when a sample is out of scope. M‡ denotes whether a root cause reported by BENZENE is aligned with an actual patch. In case of no available patch, we resort to perform a manual analysis. Note that the AURORA's rank with a parenthesis indicates the case for which we adjust its threshold (*e.g.*, $T : 0.9 \rightarrow 0.5$) to have a predicate appeared in a report.

BENZENE Failure Analysis. We investigate the following four cases in which BENZENE fails to complete the RCA: 25, 26, 53, and 54. The main reason is unsuccessful (under-constrained) state mutations that trigger non-crashes like mutating multiple locations simultaneously (§10). By contrast, AURORA accomplishes an RCA for 25 by discovering desirable behaviors using a fuzzer (*i.e.*, AFL).

9.3. Correctness of BENZENE

RCA Report. BENZENE generates a summary report that contains a predicate-ranking based on the likelihood of a root cause such that a developer can manually review them. In the majority of cases, BENZENE outperforms AURORA in precisely predicting the root cause locations.

Comparison with AURORA. We evaluate 28 cases successful for both BENZENE and AURORA by inspecting how a predicted root cause (*i.e.*, highly ranked predicate) is close to a developer-provided patch. As listed in Table 3, BENZENE and AURORA report a rank per sample with an average of 3.32 and 10.42, respectively (the lower, the better). Notably, BENZENE demonstrates better rankings for 19 samples, the same for 4 samples, and marginal differences for 5 samples. For 19 cases, our finding shows that the AURORA's statistical ranking approach suffers from the false positive from a bug-unrelated predicate (*e.g.*, memory allocators). The five cases where AURORA beats BENZENE (*e.g.*, 18, 22, 23, 35, 49) are because there are non-crashing behaviors that are closer to an initial crash than desirable behaviors for RCA. Note that we elaborate on the discrepancies on a root cause predicate between BENZENE and AURORA in Appendix A.3.

9.4. Efficiency of BENZENE

Summary. To demonstrate the efficiency of BENZENE, we measure the elapsed time required for the entire process (*i.e.*, fuzzing, tracing, and RCA) and the memory footprint for identifying the root cause. Figure 9 shows that BENZENE is much faster than AURORA and ARCUS in most cases. Overall, BENZENE is 8.1× and 1.1× faster with up to 9.1× and 53.7× less memory consumption in the RCA phase than AURORA and ARCUS, respectively. Table 4 summarizes a breakdown of the duration to complete an RCA between approaches. As shown, BENZENE surpasses AURORA and

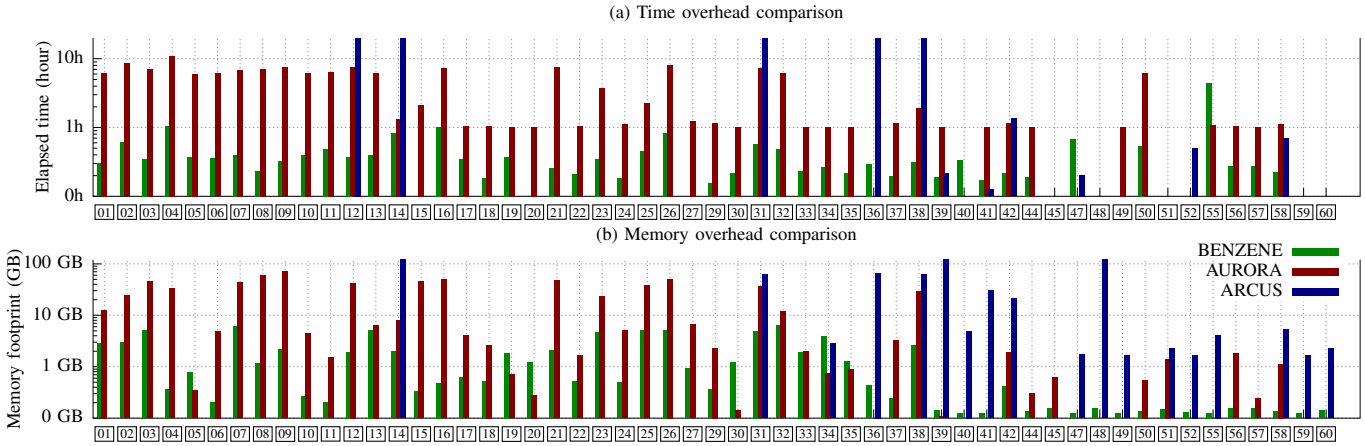

Figure 9: Comparison of performance overheads and memory footprints between BENZENE, AURORA and ARCUS in a log scale. For AURORA, BENZENE clearly demonstrates better performance in fuzzing (10.0× faster), which dominates the whole processing time. BENZENE uses only one-tenth of memory footprints compared to AURORA. On the other hand, BENZENE outperforms ARCUS with 1.1× faster in speed, but 53.7× lower memory footprints. For a memory comparison, we measure the overheads for the RCA phase alone owing to the distinctive structures of each approach. The five cases (28, 43, 46, 53, 54) are deliberately excluded because AURORA and ARCUS could not proceed. The missing elapsed time of the five samples (45, 48, 51, 59, 60) are < 5 minutes for BENZENE and ARCUS.

System	Time(sec)				Mem(MB)
	Pre+Mutation	Tracing	RCA	Total	
BENZENE	1,054	397	116	1,567	1,839
AURORA	10,643	1,621	497	12,762	16,858
Difference	10.0×	4.0×	4.2×	8.1×	9.1×
BENZENE	549	143	17	710	444
ARCUS	-	-	845	845	23,858
Difference	-	-	-	1.1×	53.7×

TABLE 4: Comparison of performance overheads with a phase breakdown: BENZENE vs. AURORA (46 cases) and BENZENE vs. ARCUS (14 cases) on average. Compared to AURORA, BENZENE clearly demonstrates its efficiency in fuzzing (8.1× faster), tracing, and RCA while reducing memory footprints (9.1× less). We exclude ARCUS's five failure cases (> 20 hours) for a fair comparison.

ARCUS in both speed and resource consumption by a large margin, well demonstrating the efficiency of BENZENE.

Comparison with AURORA. The mutation process dominates the entire processing time, wherein BENZENE is *10.0× faster* than AURORA on average. Additionally, the significant gap comes from tracing (*i.e.*, around 4.0× faster). In addition to the speed, BENZENE demonstrates the benefit of a memory footprint by consuming merely *one-nineth* of AURORA's (*i.e.*, 1.8 GB vs. 16.1 GB on average). Meanwhile, we investigate 6 exceptions: 05, 19, 20, 30, 34, and 39 where BENZENE consumes a larger memory footprint than AURORA. For 05, 19, 20, and 30, it mainly arises from the property of an ASAN-enabled executable, which inevitably requires heavy tracing due to added instructions during binary instrumentation. In the case of 34 and 39, the `exists` predicate type in BENZENE introduces an additional overhead as it collects all unique values in a certain operand up to a pre-defined threshold (*e.g.*, 1024).

Comparison with ARCUS. Since ARCUS does not have the mutation process and its tracing takes a negligible amount of time; we only consider an RCA phase for memory overheads. For a fair comparison, we consider 19 cases, excluding 15 cases that ARCUS failed to proceed (↛ in Table 3 and 13). For the five highly-structured cases (12, 14, 31, 36, and 38), ARCUS could not finish its analysis within the 20-hour time-limit. Based on an in-depth investigation, we confirm that the main reason arises from the complexity (*i.e.*, a massive number of constraints) that a symbolic execution engine populates, resulting in a huge overhead. Other than such cases, the average processing time for the RCA on BENZENE and ARCUS is comparable (*i.e.*, BENZENE is 1.1× faster than ARCUS). However, ARCUS exhibits a memory consumption around 53.7× more than BENZENE. Our investigation reveals that a massive memory requirement mainly comes from the ARCUS's RCA module for heap and stack overflow analyses.

10. Discussion and Limitations

Unsupported Bug Types. The current implementation of BENZENE lacks support for certain types of bugs that occur when a program returns a non-deterministic result for each run. This is because BENZENE relies on observing crashes, which reliably extracts predicates for RCA. For example, a multi-threaded executable may or may not crash because of a concurrency bug. In general, detecting bugs from the non-deterministic behavior of a program remains an open challenge for modern fuzzing systems [75]. Although this issue is orthogonal to RCA, we believe that a deterministic mutation approach (*e.g.*, thread-aware fuzzing) could be a promising solution, which we plan to investigate in our future work. Lastly, while BENZENE focuses primarily on security-related bugs, it has the potential to detect other types of bugs by observing behavioral differences.

Non-deterministic Nature of Mutation. BENZENE utilizes a mutation technique that incorporates randomness during the behavior exploration phase. As a result, even with the same crashing execution, the technique may produce slightly deviating results owing to the non-deterministic nature of

the mutations. This indicates that the ranking of the RCA may vary slightly for each run.

Predicate Types and Synthesis. Despite collecting a range of behaviors through mutations, BENZENE may not gain sufficient information for predicate synthesis. Additionally, the current set of predicate types in BENZENE may not adequately capture the complexity of a root cause.

Result Interpretation without Source Code. BENZENE offers a binary-only mode for RCA, which enables analysts (*e.g.*, security practitioners, vulnerability analysts) without access to the source code to diagnose problems. However, interpreting the BENZENE report may be challenging without the source code (*i.e.*, ground truth), and the report may not be entirely convincing. We recommend that BENZENE users leverage a decompiler, such as IDA [88] or Ghidra [63], to gain a more comprehensive understanding of the report.

Bug that Spans Multiple Locations. When dealing with a single bug that requires multiple predicates, BENZENE's performance may be restricted. This is particularly true in cases wherein fixing a complex or structural issue involves modifying multiple locations or an entire function. In such cases, changing a specific predicate may not generate a non-crashing sample because a single predicate cannot fully capture the root cause condition.

Dataset Representativeness. Our dataset is meticulously curated to include a diverse range of vulnerability types, such as a stack overflow, heap overflow, null dereference, type confusion, integer overflow, double free, etc. Furthermore, we emphasize that our experiments encompass all target samples from the two previous approaches (ARCUS [123] and AURORA [69]) to ensure a fair comparison.

Future Directions. To improve BENZENE, we suggest research directions of extracting a function and developing an advanced mutation. First, the current design of BENZENE relies on recognizing target functions based on the invalid memory access at a crashing site, which suffers from considerable performance overheads. One method to improve the speed of the function recognition process is to adopt a path profiling algorithm [67] instead of RR [107]. Second, we plan to develop an advanced mutation strategy that can handle complex bugs spanning multiple locations or variables. The current implementation of BENZENE cannot handle cases such as 25 and 26 in Table 3 because those require mutating multiple locations in a single mutation cycle. To tackle this problem, we intend to allow BENZENE to modify multiple variables based on user preferences.

11. Related work

Reverse Execution. Reverse execution enriches crash analysis by facilitating the analysis of the execution flow from a crash site [78], [79], [106], [120], [121]. It is common to construct a data flow on top of a reverse execution. POMP [121] proposes a backward taint analysis on both a crash dump and a control-flow trace, followed by locating a program statement that triggers a crash. POMP++ [106] advances POMP by leveraging a value-set analysis [66] that can refine data flow construction via the relation verification of a memory alias. RETRACER [79] extracts program semantics from a memory dump and performs a backward taint analysis to find a certain function on the stack that triggers a crash. REPT [78] presents an error correction mechanism to conduct both forward and backward analyses iteratively on a hardware-assisted execution trace and then restore the data flow pertaining to a crash. Similarly, BENZENE reconstructs a DFG for further RCA investigation.

Mutation-based Fuzzing. To improve the accuracy of RCA, Miller et al. [101] harness a mutation-based fuzzing technique for generating varying program inputs that trigger an unexplored behavior. Mutation-based fuzzing is largely categorized as either a symbolic-based [73], [85], [86], [108], [115] or taint-based [81], [83], [95], [111], [117] approach. Fuzzing leverages a symbolic execution into various tasks including constraint solving [73], [85], [86], program transformation [90], [108], and vulnerability detection [122], [123]. Meanwhile, a taint analysis assists fuzzing in finding interesting bytes with a mutation [83], [95], [111], data dependency inference [81], and value type [76]. However, BENZENE adopts a novel approach, the under-constrained state mutation for efficient RCA, because the above techniques incur high performance overheads.

Statistical Fault Localization. Statistical fault localization [118] assigns a score to each selected group for a program element (*e.g.*, statement, predicate) atop execution traces, and has been utilized as the key technique for RCA [65], [69], [91], [96], [97], [114], [127]. To improve statistical fault localization, it is crucial to create appropriate test cases by exploring as many distinct paths as possible within a target program. For instance, AURORA [69] generates both crashing and non-crashing inputs with an existing fuzzing technique (*i.e.*, AFL crash exploration mode), and so does VULNLOC [114] with directed fuzzing. However, such approaches suffer from generating a valid test case that requires a highly-structured syntax. KAIRUX [127] introduces a technique that pinpoints the location of a root cause with dynamic slicing when a sufficient number of unit tests and crashing inputs are provided. However, BENZENE solely requires a target program with a crashing input to generate varying (valid) test cases.

12. Conclusion

We present BENZENE, a practical system for RCA that can automatically identify problematic locations in a target program when given a crash-inducing input. In contrast to previous approaches, BENZENE can quickly identify the root cause through novel techniques based on under-constrained mutations. The system consists of three main components: dynamic binary analysis, program behavior exploration, and RCA. Our evaluation of 60 real-world applications demonstrates the effectiveness and efficiency of BENZENE. Compared with state-of-the-art approaches, BENZENE accurately pinpoints the root cause location in 93.3% of samples. BENZENE is 4.6× faster and requires 31.4× less memory on average than prior approaches.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their helpful feedback. This work was supported by the Basic Science Research Program through NRF grant funded by the Ministry of Education of the Government of South Korea (No. 2022R1F1A1074373), and three Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government (MSIT) (No. 2021-0-00624; Development of Intelligence Cyber Attack and Defense Analysis Framework for Increasing Security Level of C-ITS, No. 2022-0-01199; Graduate School of Convergence Security (Sungkyunkwan University), No.2022-0-00688; AI Platform to Fully Adapt and Reflect Privacy-Policy Changes). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the sponsor's views.

References

[1] "ABC2MTEX 1.6.1: command line stack overflow," https://www.exploit-db.com/exploits/47254.

[2] "ABC2MTEX 1.6.1: process ABC key field buffer overflow (CVE-2004-1257)," https://www.exploit-db.com/exploits/25018.

[3] "Autotrace: buffer overflow vulnerability (CVE-2017-9167)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2017-9167.

[4] "Autotrace: integer overflow vulnerability (CVE-2017-9186)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2017-9186.

[5] "Autotrace: use-after-free vulnerability (CVE-2017-9182)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2017-9182.

[6] "Bash: integer overflow," https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00042.html.

[7] "Dmitry: stack buffer overflow (CVE-2017-7938)," https://www.exploit-db.com/exploits/41898.

[8] "Gif2png: stack overflow vulnerability (CVE-2009-5018)," https://bugs.gentoo.org/346501.

[9] "GNU InetUtils 1.8-1: FTP client heap overflow," https://www.exploit-db.com/exploits/15705.

[10] "GNU: patch heap buffer overflow," https://savannah.gnu.org/bugs/?func=detailitem&item_id=54558.

[11] "GraphicsMagick: use-after-free in CloseBlob (CVE-2017-14103)," https://blogs.gentoo.org/ago/2017/09/01/graphicsmagick-use-after-free-in-closeblob-blob-c-incomplete-fix-for-cve-2017-11403.

[12] "GraphicsMagick: use-after-free in ReadMNGImage (CVE-2017-11403)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2017-11403.

[13] "Jpegtoavi: 1.5 file list buffer overflow (CVE-2004-1279)," https://www.exploit-db.com/exploits/24981.

[14] "Libbfd: global-buffer-overflow in objcopy," https://sourceware.org/bugzilla/show_bug.cgi?id=21412.

[15] "Libexif: security bug in exif_data_load_data_entry (CVE-2007-2645)," https://sourceforge.net/p/libexif/bugs/70/.

[16] "Libical: type confusion in icaltimezone_get_vtimezone_properties function in icalproperty.c," https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2019-11706.

[17] "Libiec61850 1.3: stack based buffer overflow (CVE-2018-18957)," https://www.exploit-db.com/exploits/45798.

[18] "Libpng: buffer overflow vulnerability (CVE-2004-0597)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2004-0597.

[19] "Libtiff: integer overflow in the TIFFFetchData (CVE-2006-2025)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2006-2025.

[20] "Libzip: use after free in _zip_buffer_free (CVE-2017-12858)," https://blogs.gentoo.org/ago/2017/09/01/libzip-use-after-free-in-_zip_buffer_free-zip_buffer-c/.

[21] "Lua: use after free in lua_upvaluejoin in lapi.c (CVE-2019-6706," https://security-tracker.debian.org/tracker/CVE-2019-6706.

[22] "MiniFtp: 'parseconf_load_setting' buffer overflow," https://www.exploit-db.com/exploits/46807.

[23] "Mruby: heap buffer overflow in OP_ENTER (CVE-2018-12248)," https://github.com/mruby/mruby/issues/4038.

[24] "Mruby: integer overflow in OP_GET_UPVAR (CVE-2018-10191)," https://github.com/mruby/mruby/issues/3995.

[25] "Mruby: NULL pointer dereference in mrb_vm_exec() (CVE-2021-46020)," https://github.com/mruby/mruby/issues/5613.

[26] "Mruby: out of bounds read (CVE-2022-0525)," https://huntr.dev/bounties/e19e109f-acf0-4048-8ee8-1b10a870f1e9/.

[27] "Mruby: type confusion in mrb_exc_set," https://hackerone.com/reports/185041.

[28] "Mruby: uninitialized variable leak," https://github.com/mruby/mruby/issues/3947.

[29] "Mruby: use after free in Array to_h," https://hackerone.com/reports/181321.

[30] "Mruby: use after free in File initialize_copy (CVE-2018-10199)," https://github.com/mruby/mruby/issues/4001.

[31] "Nasm: NULL pointer dereference in asm/labels.c (CVE-2018-16517)," https://bugzilla.nasm.us/show_bug.cgi?id=3392513.

[32] "Nasm: use after free in paste_tokens (CVE-2019-8343)," https://bugzilla.nasm.us/show_bug.cgi?id=3392556.

[33] "NM: stack buffer overflow," https://sourceware.org/bugzilla/show_bug.cgi?id=21670.

[34] "Ntpq: Stack-based buffer overflow (CVE-2018-12327)," https://www.exploit-db.com/exploits/44909.

[35] "O3read: buffer overflow vulnerability (CVE-2004-1288)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2004-1288.

[36] "Objdump: heap buffer overflow in disassemble_bytes (CVE-2017-9746)," https://sourceware.org/bugzilla/show_bug.cgi?id=21580.

[37] "Pdfresurrect: stack overflow vulnerability in pdf_load_xrefs (CVE-2019-14267)," https://vulmon.com/exploitdetails?qidtp=exploitdb&qid=47178.

[38] "Perl: heap buffer overflow in match_uniprop," https://github.com/Perl/perl5/issues/17384.

[39] "PHP: buffer overwrite in finfo_open with malformed magic file (CVE-2015-8865)," https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2015-8865.

[40] "PHP: integer overflow in phar_parse_tarfile() (CVE-2012-2386)," https://security-tracker.debian.org/tracker/CVE-2012-2386.

[41] "PHP: out of bounds write on heap (CVE-2019-6977)," https://bugs.php.net/bug.php?id=77270.

[42] "PHP: type confusion in imagecrop() (CVE-2013-7226)," https://hackerone.com/reports/1356.

[43] "PHP: type confusion in unserialize() with DateTimeZone (CVE-2015-0273)," https://github.com/80vul/phpcodz/blob/master/research/pch-019.md.

[44] "PHP: uninitialized variable leak (CVE-2019-11038)," https://bugs.php.net/bug.php?id=77973.

[45] "Poppler: beap buffer overflow in JPXStream," https://gitlab.freedesktop.org/poppler/poppler/-/issues/768.

[46] "Poppler: heap buffer overflow in XRef::getEntry due to integer overflow," https://gitlab.freedesktop.org/poppler/poppler/-/issues/717.

[47] "Python: heap overflow in zipimporter module (CVE-2016-5636)," https://bugs.python.org/issue26171.

[48] "Python: type confusion in partial.setstate, partial_repr, and partial_-call," https://hackerone.com/reports/116286.

[49] "Readelf: heap buffer overflow in process_mips_specific (CVE-2019-9077)," https://sourceware.org/bugzilla/show_bug.cgi?id=24243.

[50] "Redis-cli < 5.0: buffer overflow (CVE-2018-12326)," https://www.exploit-db.com/exploits/44904.

[51] "Sleuthkit: double free in fls," https://github.com/sleuthkit/sleuthkit/issues/905.

[52] "Sound eXchange (SoX) 14.4.2: multiple vulnerabilities," https://www.exploit-db.com/exploits/42398.

[53] "Sound eXchange (SoX): stack buffer overflow in fft4g.c," https://sourceforge.net/p/sox/bugs/321/.

[54] "SQLite3: division by zero in the query planner," https://www.sqlite.org/src/info/e4598ecbdd18bd82.

[55] "SQLite3: stack overflow in sqlite3_str_vappendf, caused by int overflow ," https://www.sqlite.org/src/info/23439ea582241138.

[56] "Sudo: format string vulnerability (CVE-2012-0809)," https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2012-0809.

[57] "Tcpdump: heap overread (CVE-2017-16808)," https://github.com/the-tcpdump-group/tcpdump/issues/645.

[58] "TinyCC: out of bounds write in sym_pop," https://lists.nongnu.org/archive/html/tinycc-devel/2018-12/msg00014.html.

[59] "Typespeed 0.4.1: local format string (CVE-2005-0105)," https://vulmon.com/exploitdetails?qidtp=exploitdb&qid=25106.

[60] "Xpdf: use of uninitialized variable," https://forum.xpdfreader.com/viewtopic.php?f=3&t=41890.

[61] (2022) scikit-learn, Machine Learning in Python. [Online]. Available: https://github.com/scikit-learn/scikit-learn

[62] (2022) SQLite3. [Online]. Available: https://www.sqlite.org/index.html

[63] N. S. Agency, "Ghidra Software Reverse Engineering Framework," https://ghidra-sre.org/, 2019.

[64] H. Al Salem and J. Song, "A review on grammar-based fuzzing techniques," *International Journal of Computer Science & Security (IJCSS)*, vol. 13, no. 3, pp. 114–123, 2019.

[65] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, London, UK, Jul. 2007.

[66] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[67] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 46–57.

[68] G. Birch, B. Fischer, and M. Poppleton, "Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs," *Software & Systems Modeling*, vol. 18, no. 1, pp. 445–471, 2019.

[69] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "AURORA: Statistical Crash Analysis for Automated Root Cause Explanation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2020.

[70] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[71] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 122–131.

[72] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," 2004.

[73] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[74] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 121–130.

[75] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2020.

[76] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[77] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, USA, Aug. 2019.

[78] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "REPT: Reverse Debugging of Failures in Deployed Software," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, GA, Oct. 2018.

[79] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "Retracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May 2016.

[80] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap taichi: exploiting memory allocation granularity in heap-spraying attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 327–336.

[81] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2020.

[82] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[83] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 474–484.

[84] GitHub, "Public fuzzers," https://github.com/search?q=fuzzing&type=Repositories/, 2022.

[85] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[86] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[87] Google, "OSS-Fuzz: Continuous Fuzzing for Open Source Software," https://github.com/google/oss-fuzz, 2022.

[88] I. Guilfanov, "IDA Pro - Hex Rays," https://www.hex-rays.com/products/ida/, 2018.

[89] Intel, "Intel processor trace," https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[90] L. Intel, "Circumventing fuzzing roadblocks with compiler transformations," 2016.

[91] W. Jin and A. Orso, "F3: Fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 213–223.

[92] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016.

[93] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021.

[94] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 121–132.

[95] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[96] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[97] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[98] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40. ACM, 2005, pp. 190–200.

[99] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[100] V. J. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1024–1036.

[101] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[102] Mozilla Security, "Browser fuzzing at mozilla," https://hacks.mozilla.org/2021/02/browser-fuzzing-at-mozilla/, 2022.

[103] ——, "A collection of fuzzers in a harness for testing the spidermonkey javascript engine," https://github.com/MozillaSecurity/funfuzz, 2022.

[104] ——, "A cross-platform browser fuzzing framework," https://github.com/MozillaSecurity/grizzly, 2022.

[105] ——, "Generation-based, context-free grammar fuzzer," https://github.com/MozillaSecurity/dharma, 2022.

[106] D. Mu, Y. Du, J. Xu, J. Xu, X. Xing, B. Mao, and P. Liu, "Pomp++: Facilitating postmortem program diagnosis with value-set analysis," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1929–1942, 2019.

[107] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 377–389.

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

[109] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[110] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "Nozzle: A defense against heap-spraying code injection attacks." in *Proceedings of the 18th USENIX Security Symposium (Security)*, Montreal, Canada, Aug. 2009.

[111] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[112] M. Research, "Project onefuzz," https://www.microsoft.com/en-us/research/project/project-onefuzz/, 2022.

[113] K. Serebryany, "Continuous Fuzzing with libFuzzer and AddressSanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 157–157.

[114] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *Proceedings of the 16th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Virtual, Jun. 2021.

[115] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[116] The Chromium Projects, "Google chromium security," https://www.chromium.org/Home/chromium-security/bugs/, 2022.

[117] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[118] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[119] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the International Conference on Software Engineering*, 2022.

[120] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards locating a memory corruption vulnerability with your core dump," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[121] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[122] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, "Automated bug hunting with data-driven symbolic root cause analysis," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual, Nov. 2021.

[123] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "Arcus: Symbolic root cause analysis of exploits in production systems," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[124] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[125] M. Zalewski. (2022) American Fuzzy Lop. [Online]. Available: https://github.com/google/AFL

[126] K. Zetter, "A famed hacker is grading thousands of programs—and may revolutionize software in the process," 2016.

[127] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan, "The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 131–146.

[128] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

Appendix A.
In-depth Analysis and Case Studies

A.1. AURORA Failure Analysis

With a careful analysis, we categorize three main reasons of failure cases by AURORA as followings.

- **Fuzzing Failure.** The AFL's crash exploration mode could not find proper non-crashing behaviors for RCA within 6-hour time-limit, including 02 PHP-6977, 04 Poppler-12293, 06 TinyCC-20375, and 26 Python-116286.
- **Insufficient Non-crashing Behaviors.** A majority of samples fall into this category where non-crashing behaviors are unsatisfactory to discover a root cause candidate (*i.e.*, within Top 50), including 01 PHP-8865 (2920), 03 mruby-0525 (56), 05 SoX-11358 (3888), 07 mruby-46020 (270), 08 PHP-7226 (289), 09 PHP-0273 (124), 10 libical-11706 (485), 11 SoX-8356 (264), 13 PHP-2386 (2175), 15 SQLite-16168 (606), 29 Xpdf (237), and 30 nm-21670 (365). Note that the numbers in a parenthesis represent the predicate ranking of the root cause in a report.
- **Unsupported Predicate Type.** We found one sample that fails RCA due to the lack of a predicate type: 50 abc2mtex-1257 (`strlen_geq`).

A.2. ARCUS Failure Analysis

With a careful analysis, we categorize six main reasons of failure cases for ARCUS as followings.

- **Symbolic Execution Failure.** A symbolic execution engine ran over the 20-hour time-limit, which includes 12 mruby-181321 (36.3%), 14 Poppler-7310 (<0.01%), 31 mruby-10199 (1.11%), 36 Python-5636 (0.75%), and 38 mruby-10191 (19.8%) Note that the values in a parenthesis represent the progress (*i.e.*, $\frac{cur\_trace\_cnt}{total\_trace\_cnt} \times 100$) of each reconstruction analysis after 20 hours.
- **Crash in a Hook.** The samples in this category encounter a crash while hooking a function by `angr` because of trace discrepancies between symbolic executions and PT traces, including 56 libtiff-2025 and 57 libexif-2645.
- **Unsupported External Libraries in a Symbolic Engine.** Because the symbolic execution engine in `angr` does not fully support external libraries like `libc`, an execution within a library (*i.e.*, following recorded Intel PT-based traces) often leads an abort during a state reconstruction step. The samples that fall into this category include 32 Lua-6706 (`angr` reaches an unmapped region), 33 nasm-8343 (`localtime()`), 37 Bash (`gethostname()`), and 46 dmitry-7938 (`gethostbyname()`).
- **Constraint Solving-Failure.** We found one sample that fails to solve a complex constraint in a symbolic engine: 13 PHP-2386.
- **Inadequate Rule.** The RCA failure of the following two samples arises from missing rules to discover a root cause, which includes 34 Sleuthkit and 42 autotrace-9167.
- **Failure in Resolving Dynamic Symbols.** We found the cases that fail to resolve the function address from an external library during symbolic execution, including 16 SQLite-13434, 35 libzip-12858, 43 Redis-12326, 44 ftp-15705, 49 pdf-re-14267. 53 GM-11403, 54 GM-14103, and 55 autotrace-9182.

A.3. Root Cause Predicate Discrepancies

Both BENZENE and AURORA produce a ranking report with the candidate of a root cause predicate. In case of even successful RCA with both approaches, we recognize the discrepancies of root cause predicates. However, we manually inspect that those root cause indicates semantically fruitful information that can assist a developer for a bug fix. For instance, in the case of 38 mruby-10191, both BENZENE and AURORA pinpoint the predicate of `[eax >= 0xfc]` and `[min rdx >= 0xfd]`, respectively. Similarly, both report `[¬(esi <= 0x4de5fadb)]` and `[max r12 >= 0x80000011]`, which indicates a large value when the root cause of 14 Poppler-7310 is an integer overflow. We hypothesize that such discrepancies predominately arise from the reliance on different (nondeterministic) mutation processes, approaches (*e.g.*, state mutation, predicate type), and implementations.

A.4. Case Studies

In this section, we deal with three case studies: two cases that BENZENE can successfully pinpoint a root cause, and one case (03 mruby-0525) that BENZENE is incapable of.

CVE-2016-5636. Figure 10 presents a heap overflow vulnerability in `get_data()`, at the `zipimporter` module from python 3.6. In a nutshell, the four-byte `bytes_size` variable becomes `0x0` in line 7 due to an overflow when `data_size` is incremented by one from `0xffffffff`. As a consequence, `get_data()` allocates a small amount of memory based on the `bytes_size` in line 13, whereas the actual size of the data is bigger than the allocation, resulting in an out-of-bound write. The vulnerability has been patched by checking if `data_size` exceeds a limit value. In this case, BENZENE successfully pinpoints the root cause at the 5th predicate. BENZENE targets varying functions for fuzzing, including the culprit function, collecting a non-crashing behavior (*e.g.*, `data_size` $\neq$ `0xffffffff`), followed by computing a distance score based on PCA.

CVE-2013-7226. Figure 11 displays a type confusion vulnerability in `imagecrop()` from PHP 5.5.8. `zval` type in line 6 represents an internal structure, which handles varying variable types (*e.g.*, `IS_LONG` defines an integer value). Here, the absence of a type check on `zval` in Line 11 triggers the vulnerability. To exemplify, a carefully crafted input can have the `tmp` points to a string type (*i.e.*, `IS_STRING`) `zval`, while it is assumed to be an integer type (*i.e.*, `IS_LONG`). It subsequently makes `rect.x` wrongly contains a pointer, which is a large value as an integer. Since `gdImageCrop()` in Line 15 calls `memcpy()` with the `rect.x` as a source pointer, a segmentation fault occurs due to a buffer over-read. The

```
1  // Modules/zipimport.c:1061
2  static PyObject *
3  get_data (PyObject *archive, PyObject *toc_entry)
4  {
5      ...
6      // bytes_size: data_size(0xffffffff) + 1 -> overflow.
7      bytes_size = compress == 0 ? data_size : data_size + 1;
8      if (bytes_size == 0)
9          // bytes_size: 0x1
10         bytes_size++;
11
12     // raw_data: a buffer with size of bytes_size (0x1).
13     raw_data = PyBytes_FromStringAndSize(NULL, bytes_size);
14     ...
15
16     // buf points at 1-size buffer
17     buf = PyBytes_AsString(raw_data);
18     if (err == 0) {
19         // buf is 1-size but data_size is 0xffffffff.
20         // resulting in the Out-of-Bound write
21         bytes_read = fread(buf, 1, data_size, fp);
22     }
23 }
```

Figure 10: Code snippet for CVE-2016-5636. An out-of-bounds write occurs (Line 21) due to the insufficient allocation size overflowed by `data_size + 1` (Line 7). We simplified the code for brevity.

```
1  // ext/gd.c:4941
2  PHP_FUNCTION(imagecrop)
3  {
4      ...
5      gdRect rect;
6      zval **tmp;
7      if (zend_hash_find(HASH_OF(z_rect),
8              "x", sizeof("x"), (void **)&tmp) != FAILURE) {
9          // tmp points IS_STRING type zval,
10         // resulting in a type confusion bug
11         rect.x = Z_LVAL_PP(tmp);
12     }
13     ...
14     // Out-of-bound access occurs due to rect.x.
15     im_crop = gdImageCrop(im, &rect);
16 }
```

Figure 11: Code snippet for CVE-2013-7226. A type confusion vulnerability in line 11 makes an integer variable `rect.x` contain a pointer value, causing a segmentation fault during `gdImageCrop()` in Line 15.

vulnerability has been patched by adding a type checking routine in Line 11 (*e.g.*, ensuring that `zval` contains `IS_LONG`). In this case, BENZENE redirects `tmp` to point to `zval` of an integer value through a mutation with BoV, obtaining a (desirable) non-crashing behavior. Notably, a pure-random mutation hardly discovers such a non-crash because `tmp` must point to the valid `zval` structure.

CVE-2022-0525. Here we introduce a case that BENZENE cannot handle, which is a heap overflow vulnerability in `gen_assignment()` from `mruby` version 3.0.0. The vulnerability occurs when the `mruby` compiler mishandles an internal stack position (`cur->sp`) of interpreter while generating array-related bytecodes. Due to its complexity, it requires two consequent patches for this bug, which contains multiple root causes. Although BENZENE fails to pinpoint the locations of the original patches, it provides a meaningful insight about a given crash. With an under-constrained state mutation, BENZENE is able to find one of non-crashing behaviors from the target function (`gen_hash()`) by modifying a stack-relevant variable (`cur->sp`). Although it differs from the patched locations, this implies that a stack position value in `mruby` has a possible root cause of the crash.

Target	CVE/Issue	Bug Type	# of Instructions
01 PHP-8865	CVE-2015-8865 [39]	Heap Overflow	9,809,095
02 PHP-6977	CVE-2019-6977 [41]	Heap Overflow	12,491,703
03 mruby-0525	CVE-2022-0525 [26]	Heap Overflow	3,134,273
04 Poppler-12293	CVE-2019-12293 [45]	Heap Overflow	125,813,540
05 SoX-11358	CVE-2017-11358 [52]	Heap Overflow	680,855
06 TinyCC-20375	CVE-2018-20375 [58]	Heap Overflow	614,608
07 mruby-46020	CVE-2021-46020 [25]	Null Dereference	2,243,674
08 PHP-7226	CVE-2013-7226 [42]	Type Confusion	9,956,549
09 PHP-0273	CVE-2015-0273 [43]	Type Confusion	9,516,412
10 libical-11706	CVE-2019-11706 [16]	Type Confusion	3,975,381
11 SoX-8356	CVE-2019-8356 [53]	Stack Overflow	141,771,429
12 mruby-181321	hackerone-181321 [29]	Use After Free	13,497,624
13 PHP-2386	CVE-2012-2386 [40]	Integer Overflow	9,285,250
14 Poppler-7310	CVE-2019-7310 [46]	Integer Overflow	16,506,068
15 SQLite-13434	CVE-2020-13434 [54]	Integer Overflow	1,331,697
16 SQLite-16168	CVE-2019-16168 [55]	Division by Zero	1,338,307
17 libbfd-8393	CVE-2017-8393 [14]	Global Overflow	539,511
18 readelf-9077	CVE-2019-9077 [49]	Heap Overflow	671,011
19 objdump-9746	CVE-2017-9746 [36]	Heap Overflow	6,411,352
20 tcpdump-16808	CVE-2017-16808 [57]	Heap Overflow	1,138,487
21 perl-17384	Issue-17384 [38]	Heap Overflow	1,314,300
22 patch-54558	Issue-54558 [10]	Heap Overflow	317,534
23 mruby-12248	CVE-2018-12248 [23]	Heap Overflow	17,393,851
24 nasm-16517	CVE-2018-16517 [31]	Null Dereference	955,041
25 mruby-185041	hackerone-185041 [27]	Type Confusion	13,420,945
26 Python-116286	hackerone-116286 [48]	Type Confusion	44,778,829
27 mruby-3947	Issue-3947 [28]	Uninitialized Var.	25,646,156
28 PHP-11038	CVE-2019-11038 [44]	Uninitialized Var.	11,683,900
29 Xpdf	N/A [60]	Uninitialized Var.	7,245,661
30 nm-21670	Issue-21670 [33]	Stack Overflow	515,784
31 mruby-10199	CVE-2018-10199 [30]	Use After Free	17,191,784
32 Lua-6706	CVE-2019-6706 [21]	Use After Free	960,676
33 nasm-8343	CVE-2019-8343 [32]	Use After Free	962,302
34 Sleuthkit	N/A [51]	Double Free	2,463,131
35 libzip-12858	CVE-2017-12858 [20]	Double Free	352,871
36 Python-5636	CVE-2016-5636 [47]	Integer Overflow	105,414,833
37 bash	N/A [6]	Integer Overflow	1,423,765
38 mruby-10191	CVE-2018-10191 [24]	Integer Overflow	28,965,409
39 libpng-0597	CVE-2004-0597 [18]	Heap Overflow	266,061
40 jpegtoavi-1279	CVE-2004-1279 [13]	Heap Overflow	332,757
41 o3read-1288	CVE-2004-1288 [35]	Heap Overflow	258,545
42 autotrace-9167	CVE-2017-9167 [3]	Heap Overflow	1,239,089
43 Redis-12326	CVE-2018-12326 [50]	Heap Overflow	1,164,556
44 ftp-15705	EDB-15705 [9]	Heap Overflow	478,490
45 gif2png-5018	CVE-2009-5018 [8]	Stack Overflow	308,479
46 dmitry-7938	CVE-2017-7938 [7]	Stack Overflow	537,600
47 ntpq-12327	CVE-2018-12327 [34]	Stack Overflow	1,195,848
48 libiec-18957	CVE-2018-18957 [17]	Stack Overflow	237,628
49 pdf-re-14267	CVE-2019-14267 [37]	Stack Overflow	1,888,476
50 abc2mtex-1257	CVE-2004-1257 [2]	Stack Overflow	266,245
51 abc2mtex-47254	EDB-47254 [1]	Stack Overflow	233,802
52 MiniFtp-46807	EDB-46807 [22]	Stack Overflow	274,528
53 GM-11403	CVE-2017-11403 [12]	Use After Free	1,289,365
54 GM-14103	CVE-2017-14103 [11]	Use After Free	7,534,186
55 autotrace-9182	CVE-2017-9182 [5]	Use After Free	190,298,343
56 libtiff-2025	CVE-2006-2025 [19]	Integer Overflow	462,148
57 libexif-2645	CVE-2007-2645 [15]	Integer Overflow	289,887
58 autotrace-9186	CVE-2017-9186 [4]	Integer Overflow	1,241,386
59 typespeed-0105	CVE-2005-0105 [59]	Format String	474,589
60 sudo-0809	CVE-2012-0809 [56]	Format String	1,005,898

TABLE 5: Evaluation dataset. We newly add 17 samples (01–17), employ 21 samples (18–38) from AURORA, and 22 samples (39–60) from ARCUS.

A.5. Evaluation Dataset

Table 5 presents our evaluation dataset in detail, including a CVE or issue number, bug type, and the number of exercised instructions to reach a vulnerability.

Appendix B.
Meta-Review

B.1. Summary

This paper presents BENZENE, a novel approach for conducting root cause analysis for crashes that are caused by vulnerabilities. BENZENE is built upon a new technique, underconstrained state mutation, that is able to generate both crashing and non-crashing behavior to assist with root cause analysis. The authors evaluated BENZENE with 60 vulnerabilities showing that the root cause can be identified for 93.3% of samples and that BENZENE is superior to existing state-of-the-art approaches with regards to speed and memory footprint.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) Finding non-crashing behavior close to crashing behavior is a fundamental challenge to applying statistical root cause analysis in the real world. This paper mitigates this long-known issue by proposing the state mutation technique.
2) BENZENE is a valuable step forward in an established field, namely root cause analysis. BENZENE is based on a novel technique, underconstrained state mutation, to generate crashing and non-crashing behaviors which are used to determine the root cause for program failure.
3) The paper creates a new tool to enable future science. The authors have made BENZENE open-source to promote the use of automated root cause analysis in the future.