AURORA: Statistical Crash Analysis for Automated Root Cause Explanation

Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz, *Ruhr-Universität Bochum*

https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

AURORA: Statistical Crash Analysis for Automated Root Cause Explanation

Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi,
Joel Frank, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum, Germany

Abstract

Given the huge success of automated software testing techniques, a large amount of crashes is found in practice. Identifying the root cause of a crash is a time-intensive endeavor, causing a disproportion between finding a crash and fixing the underlying software fault. To address this problem, various approaches have been proposed that rely on techniques such as reverse execution and backward taint analysis. Still, these techniques are either limited to certain fault types or provide an analyst with assembly instructions, but no context information or explanation of the underlying fault.

In this paper, we propose an automated analysis approach that does not only identify the root cause of a given crashing input for a binary executable, but also provides the analyst with context information on the erroneous behavior that characterizes crashing inputs. Starting with a single crashing input, we generate a diverse set of similar inputs that either also crash the program or induce benign behavior. We then trace the program's states while executing each found input and generate predicates, i. e., simple Boolean expressions that capture behavioral differences between crashing and non-crashing inputs. A statistical analysis of all predicates allows us to identify the predicate pinpointing the root cause, thereby not only revealing the location of the root cause, but also providing an analyst with an explanation of the misbehavior a crash exhibits at this location. We implement our approach in a tool called AURORA and evaluate it on 25 diverse software faults. Our evaluation shows that AURORA is able to uncover root causes even for complex bugs. For example, it succeeded in cases where many millions of instructions were executed between developer fix and crashing location. In contrast to existing approaches, AURORA is also able to handle bugs with no data dependency between root cause and crash, such as type confusion bugs.

1 Introduction

Fuzz testing (short: *fuzzing*) is a powerful software testing technique that, especially in recent years, gained a lot of traction both in industry and academia [28, 29, 31, 47, 49, 53, 59]. In essence, fuzzing capitalizes on a high throughput of inputs that are successively modified to uncover different paths within a target program. The recent focus on new fuzzing methods has produced a myriad of crashes for software systems, sometimes overwhelming the developers who are tasked with fixing them [37, 50]. In many cases, finding a new crashing input has become the easy and fully automated part, while triaging crashes remains a manual, labor-intensive effort. This effort is mostly spent on identifying the actual origin of a crash [58]. The situation is worsened as fuzzing campaigns often result in a large number of crashing inputs, even if only one actual bug is found: a fuzzer can identify multiple paths to a crash, while the fault is always the same. Thus, an analyst has to investigate an inflated number of potential bugs. Consequently, developers lose time on known bugs that could be spent on fixing others.

To reduce the influx of crashes mapping to the same bug, analysts attempt to *bucket* such inputs. Informally speaking, bucketing groups crashing inputs according to some metric—often coverage or hashes of the call stack—into equivalence classes. Typically, it is assumed that analyzing one input from each class is sufficient. However, recent experiments have shown that common bucketing schemes produce far too many buckets and, even worse, cluster distinct bugs into the same bucket [42]. Even if there are only a few inputs to investigate, an analyst still faces another challenge: Understanding the reasons why a given input leads to a crash. Often, the real cause of a crash—referred to as *root cause*—is not located at the point the program crashes; instead, it might be far earlier in the program's execution flow. Therefore, an analyst needs to analyze the path from the crashing location backward to find the root cause, which requires significant effort.

Consider, for example, a type confusion bug: a pointer to an object of type `A` is used in a place where a pointer to `B` is expected. If a field of `B` is accessed, an invalid access on a subsection of `A` can result. If the structures are not compatible (e. g., `A` contains a string where a pointer is expected by `B`), this can cause memory corruption. In this case, the crashing

location is most likely not the root cause of the fault, as the invariant "points to an instance of B" is violated in a different spot. The code that creates the object of type A is also most likely correct. Instead, the particular control flow that makes a value from type A end up in B's place is at fault.

In a naive approach, an analyst could inspect stack and register values with a debugger. Starting from the crash, they can manually backtrace the execution to the root cause. Using state-of-the-art sanitizers such as the ASAN family [51] may detect illegal memory accesses closer to the root cause. In our example, the manual analysis would start at the crashing location, while ASAN would detect the location where the memory corruption occurred. Still, the analyst has to manually recognize the type confusion as the root cause—a complicated task since most code involved is behaving correctly.

More involved approaches such as POMP [57], RETRACER [33], REPT [32] and DEEPVSA [38] use automated reverse execution and backward taint analysis. These are particularly useful if the crash is not reproducible. For example, REPT and RETRACER can analyze crashes that occurred on end-devices by combining core dumps and Intel PT traces. However, these approaches generally do not allow to automatically identify the root cause unless there is a direct data dependency connecting root cause and crashing instruction. Furthermore, REPT and RETRACER focus on providing an interactive debugging session for an analyst to inspect manually what happened before the crash.

In cases such as the type confusion above, or when debugging JIT-based software such as JavaScript engines, a single crashing input may not allow identifying the root cause without extensive manual reasoning. Therefore, one can use a fuzzer to perform *crash exploration*. In this mode, the fuzzer is seeded with crashing inputs which it mutates as long as they keep crashing the target application. This process generates new inputs that are related to the original crashing input, yet slightly different (e. g., they could trigger the crash via a different path). A diverse set of crashing inputs that mostly trigger the same bug can aid analysis. Observing multiple ranges of values and different control-flow edges taken can help narrow down potential root causes. However, none of the aforementioned methods takes advantage of this information. Consequently, identifying the root cause remains a challenging task, especially if there is no direct data dependency between root cause and crashing instruction. Although techniques such as ASAN, POMP, REPT and RETRACER provide more context, they often fail to identify the root cause and provide no explanation of the fault.

In this paper, we address this problem by developing an automated approach capable of finding the root cause given a crashing input. This significantly reduces human effort: unlike the approaches discussed previously, we do not only identify a code location, but also an explanation of the problem. This also reduces the number of locations an analyst has to inspect, as AURORA only considers instructions with a plausible explanation.

To enable precise identification of the root cause, we first pick one crashing input and produce a diverse set of similar inputs, some of which cause a crash while others do not. We then execute these newly-generated inputs while tracking the binary program's internal state. This includes control-flow information and relevant register values for each instruction. Given such detailed traces for many different inputs, we create a set of simple Boolean expressions (around 1,000 per instruction) to predict whether the input causes a crash. Intuitively, these predicates capture interesting runtime behavior such as whether a specific branch is taken or whether a register contains a suspiciously small value.

Consider our previous type confusion example and assume that a pointer to the constructor is called at some location in the program. Using the tracked information obtained from the diversified set of inputs, we can observe that (nearly) all calls in crashing inputs invoke the constructor of type A, while calls to the constructor of B imply that the input is not going to cause a crash. Thus, we can pinpoint the problem at an earlier point of the execution, even when no data taint connection exists between crashing location and root cause. This example also demonstrates that our approach needs to evaluate a large set of predicates, since many factors have to be captured, including different program contexts and vulnerability types. Using the predicates as a metric for each instruction, we can automatically pinpoint the possible root cause of crashes. Additionally, the predicates provide a concrete explanation of *why* the software fault occurs.

We built a prototype implementation of our approach in a tool called AURORA. To evaluate AURORA, we analyze 25 targets that cover a diverse set of vulnerability classes, including five use-after-free vulnerabilities, ten heap buffer overflows and two type confusion vulnerabilities that previous work fails to account for. We show that AURORA reliably allows identifying the root cause even for complex binaries. For example, we analyzed a type confusion bug in `mruby` where an exception handler fails to raise a proper exception type. It took an expert multiple days to identify the actual fault. Using our technique, the root cause was pinpointed automatically.

In summary, our key contributions are threefold:

- We present the design of AURORA, a generic approach to automatically pinpoint the location of the root cause and provide a semantic explanation of the crash.

- We propose a method to synthesize domain-specific predicates for binary programs, tailored to the observed behavior of the program. These predicates allow accurate predictions on whether a given input will crash or not.

- We implement a prototype of AURORA and demonstrate that it can automatically and precisely identify the root cause for a diverse set of 25 software faults.

To foster research on this topic, we release the implementation of AURORA at https://github.com/RUB-SysSec/aurora.

2 Challenges in Root Cause Analysis

Despite various proposed techniques, root cause identification and explanation are still complex problems. Thus, we now explore different techniques and discuss their limitations.

2.1 Running Example

The following code snippet shows a minimized example of Ruby code that leads to a type confusion bug in the `mruby` interpreter [16] found by a fuzzer:

```
1 NotImplementedError = String
2 Module.constants
```

In the first line, the exception type `NotImplementedError` is modified to be an alias of type `String`. As a consequence, each instance of `NotImplementedError` created in the future will be a `String` rather than the expected exception. In the second line, we call the `constants` function of `Module`. This function does not exist, provoking `mruby` to raise a `NotImplementedError`. Raising the exception causes a crash in the `mruby` interpreter.

To understand why the crash occurs, we need to dive into the C code base of the `mruby` interpreter. Note that `mruby` types are implemented as structs on the interpreter level. When we re-assign the exception type `NotImplementedError` to `String`, this is realized on C level by modifying the pointer such that it points to a struct representing the `mruby String` type. The method `Module.constants` is only a stub that creates and raises an exception. When the exception is raised in the second line, a new instance of `NotImplementedError` is constructed (which now actually results in a `String` object) and passed to `mruby`'s custom exception handling function. This function assumes that the passed object has an exception type without checking this further. It proceeds to successfully attach some error message—here "Module.constants not implemented" (length $0x20$)—to the presumed exception object. Then, the function continues to fill the presumable exception with debug information available. During this process, it attempts to dereference a pointer to a table that is contained within all exception objects. However, as we have replaced the exception type by the string type, the layout of the underlying struct is different: At the accessed offset, the `String` struct stores the length of the contained string instead of a pointer as it would be the case for the exception struct. As a result, we do not dereference the pointer but interpret the length field as an address, resulting in an attempt to dereference $0x20$. Since this leads to an illegal memory access, the program crashes.

To sum up, redefining an exception type with a string leads to a type confusion vulnerability, resulting in a crash when this exception is raised. The developer fix introduces a type check, thus preventing this bug from provoking a crash.

2.2 Crash Triaging

Assume our goal is to triage the previously explained bug, given only the crashing input (obtained from a fuzzing run) as a starting point. In the following, we discuss different approaches to solve this task and explain their challenges.

Debugger. Starting at the crashing location, we can manually inspect the last few instructions executed, the registers at crashing point and the call stack leading to this situation. Therefore, we can see that $0x20$ is first loaded to some register and then dereferenced, resulting in the crash. Our goal then is to identify *why* the code attempts to dereference this value and *how* this value ended up there. We might turn towards the call stack, which indicates that the problem arises during some helper function that is called while raising an exception. From this point on, we can start investigating by manually following the flow of execution backward from the crashing cause up to the root cause. Given that the code of the `mruby` interpreter is non-trivial and the bug is somewhat complex, this takes a lot of time. Thus, we may take another angle and use some tool dedicated to detecting memory errors, for example, sanitizers.

Sanitizer. Sanitizers are a class of tools that often use compile-time instrumentation to detect a wide range of software faults. There are various kinds of sanitizers, such as MSAN [52] to detect usage of uninitialized memory or ASAN [51] to detect heap- and stack-based buffer overflows, use-after-free (UAF) errors and other faults. Sanitizers usually rely on the usage of shadow memory to track whether specific memory can be accessed or not. ASAN guards allocated memory (e. g., stack and heap) by marking neighboring memory as non-accessible. As a consequence, it detects out-of-bounds accesses. By further marking freed memory as non-accessible (as long as other free memory is available for allocation), temporal bugs can be detected. MSAN uses shadow memory to track for each bit, whether it is initialized or not, thereby preventing unintended use of uninitialized memory.

Using such tools, we can identify invalid memory accesses even if they are not causing the program to crash immediately. This situation may occur when other operations do not access the overwritten memory. Additionally, sanitizers provide more detailed information on crashing cause and location. As a consequence, sanitizers are more precise and pinpoint issues closer to the root cause of a bug.

Unfortunately, this is not the case for our example: recompiling the binary with ASAN provides no new insights because the type confusion does not provoke any memory errors that can be detected by sanitizers. Consequently, we are stuck at the same crashing location as before.

Backward Taint Analysis. To deepen our understanding of the bug, we could use automated root cause analysis tools [32,33,57] that are based on reverse execution and backward taint tracking to increase the precision further. However, in our example, there is no direct data flow between the crash site and the actual root cause. The data flow ends in the constructor of a new `String` that is unrelated to the actual root cause. As taint tracking does not provide interesting information, we try to obtain related inputs that trigger the same bug in different crashing locations. Finding such inputs would give us a different perspective on the bug's behavior.

Crash Exploration. To achieve this goal, we can use the so-called *crash exploration* mode [58] that fuzzers such as AFL [59] provide. This mode takes a crashing input as a seed and mutates it to generate new inputs. From the newly generated inputs, the fuzzer only keeps those in the fuzzing queue that still result in a crash. Consequently, the fuzzer creates a diverse set of inputs that mostly lead to the same crash but exhibited new code coverage by exercising new paths. These inputs are likely to trigger the same bug via different code paths.

To gain new insights into the root cause of our bug, we need the crash exploration mode to trigger new behavior related to the type confusion. In theory, to achieve this, the fuzzer could assign another type than `String` to `NotImplementedError`. However, fuzzers such as AFL are more likely to modify the input to something like "`Stringgg`" or "`Strr`" than assigning different, valid types. This is due to the way its mutations work [30]. Still, AFL manages to find various crashing inputs by adding new `mruby` code unrelated to the bug.

To further strengthen the analysis, a fuzzer with access to domain knowledge, such as *grammar-based fuzzers* [28,35,48], can be used. Such a fuzzer recognizes that `String` is a grammatically valid element for Ruby which can be replaced by other grammar elements. For example, `String` can be replaced by `Hash`, `Array` or `Float`. Assume that the fuzzer chooses `Hash`; the newly derived input crashes the binary at a later point of execution than our original input. This result benefits the analyst as comparing the two inputs indicates that the crash could be related to `NotImplementedError`'s type. As a consequence, the analyst might start focusing on code parts related to the object type, reducing the scope of analysis. Still, this leaves the human analyst with an additional input to analyze, which means more time spent on debugging.

Overall, this process of investigating the root cause of a given bug is not easy and—depending on the bug type and its complexity—may take a significant amount of time and domain knowledge. Even though various methods and tools exist, the demanding tasks still have to be accomplished by a human. In the following, we present our approach to automate the process of identifying and explaining the root cause for a given crashing input.

3 Design Overview

Given a crashing input and a binary program, our goal is to find an explanation of the underlying software fault's root cause. We do so by locating behavioral differences between crashing and non-crashing inputs. In its core, our method conducts a statistical analysis of differences between a set of crashing and non-crashing inputs. Thus, we first create a dataset of diverse program behaviors related to the crash, then monitor relevant input behavior and, finally, comparatively analyze them. This is motivated by the insight that crashing inputs must—at some point—semantically deviate from non-crashing inputs. Intuitively, the first relevant behavior during program execution that causes the deviation is the root cause.

In a first step, we create two sets of related but diverse inputs, one with crashing and one with non-crashing inputs. Ideally, we only include crashing inputs caused by the same root cause. The set of non-crashing inputs has no such restrictions, as they are effectively used as counterexamples in our method. To obtain these sets, we perform crash exploration fuzzing on one initial crashing input (a so-called *seed*).

Given the two sets of inputs, we observe and monitor (i. e., trace) the program behavior for each input. These traces allow us to correlate differences in the observations with the outcome of the execution. Using this statistical reasoning, we can identify differences that predict whether a program execution will crash or not. To formalize these differences, we synthesize predicates that state whether a bug was triggered. Intuitively, the first predicate that can successfully predict the outcome of all (or most) executions also explains the root cause. As the final result, we provide the analyst with a list of relevant explanations and addresses, ordered by the quality of their prediction and time of execution. That is, we prefer explanations that predict the outcome well. Amongst good explanations, we prefer these that are able to predict the crash as early as possible.

On a high-level view, our design consist of three individual components: (1) *input diversification* to derive two diverse sets of inputs (crashing and non-crashing), (2) *monitoring input behavior* to track how inputs behave and (3) *explanation synthesis* to synthesize descriptive predicates that distinguish crashing from non-crashing inputs. In the following, we present each of these components.

3.1 Input Diversification

As stated before, we need to create a diverse but similar set of inputs for the single crashing seed given as input to our approach. On the one hand, the inputs should be diverse such that statistical analysis reveals measurable differences. On the

other hand, the inputs should share a similar basic structure such that they explore states similar to the root cause. This allows for a comparative analysis of how crashes and non-crashes behave on the buggy path.

To efficiently generate such inputs, we can use the *crash exploration mode* bundled with fuzzers such as AFL. As described previously, this mode applies mutations to inputs as long as they keep crashing. Inputs not crashing the binary are discarded from the queue and saved to the non-crashing set; all inputs remaining within the fuzzing queue constitute the crashing set. In general, the more diversified inputs crash exploration produces, the more precise the statistical analysis becomes. Fewer inputs are produced in less time but cause more false positives within the subsequent analysis. Once the input sets have been created, they are passed to the analysis component.

3.2 Monitoring Input Behavior

Given the two sets of inputs—crashing and non-crashing—we are interested in collecting data allowing semantic insights into an input's behavior. To accommodate our binary-only approach, we monitor the runtime execution of each input, collecting the values of various expressions. For each instruction executed, we record the minimum and maximum value of all modified registers (this includes general-purpose registers and the flag register). Similarly, we record the maximum and minimum value stored for each memory write access. Notably and perhaps surprisingly, we did not observe any benefit in tracing the memory addresses used; therefore, we do not aggregate information on the target addresses. It seems that the resulting information is too noisy and all relevant information is already found in observed registers. We only trace the minimum and maximum of each value to limit the amount of data produced by loops. This loss of information is justified by the insight that values causing a crash usually surface as either a minimum or maximum value. Our evaluation empirically supports this thesis. This optimization greatly increases the performance, as the amount of information stored per instruction is constant. At the same time, it is precise enough to allow statistical identification of differences. Besides register and memory values, we store information on control-flow edges. This allows us to reconstruct a coarse control-flow graph for a specific input's execution. Control flow is interesting behavior, as it may reveal code that is only executed for crashing inputs. Furthermore, we collect the address ranges of stack and heap to test whether certain pointers are valid heap or stack pointers.

We do not trace any code outside of the main executable, i. e., shared libraries. This decreases overhead significantly while removing tracing of code that—empirically—is not interesting for finding bugs within a given binary program. For each input, we store this information within a trace file that is passed on to the statistical analysis.

3.3 Explanation Synthesis

Based on the monitoring, explanation synthesis is provided with two sets of trace files that describe intrinsic behaviors of crashing and non-crashing inputs. Our goal is to isolate behavior in the form of predicates that correlate to differences between crashing and non-crashing runs. Any such predicate pointing to an instruction indicates that this particular instruction is related to a bug. Our predicates are Boolean expressions describing concrete program behavior, e. g., "the maximum value of `rax` at this position is less than 2". A predicate is a triple consisting of a semantic description (i. e., the Boolean expression), the instruction's address at which it is evaluated and a score indicating the ability to differentiate crashes from non-crashes. In other words, the score expresses the probability that an input crashes for which the predicate evaluates to true. Consequently, predicates with high scores identify code locations somewhere on the path between root cause and crashing location. In the last step, we sort these predicates first by score, then by the order in which they were executed. Given this sorted list of predicates, a human analyst can then manually analyze the bug. Since these predicates and the calculation of the score are the core of our approach, we present more details in the following section.

4 Predicate-based Root Cause Analysis

Given the trace information for all inputs in both sets, we can reason about potential root cause locations and determine predicates that explain the root cause. To this end, we construct predicates capable of discriminating crashing and non-crashing runs, effectively pinpointing conditions within the program that are met only when encountering the crash. Through the means of various heuristics described in Section 4.4, we filter the conditions and deduce a set of locations close to the root cause of a bug, aiding a developer in the tedious task of finding and fixing the root cause. This step potentially outputs a large number of predicates, each of which partitions the two sets. In order to determine the predicate explaining the root cause, we set conditional breakpoints that represent the predicate semantics. We then proceed to execute the binary for each input in the crashing set, recording the order in which predicates are triggered. As a result, we obtain for each input the order in which the predicates were encountered during execution. Given this information and the predicates' scores, we can define a ranking over all predicates. In the following, we present this approach in detail.

The first step is to read the results obtained by tracing the inputs' behavior. Given these traces, we collect all control-flow transitions observed in crashing and non-crashing inputs and construct a joined control-flow graph that is later used to synthesize control-flow predicates. Afterward, we compute the set of instructions identified by their addresses that are relevant for our predicate-based analysis. Since we are interested

in behavioral differences between crashes and non-crashes, we only consider addresses that have been visited by at least one crashing and one non-crashing input. Note that—as a consequence—some addresses are discarded if they are visited in crashes but not in non-crashes. However, in such a situation, we would observe control-flow transitions to these discarded addresses from addresses that are visited by inputs from both sets. Consequently, we do not lose any precision by removing these addresses.

Based on the trace information, we generate many predicates for each address (i. e., each instruction). Then, we test all generated predicates and store only the predicate with the highest score. In the following, we describe the types of predicates we use, how these predicates can be evaluated and present our ranking algorithm. Note that by assumption a predicate forecasts a non-crash, if it is based on an instruction that was never executed. This is motivated by the fact that not-executed code cannot be the cause of a crash.

4.1 Predicate Types

To capture a wide array of possible explanations of a software fault's root cause, we generate three different categories of predicates, namely (1) control-flow predicates, (2) register and memory predicates, as well as (3) flag predicates. In detail, we use the following types of predicates:

Control-flow Predicates. Based on the reconstructed control-flow graph, we synthesize edge predicates that evaluate whether crashes and non-crashes differ in execution flow. Given a control-flow edge from `x` to `y`, the predicate `has_edge_to` indicates that we observed at least one transition from `x` to `y`. Contrary, `always_taken_to` expresses that *every* outgoing edge from `x` has been taken to `y`. Finally, we evaluate predicates that check if the number of successors is greater than or equal to $n \in \{0, 1, 2\}$.

Register and Memory Predicates. For each instruction, we generate predicates based on various expressions: the minimum and the maximum of all values written to a register or memory, respectively. For each such expression (e. g., $r = \texttt{max(rax)}$) we introduce a predicate $r < c$. We synthesize constants for c such that the predicate is a good predictor for crashing and non-crashing inputs. The synthesis is described in Section 4.3. Additionally, we have two fixed predicates testing whether expressions are valid heap or stack pointers, respectively: `is_heap_ptr`(r) and `is_stack_ptr`(r).

Flag Predicates. On the x86 and x86-64 architecture, the flag register tracks how binary comparisons are evaluated and whether an overflow occurred, making it an interesting target for analysis. We use flag predicates that each check one of the flag bits, including the carry, zero and overflow flag.

4.2 Predicate Evaluation

For each address, we generate and test predicates of all types and store the predicate with the highest score. In the following, we detail how to evaluate and score an individual predicate. Generally speaking, we are interested in measuring the quality of a predicate, i. e., how well it *predicts* the actual behavior of the target application. Thus, it is a simple binary classification. If the target application crashes on a given input—also referred to as *test case* in the following—the predicate should evaluate to true. Otherwise, it should evaluate to false. We call a predicate *perfect* if it correctly predicts the outcome of all test cases. In other words, such a predicate perfectly separates crashing and non-crashing inputs.

Unfortunately, there are many situations in which we cannot find a perfect predicate; consequently, we assign each predicate a probability on how well it predicts the program's behavior given the test cases. For example, if there are multiple distinct bugs within the input set, no predicate will explain all crashes. This can occur if the crash exploration happens to modify a crashing input in such a way that it triggers one or multiple other bug(s). Alternatively, the actual best predicate might be more complex than predicates that could be synthesized automatically; consequently, it cannot predict all cases perfectly.

To handle such instances, we model the program behavior as a noisy evaluation of the given predicate. In this model, the final outcome of the test case is the result of the predicate XORed with some random variable. More precisely, we define a predicate p as a mapping from an input trace to a Boolean variable ($p : trace \mapsto \{0, 1\}$) that predicts whether the execution crashes. Using this predicate, we build a statistical model $O(\text{input}) = p(\text{input}) \oplus R$ to approximate the observed behavior. The random variable R is drawn from a Bernoulli distribution ($R \sim \text{Bernoulli}(\theta)$) and denotes the noise introduced by insufficiently precise predicates. Whenever $R = 0$, the predicate $p(\text{input})$ correctly predicts the outcome. When $R = 1$, the predicate mispredicts the outcome. Our stochastic model has a single parameter θ that represents the probability that the predicate mispredicts the actual outcome of the test case. We cannot know the real value of θ without simulating every possible behavior of a program. Instead, we perform maximum likelihood estimation using the sample of actual test inputs to approximate a $\hat{\theta}$. This value encodes the uncertainty of the predictions made by the predicate. We later employ this uncertainty to rank the different predicates:

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

We count the number of both mispredicted crashes (C_f) and mispredicted non-crashes (N_f) divided by the number of all predictions, i. e., the number of all mispredicted inputs as well as the number of all correctly predicted crashed (C_t) and non-crashes (N_t).

As we demonstrate in Section 6.3, using crash exploration to obtain samples can cause a significant class imbalance, i. e., we may find considerably more non-crashing than crashing inputs. To avoid biasing our scoring scheme towards the bigger class, we normalize each class by its size:

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

If $\hat{\theta} = 0$, the predicate is perfect. If $\hat{\theta} = 1$, the negation of the predicate is perfect. The closer $\hat{\theta}$ is to 0.5, the worse our predicate performs in predicting the actual outcome.

Finally, we calculate a score using $\hat{\theta}$. To obtain a score in the range of $[0, 1]$, where 0 is the worst and 1 the best possible score, we calculate $2 * \texttt{abs}(\hat{\theta} - 0.5)$. We use this score to pick the best predicate for each instruction that has been visited by at least one crashing and one non-crashing input. While the score is used to rank predicates, $\hat{\theta}$ indicates whether p or its negation $\neg p$ is the better predictor. Intuitively, if $\hat{\theta} > 0.5$, p is a good predictor for non-crashing inputs. As our goal is to predict crashes, we use the negated predicate in these cases.

Example 1. *Assume that we have* $1,013$ *crashing and* $2,412$ *non-crashing inputs. Furthermore, consider a predicate* p_1*, with* $p_1 := \texttt{min}(\texttt{rax}) < 0xff$*. Then, we count* $C_f := 1013$*,* $C_t = 0$*,* $N_f = 2000$ *and* $N_t = 412$*. Therefore, we estimate* $\hat{\theta}_1 = \frac{1}{2} \cdot (\frac{1013}{1013} + \frac{2000}{2000+412}) \approx 0.9146$*). The predicate score is* $s_1 = 2 \cdot \text{abs}(0.9146 - 0.5) \approx 0.8292$*, indicating that the input is quite likely to crash the program. Even though* $\hat{\theta}$ *is large and the majority of the outcomes is mispredicted, this high score is explained by the fact that—as* $\hat{\theta}_1 > 0.5$*—we invert the predicate* p_1*. Thus, true and false positives/negatives are switched, resulting in a large amount of true positives (*$C_t = 1013$*) and true negatives (*$N_t = 2000$*) for the inverted predicate:* $\neg p_1 := \texttt{min}(\texttt{rax}) \geq 0xff$

Testing another predicate p_2 *for the same instruction with* $\hat{\theta}_2 = 0.01$*, we calculate the score* $s_2 = 2 \cdot \text{abs}(0.01 - 0.5) = 0.98$*. Since* $s_2 > s_1$*, consequently we only store* p_2 *as best predicate for this instruction.*

4.3 Synthesis of Constant Values

When computing our register and memory predicates of type $r < c$, we want to derive a constant c that splits the test inputs into crashing and non-crashing inputs based on all values observed for r during testing. These predicates can only be evaluated once a value for c is fixed. Since c can be any 64-bit value, it is prohibitively expensive to try all possible values. However, c splits the inputs into exactly two sets: Those where r is observed to be smaller than c and the rest. The only way to change the quality of the predicate is to choose a value of c that flips the prediction of at least one value of r. All constants c between two different observations of r perform the exact same split of the test inputs. Consequently, the only values that change the behavior of the predicate are exactly the observed values of r. We exploit this fact to find the best value(s) for c using only $O(n * \log(n))$ steps where n is the number of test cases.

To implement this, we proceed as follows: In a preprocessing step, we collect all values for an expression r at the given instruction and sort them. Then, we test each value observed for r as a candidate for c. We then want to evaluate our candidate for c on all inputs reaching the address. Naively, we would recompute the score for each value of c; however, this would yield a quadratic runtime. To increase the performance, we exploit the fact that we only need C_t, C_f, N_t, N_f to calculate the score. This property of our scoring scheme allows us to update the score in constant time when checking the next candidate value of c.

To calculate the score for any candidate value c_i, we start at the smallest candidate c_0 and calculate the predicate's score by evaluating the predicate on all inputs and counting the number of correctly predicted outcomes. After calculating the score of the i^{th} possible candidate c_i, we can update the score for the candidate c_{i+1} by tracking the number of correctly predicted crashes and non-crashes. Since using c_{i+1} instead of c_i only flips a single prediction, we can efficiently update C_t, C_f, N_t, N_f in constant time. When using c_i resulted in a correctly predicted crash for the i^{th} observation, we decrement C_t. Likewise, if the old prediction was an incorrectly predicted non-crash, we decrement N_f. The other cases are handled accordingly. Afterward, we increment the number of observed outcomes based on the results of the new predicate in the same fashion. This allows us to track C_t, C_f, N_t, N_f while trying all values of c to determine the value which maximizes the score. Finally, we might have dropped some inputs that did not reach the given instruction; thus, we then perform one re-evaluation of the score on the whole dataset to determine the final score for this predicate.

Note that the predicate is constructed involving all addresses reaching that instruction. Consequently, it is perfect with respect to the whole dataset: all data not yet evaluated does not reach this address and thus cannot affect the synthesized value. Another consequence of this fact is that our synthesis works both for ranges and single values.

Example 2. *Consider that we want to synthesize a value* c *that maximizes the score of the predicate* $p(r) = r < c$*. Assume that we have four inputs reaching the address where the predicate is evaluated and we observed the following data:*

outcome	*crash*	*crash*	*non-crash*	*non-crash*
values of r	*0x08*	*0x0f*	*0x400254*	*0x400274*

In this example, the values are already sorted. Remember that we are interested in locating the cutoff value, i. e., the value of c *that separates crashing and non-crashing inputs best. Hence, we proceed to calculate the score for each candidate, starting with the smallest* $c = 0x8$*. Since* $r < 0x8$ *is never true*

for our four inputs, they are all predicted to be non-crashing. Therefore, we obtain $C_f = 2$, $C_t = 0$, $N_f = 0$,$N_t = 2$. *This results in* $\hat{\theta} = \frac{1}{2}\left(\frac{2}{2+0} + \frac{0}{0+2}\right) = 0.5$ *and, consequently, in a score* $= 2 * abs(\hat{\theta} - 0.5) = 0$, *indicating that this is not a good candidate for c. Using the next candidate* $c = 0x0\text{f}$, *we now predict that the first input is crashing. Since the first input triggered a crash, we update* C_f *and* C_t *by incrementing* C_t *and decrementing* C_f. *Consequently, we obtain* $C_f = 1$, $C_t = 1$, $N_f = 0$ *and* $N_t = 2$, *resulting in* $\hat{\theta} = 0.75$ *and a final score of* 0.5. *Repeating this for the next step, we obtain a perfect score for the next value* $0x400254$ *as both crashing values are smaller. This yields the final predicate* $p(r) = x < 0x400254$ *that will be re-evaluated on the whole dataset.*

We observed that if *all* recorded constants are either valid stack or heap addresses (i. e., pointers), we receive a high number of false positives since these addresses are too noisy for statistical analysis. Accordingly, we do not synthesize predicates other than `is_heap_ptr` and `is_stack_ptr` for these cases.

4.4 Ranking

Once all steps of our statistical analysis are completed, we obtain the best predicate for each instruction. A predicate's score indicates how well a predicate separates crashing and non-crashing inputs. Since we synthesize one predicate for each instruction, we obtain a large number of predicates. Note that most of them are independent of the bug; thus, we discard predicates with a score lower than the empirically determined threshold of 0.9. Consequently, the remaining predicates identify locations that are related to the bug.

Still, we do not know in which order relevant predicates are executed; therefore, we cannot distinguish whether a predicate is related to the root cause or occurs later on the path to the crash site. As predicates early in the program trace are more likely to correspond to the root cause, we introduce a new metric called the *execution rank*. To calculate the execution rank, we determine the temporal order in which predicates are executed. To do so, we add a conditional breakpoint for each relevant predicate p. This breakpoint triggers if the predicate evaluates to true. For each crashing input, we can execute the program, recording the order in which breakpoints are triggered. If some predicate p is at program position i and we observed n predicates in total, p's execution rank is $\frac{i}{n}$. If some predicate is not observed for a specific run, we set its execution rank to 2 as a penalty. Since a predicate's execution rank may differ for each crashing input due to different program paths taken, we average over all executions.

However, the primary metric is still its prediction score. Thus, we sort predicates by their prediction score and resolve ties by sorting according to the execution rank.

Example 3. *Consider three predicates* p_1, p_2 *and* p_3 *with their respective scores* 1, 0.99 *and* 0.99. *Furthermore, assume that we have the crashing inputs* i_1 *and* i_2. *Let the observed predicate order be* (p_1, p_3) *for* i_1 *and* (p_1, p_3, p_2) *for* i_2. *Then, we obtain the execution ranks:*

p_1: $\frac{1}{2} \cdot \left(\frac{1}{2} + \frac{1}{3}\right) \approx 0.41$
p_2: $\frac{1}{2} \cdot \left(2 + \frac{3}{3}\right) = 1.5$
p_3: $\frac{1}{2} \cdot \left(\frac{2}{2} + \frac{2}{3}\right) \approx 0.83$

Since we sort first by score and then by execution rank, we obtain the final predicate order (p_1, p_3, p_2).

5 Implementation

To demonstrate the practical feasibility of the proposed approach, we implemented a prototype of AURORA. We briefly explain important implementation aspects in the following, the full source code is available at https://github.com/RUB-SysSec/aurora.

Input Diversification. For the purpose of exploring inputs close to the original crash, we use AFL's *crash exploration mode* [58]. Given a crashing input, it finds similar inputs that still crash the binary. Inputs not crashing the program are not fuzzed any further. We modified AFL (version 2.52b) to save these inputs to the *non-crashing set* before discarding them from the queue.

Monitoring Input Behavior. To monitor the input behavior, we implemented a pintool for Intel PIN [40] (version 3.7). Relying on Intel's generic and architecture-specific inspection APIs, we can reliably extract relevant information.

Explanation Synthesis. The explanation synthesis is written in Rust. It takes two folders containing traces of crashes and non-crashes as input. Then, it reconstructs the joined control-flow graph and then synthesizes and evaluates all predicates. Finally, it monitors and ranks the predicates as described before. To monitor the execution of the predicates, we set conditional breakpoints using the `ptrace` syscall. In a final step, we use `binutils`' `addr2line` [36] to infer the source file, function name and line for each predicate. If possible, all subsequent analysis parts are performed in parallel. Overall, about 5,000 lines of code were written for this component.

6 Experimental Evaluation

Based on the prototype implementation of AURORA, we now answer the following research questions:

RQ 1: Is AURORA able to identify and explain the root cause of complex and highly exploitable bug classes such as type confusions, use-after-free vulnerabilities and heap buffer overflows?

RQ 2: How close is the automatically identified explanation to the patch implemented by the developers?

RQ 3: How many predicates are related to the fault?

To answer these research questions, we devise a set of experiments where we analyze various types of software faults. For each fault, we have manually analyzed and identified the

root cause; furthermore, we considered the patches provided by the developers.

6.1 Setup

All of our experiments are conducted within a cloud VM with 32 cores (based on Intel Xeon Silver 4114, 2.20 GHz) and 224 GiB RAM. We use the Ubuntu 18.04 operating system. To facilitate deterministic analysis, we disable address space layout randomization (ASLR).

We selected 25 software faults in different well-known applications, covering a wide range of fault types. In particular, we picked the following bugs:

- ten heap buffer overflows, caused by an integer overflow (#1 `mruby` [1]), a logic flaw (#2 `Lua` [2], #3 `Perl` [3] and #4 `screen` [4]) or a missing check (#5 `readelf` [5], #6 `mruby` [6], #7 `objdump` [7], #8 `patch` [8]), #9 `Python` 2.7/3.6 [9] and #10 `tcpdump` [10])
- one null pointer dereference caused by a logic flaw (#11 `NASM` [11])
- three segmentation faults due to integer overflows (#12 `Bash` [12] and #13 `Bash` [13]) or a race condition (#14 `Python` 2.7 [14])
- one stack-based buffer overflow (#15 `nm` [15])
- two type confusions caused by missing checks (#16 `mruby` [16] and #17 `Python` 3.6 [17])
- three uninitialized variables caused by a logic flaw (#18 `Xpdf` [18]) or missing checks (#19 `mruby` [19] and #20 `PHP` [20])
- five use-after-frees, caused by a double free (#21 `libzip` [21]), logic flaws (#22 `mruby` [22], #23 `NASM` [23] and #24 `Sleuthkit` [24]) or a missing check (#25 `Lua` [25])

These bugs have been uncovered during recent fuzzing runs or found in the bug tracking systems of well-known applications. Our general selection criteria are (i) the presence of a proof-of-concept file crashing the application and (ii) a developer-provided fix. The former is required as a starting point for our analysis, while the latter serves as ground truth for the evaluation.

For each target, we compile two binaries: One instrumented with AFL that is used for crash exploration and one non-instrumented binary for tracing purposes. Note that some of the selected targets (e. g., #1, #5 or #19) are compiled with sanitizers, ASAN or MSAN, because the bug only manifests when using a sanitizer. The targets compiled without any sanitizer are used to demonstrate that we are not relying on any sanitizers or requiring source code access. The binary used for tracing is always built with debug symbols and without sanitizers. For the sake of the evaluation, we need to measure the quality of our explanations, as stated in the **RQ 1** and **RQ 2**. Therefore, we use debug symbols and the application's source code to compare the identified root cause with the developer fix. To further simplify this process, we derive source line, function name and source file for each predicate via `addr2line`. This does not imply that our approach by any means requires source code: all our analysis steps run on the binary level regardless of available source code. Experiments using a binary-only fuzzer would work the exact same way. However, establishing the ground truth would be more complex and hence we use source code strictly for evaluation purposes.

For our evaluation, we resort to the well-known AFL fuzzer and run its crash exploration mode for two hours with the proof-of-concept file as seed input. We found that this is enough time to produce a sufficiently large set of diverse inputs for most targets. However, due to the highly structured nature of the input languages for `mruby`, `Lua`, `nm`, `libzip`, `Python` (only #17) and `PHP`, AFL found less than 100 inputs within two hours. Thus, we repeat the crash exploration with 12 hours instead of 2 hours. Each input found during exploration is subsequently traced. Since some inputs do not properly terminate, we set a timeout of five minutes after which tracing is aborted. Consequently, we do lose a few inputs, see Table 4 for details. Similarly, our predicate ranking component may encounter timeouts. As monitoring inputs with conditional breakpoints is faster than tracing an input, we empirically set the default timeout to 60 seconds.

6.2 Experiment Design

An overview of all analysis results can be found in Table 1. Recall that in practice the crashing cause and root cause of a bug differ. Thus, for each bug, we first denote its root cause as identified by AURORA and verified by the developers' patches. Subsequently, we present the crashing cause, i. e., the reason reported by ASAN or identified manually. For each target, we record the best predicate score observed. Furthermore, we investigate each developer fix, comparing it to the root cause identified by our automated analysis. We report the number of predicates an analyst has to investigate before finding the location of the developers' fix as *Steps to Dev. Fix*. We additionally provide the number of source code lines (column *SLOC*) a human analyst needs to inspect before arriving at the location of the developer fix since these fixes are applied on the source code level. Note that this number may be smaller than the number of predicates as one line of source code usually translates to multiple assembly instructions. Up to this day, no developer fix was provided for bug #23 (`NASM`). Hence, we manually inspected the root cause, identifying a reasonable location for a fix. Bug #11 has no unique root cause; the bug was fixed during a major rewrite of program logic (20 files and 500 lines changed). Thus, we excluded it from our analysis.

To obtain insights into whether our approach is actually capable of identifying the root cause even when it is separated from the crashing location by the order of thousands of instructions, we design an experiment to measure the dis-

Table 1: Results of our root cause explanations. For 25 different bugs, we note the target, root and crashing cause as well as whether the target has been compiled using a sanitizer. Furthermore, we provide the number of predicates and source lines (SLOC) a human analyst has to examine until the location is reached where the developers applied the bug fix (denoted as *Steps to Dev. Fix*). Finally, the number of true and false positives (denoted as TP and FP) of the top 50 predicates are shown. * describes targets where no top 50 predicates with a score above or equal to 0.9 exist.

	Target	Root Cause	Crash Cause	Sanitizer	Best Score	Steps to Dev. Fix		Top 50	
						#Predicates	#SLOC	TP	FP
#1	mruby	int overflow	heap buffer overflow	ASAN	0.998	1	1	50	0
#2	Lua	logic flaw	heap buffer overflow	ASAN	1.000	1	1	50	0
#3	Perl	logic flaw	heap buffer overflow	-	1.000	13	10	43	7
#4	screen *	logic flaw	heap buffer overflow	-	0.999	26	16	30	0
#5	readelf	missing check	heap buffer overflow	ASAN	1.000	7	5	50	0
#6	mruby	missing check	heap buffer overflow	ASAN	1.000	1	1	12	38
#7	objdump	missing check	heap buffer overflow	ASAN	0.981	3	3	48	2
#8	patch	missing check	heap buffer overflow	ASAN	0.997	1	1	50	0
#9	Python	missing check	heap buffer overflow	-	1.000	46	28	44	6
#10	tcpdump	missing check	heap buffer overflow	-	0.994	1	1	50	0
#11	NASM	logic flaw	nullptr dereference	-	1.000	-	-	50	0
#12	Bash	int overflow	segmentation fault	-	0.992	10	6	28	22
#13	Bash	int overflow	segmentation fault	-	0.999	9	6	35	15
#14	Python	race condition	segmentation fault	-	1.000	13	13	27	23
#15	nm *	missing check	stack buffer overflow	ASAN	0.980	1	1	35	0
#16	mruby	missing check	type confusion	-	1.000	33	15	50	0
#17	Python	missing check	type confusion	-	1.000	215	141	7	43
#18	Xpdf	logic flaw	uninitialized variable	ASAN	0.997	16	11	50	0
#19	mruby	missing check	uninitialized variable	MSAN	1.000	16	5	50	0
#20	PHP	missing check	uninitialized variable	MSAN	1.000	42	19	29	21
#21	libzip *	double free	use-after-free	ASAN	1.000	1	1	39	0
#22	mruby	logic flaw	use-after-free	ASAN	1.000	9	6	42	8
#23	NASM *	logic flaw	use-after-free	-	0.957	1	1	14	9
#24	Sleuthkit	logic flaw	use-after-free	-	1.000	2	2	48	2
#25	Lua	missing check	use-after-free	ASAN	1.000	3	3	50	0

tance between developer fix and crashing location in terms of executed assembly instructions. More specifically, for each target, we determine the maximum distance, the average distance over all crashing inputs and—to put this number in relation—the average of total instructions executed during a program run. Each metric is given in the number of assembly instructions executed and unique assembly instructions executed, where each instruction is counted at most once. Note that some bugs only crash in the presence of a sanitizer (as indicated by ASAN or MSAN in Table 1) and that our tracing binaries are never instrumented to avoid sanitizer artifacts disturbing our analysis. As a consequence, our distance measurement would run until normal program termination rather than program crash for such targets. Since this would distort the experiment, we exclude such bugs from the comparison.

Finally, to provide an intuition of how well our approach performs, we analyze the top 50 predicates (if available) produced for each target, stating whether they are related to the bug or unrelated false positives. We consider predicates as related to the bug when they pinpoint a location on the path from root cause to crashing location and separate crashing and non-crashing inputs. For false positives, we evaluate various heuristics that allow to identify them and thereby reduce the amount of manual effort required.

6.3 Results

Following AURORA's results, the developer fix will be covered by the first few explanations. Typically, an analyst would have to inspect less than ten source code lines to identify the root cause. Exceptions are larger targets, such as Python (13 MiB) and PHP (31 MiB), or particularly challenging bugs such as type confusions (#16 and #17). Still, the number of source code lines to inspect is below 28 for all but the Python type confusion (#17), which contains a large amount of false positives. Despite the increased number of source code lines to investigate, the information provided by AURORA is still useful: for instance, for bug #16—where 15 lines are needed—most of the lines are within the same function and only six functions are identified as candidates for containing the root

Table 2: Maximum and average distance between developer fix and crashing location in both all and unique executed assembly instructions. For reference, the average amount of instructions executed between program start and crash is also provided.

	Target	Maximum #Instructions		Average #Instructions		Average Total #Instructions	
		all	unique	all	unique	all	unique
#3	Perl	845,689	7,321	435,873	5,697	1,355,013	32,259
#4	screen	28,289,736	3,441	127,459	1,932	397,595	9,456
#9	Python	3,759,699	9,330	743,216	5,445	34,914,300	60,508
#10	tcpdump	6,727	1,567	2,263	546	103,655	3,622
#11	NASM	22,678,105	8,256	1,940,592	4,383	2,546,740	9,729
#12	Bash	450,428	3,549	11,965	116	1,053,498	19,221
#13	Bash	2,584,606	1,094	178,873	612	1,100,495	16,817
#14	Python	3,923,167	13,028	58,990	835	29,226,209	60,917
#16	mruby	253,173	840	2,154	533	14,926,707	26,982
#17	Python	800	428	498	407	46,112,224	74,590
#23	NASM	7,401,732	4,842	184,036	2,919	2,885,104	8,244
#24	Sleuthkit	199	156	197	155	25,780	5,960

cause. We explain the increased number of false positives found for these targets at the end of this section in detail.

Another aspect of a bug's complexity is the distance between the root cause and crashing location. As Table 2 indicates, AURORA is capable of both identifying root causes when the distance is small (a few hundred instructions, e. g., 197 for `Sleuthkit`) and significant (millions of instructions, e. g., roughly 28 million for `screen`). Overall, we conclude **RQ 1** and **RQ 2** by finding that AURORA is generally able to provide automated root cause explanations close to the root cause—less than 30 source code lines and less than 50 predicates—for diverse bugs of varying complexity.

The high quality of the explanations generated by AURORA is also reflected by its high precision (i. e., the ratio of true positives to all positives). Among the top 50 predicates, there are significantly more true positives than negatives. More precisely, for 18 out of 25 bugs, we have a precision ≥ 0.84, including 12 bugs with a precision of 1.0 (no false positives). Only for two bugs, the precision is less than 0.5—0.14 for #17 and 0.24 for #6. Note that for #6, the predicate pinpointing the developer fix is at the top of the list, rendering all these false positives irrelevant to triaging the root cause.

Despite the high precision, some false positives are generated. During our evaluation, we observed that they are mostly related to (1) (de-)allocation operations as well as garbage collectors, (2) control-flow, i. e., predicates which indicate that non-crashes executed the pinpointed code in diverse program contexts (e. g., different function calls or more loop iterations), (3) operations on (complex) data structures such as hash maps, arrays or locks, (4) environment, e. g., parsing command-line arguments or environment variables (5) error handling, e. g., signals or stack unwinding. Such superficial features may differentiate crashes and non-crashes but are generally not related to the actual bug (excluding potential edge cases like bugs in the garbage collector). Many of these false positives occur due to insufficient code coverage achieved during crash exploration, causing the sets of crashing and non-crashing inputs to be not diverse enough.

To detect such false positives during our evaluation, we employed various heuristics: First, we use the predicate's annotations to identify functions related to one of the five categories of false positives and discard them. Then, for each predicate, we inspect concrete values that crashes and non-crashes exhibit during execution. This allows us to compare actual values to the predicate's explanation and—together with the source code line—recognize semantic features such as loop counters or constants based on timers. Once a false positive is identified, we discard any propagation of the predicate's explanation and thereby subsequent related predicates. In our personal experience, these heuristics allow us to reliably detect many false positives without considering data-flow dependencies or other program context. This is supported by our results detailed in Table 3. Based on the five categories, we evaluate how many false positives within the top 50 predicates can be identified heuristically. Additionally, we denote the number of propagations as well as the number of false positives that must be analyzed in-depth. Note that an analyst had to conduct such an analysis for only half of the targets with false positives. We note that this may differ for other bugs or other target applications, especially edge cases such as bugs in the allocator or garbage collector.

Since we use a statistical model, false positives are a natural side effect, yet, precisely this noisy model is indispensable. For 15 of the analyzed bugs, we could find a perfect predicate (with a score of 1.0), i. e., predicates that perfectly distinguish crashes and non-crashes. In the remaining ten cases, some noise has been introduced by crash exploration. However, as our results indicate, small amounts of noise do not impair our analysis. Therefore, we answer **RQ 3**, concluding that nearly

Table 3: Analysis results of false positives within the top 50 predicates. For each target, we classify its false positives into the categories they are related to: allocation or garbage collector (Alloc), control flow (CF), data structure (DS), environment (Env) or error handling (Error). Additionally, we track the number of predicates an analyst has to inspect in more detail (In-depth Analysis) as well as propagations of false positives that can be discarded easily.

	Target	False Positive Categories					Propagations	In-depth Analysis
		Alloc	CF	DS	Env	Error		
#3	Perl	-	-	7	-	-	-	-
#6	mruby	-	-	38	-	-	-	-
#7	objdump	-	2	-	-	-	-	-
#9	Python	-	1	-	2	3	-	-
#12	Bash	1	1	-	1	4	8	7
#13	Bash	1	1	-	-	4	5	4
#14	Python	-	-	-	3	-	15	5
#17	Python	40	-	2	-	-	-	1
#20	PHP	-	-	-	21	-	-	-
#22	mruby	-	1	-	-	-	4	3
#23	NASM	3	-	-	-	2	2	2
#24	Sleuthkit	-	2	-	-	-	-	-

all predicates found by AURORA are strongly related to the actual root cause.

Since the statistical model is only as good as the data it operates on, we also investigate the crash exploration and tracing phases. The results are presented in Table 4. Most traces produced by crash exploration could be traced successfully. The only exception being `Bash`, which caused many non-terminating runs that we excluded from subsequent analysis. Note that we were still able to identify the root cause.

We also investigate the time required for tracing, predicate analysis and ranking. We present the results in Table 5. On average, AURORA takes about 50 minutes for tracing, while the predicate analysis takes roughly 18 minutes and ranking four minutes. While these numbers might seem high, remember that the analysis is fully automated. In comparison, an analyst triaging these bugs might spend multiple days debugging specific bugs and identifying why the program crashes.

6.4 Case Studies

In the following, we conduct in-depth case studies of various software faults to illustrate different aspects of our approach.

6.4.1 Case Study: Type Confusion in `mruby`

First, we analyze the results of our automated analysis for the example given in Section 2.1 (Bug #16). As described, the `NotImplementedError` type is aliased to the `String` type, leading to a type confusion that is hard to spot manually. Consequently, it is particularly interesting to see whether our automated analysis can spot this elusive bug. As exploring the behavior of `mruby` was challenging for AFL, we ran the initial crash exploration step for 12 hours in order to get more than 100 diversified crashes and non-crashes. Running our subsequent analysis on the best 50 predicates reported by AURORA, we manually found that all of the 50 predicates are related to the bug and provide insight into some aspects of the root cause.

The line with the predicate describing the location of the developers' fix is ranked 15th. This means that an analyst has to inspect 14 lines of code that are related to the bug but do not point to the developer fix. In terms of predicates, the 33rd predicate explains the root cause. This discrepancy results from the fact that one source code line may translate to multiple assembly instructions. Thus, multiple predicates may refer to values used in the same source code line.

The root cause predicate itself conditions on the fact that the minimal value in register `rax` is smaller than 17. Remember that the root cause is the missing type check. Types in `mruby` are implemented as `enum`, as visible in the following snippet of mruby's source code (`mruby/value.h`):

```
112    MRB_TT_STRING,        /* 16 */
113    MRB_TT_RANGE,         /* 17 */
114    MRB_TT_EXCEPTION,     /* 18 */
```

Our identified root cause pinpoints the location where the developers insert their fix and semantically states that the type of the presumed exception object is smaller than 17. In other words, the predicate distinguishes crashes and non-crashes according to their type. As can be seen, the `String` type has a value of 16; thus, it is identified as crashing input, while the exception type is assigned 18. This explains the type confusion's underlying fault.

The other predicates allow tracing the path from the root cause to the crashing location. For example, the predicates rated best describe the freeing of an object within the garbage collector. This is because the garbage collector spots that `NotImplemenetedError` is changed to point to `String` in-

Table 4: Number of crashing (#c) and non-crashing (#nc) inputs found by crash exploration (Exploration) as well as the percentage of how many could be successfully traced (Tracing).

	Target	Exploration		Tracing	
		#c	#nc	#c	#nc
#1	mruby	120	2708	100%	99.9%
#2	Lua	398	1482	100%	100%
#3	Perl	1591	6037	100%	99.9%
#4	screen	858	2164	100%	100%
#5	readelf	687	1803	100%	100%
#6	mruby	809	3914	100%	99.9%
#7	objdump	27	122	100%	100%
#8	patch	266	886	74.8%	89.7%
#9	Python	211	1546	100%	100%
#10	tcpdump	161	619	100%	100%
#11	NASM	2476	2138	100%	100%
#12	Bash	842	5483	7.1%	15.9%
#13	Bash	213	2102	50.7%	55.5%
#14	Python	253	1695	98.0%	98.2%
#15	nm	111	468	100%	100%
#16	mruby	1928	4063	100%	100%
#17	Python	705	2536	99.7%	99.8%
#18	Xpdf	779	545	100%	100%
#19	mruby	1128	2327	99.7%	99.9%
#20	PHP	800	2081	100%	100%
#21	libzip	36	286	100%	100%
#22	mruby	1629	3557	100%	99.9%
#23	NASM	590	1787	99.8%	100%
#24	Sleuthkit	108	175	100%	100%
#25	Lua	579	1948	100%	100%

Table 5: Time spent on tracing, predicate analysis (PA) and ranking of each target (in hours:minutes).

	Target	Tracing	PA	Ranking
#1	mruby	01:08	00:19	00:04
#2	Lua	00:09	00:03	< 1 min
#3	Perl	00:53	01:52	00:17
#4	screen	00:11	00:04	< 1 min
#5	readelf	00:05	00:02	< 1 min
#6	mruby	01:44	00:42	00:16
#7	objdump	< 1 min	< 1 min	< 1 min
#8	patch	00:36	< 1 min	< 1 min
#9	Python	01:20	00:15	00:05
#10	tcpdump	00:01	< 1 min	< 1 min
#11	NASM	00:20	00:12	00:07
#12	Bash	00:49	00:01	00:03
#13	Bash	00:26	00:02	00:01
#14	Python	01:23	00:14	00:08
#15	nm	00:01	< 1 min	< 1 min
#16	mruby	01:47	00:49	00:02
#17	Python	04:03	00:55	00:03
#18	Xpdf	00:19	00:01	00:03
#19	mruby	01:58	00:21	00:22
#20	PHP	01:16	00:47	00:03
#21	libzip	< 1 min	< 1 min	< 1 min
#22	mruby	01:57	00:49	00:16
#23	NASM	00:10	00:03	00:02
#24	Sleuthkit	< 1 min	< 1 min	< 1 min
#25	Lua	00:11	00:07	< 1 min

stead of the original class. As a consequence, the garbage collector decides to free the struct containing the original class `NotImplementedError`, a very uncommon event. Subsequent predicates point to locations where the string is attached to the presumed exception object during the raising of the exception. Additionally, predicates pinpoint the crashing location by stating that a crash will occur if the dereferenced value is smaller than a byte.

6.4.2 Case Study: Heap Buffer Overflow in `readelf`

GNU Binutils' `readelf` application may crash as a result of a heap buffer overflow when parsing a corrupted MIPS option section [5]. This bug (Bug #5) was assigned CVE-2019-9077. Note that this bug only crashes when ASAN is used. Consequently, we use a binary compiled with ASAN for crash exploration but run subsequent tracing on a non-ASAN binary. The bug is triggered when parsing a binary input where a field indicates that the size is set to 1 despite the actual size being larger. This value is then processed further, amongst others, by an integer division where it is divided by $0x10$, resulting in a value of 0. The 0 is then used as size for allocating memory for some struct. More specifically, it is passed to the `cmalloc` function that delegates the call to `xmalloc`. In this function, the size of 0 is treated as a special case where one byte should be allocated and returned. Subsequently, writing any data larger than one byte—which is the case for the struct the memory is intended for—is an out-of-bounds write. As no crucial data is overwritten, the program flow continues as normal unless it was compiled with ASAN, which spots the out-of-bounds write.

To prevent this bug, the developers introduced a fix where they check whether the allocated memory's size is sufficient to hold the struct. Analyzing the top 50 predicates, we observe that each of these predicates is assigned a score larger than or equal 0.99. Our seventh predicate pinpoints the fix by making the case that an input crashes if the value in `rcx` is smaller than 7. The other predicates allow us to follow the propagation until the crashing location. For instance, two predicates exist that point to the integer division by $0x10$, which causes the 0. The first predicate states that crashes have a value smaller than $0x7$ after the division. The second predicate indicates that the zero flag is set, demonstrating a use case for our flag predicates. We further see an edge predicate, which indicates

that only crashes enter the special case, which is triggered when `xmalloc` is called with a size of 0.

6.4.3 Case Study: Use-after-free in `Lua`

In version 5.3.5, a use-after-free bug (#25, CVE-2019-6706) was found in the `Lua` interpreter [25]. `Lua` uses so-called *upvalues* to implement closures. More precisely, upvalues are used to store a function's local variables that have to be accessed after returning from the function [39]. Two upvalues can be joined by calling `lua_upvaluejoin`. The function first decreases the first upvalue's reference count and, critically, frees it if it is not referenced anymore, before then setting the reference to the second upvalue. The function does not check whether the two passed parameters are equal, which semantically has no meaning. However, in practice, the upvalue will be freed before setting the reference, thus provoking a use-after-free. ASAN detects the crash immediately while regular builds crash with a segmentation fault a few lines later.

Our approach manages to create three predicates with a score of 1. All of these three predicates are edge predicates, i. e., detecting that for crashes, another path was taken. More precisely, for the very first predicate, we see the return from the function where the second upvalue's index was retrieved. Note that this is before the developers' fix, but the first point in the program where things go wrong. The second predicate describes the function call where the upvalue references are fetched, which are then compared for equality in the developer fix, i. e., it is located closely before the fix. The third predicate is located right after the developer fix; thus, we have to inspect three predicates or three source lines until we locate the developer fix. It describes the return from the function decreasing the reference count. All other predicates follow the path from the root cause to the crashing location.

6.4.4 Case Study: Uninitialized Variable in `mruby`

The `mruby` interpreter contains a bug where uninitialized memory is accessed (Bug #19). This happens in the `unpack_m` function when unpacking a base64 encoded value from a packed string. A local `char` array of size four is declared without initialization. Then, a state machine implemented as a while loop iterates over the packed string, processing it. The local `char` array is initialized in two states during this processing step. However, crafting a specific packed string allows to avoid entering these two states. Thereby, the local array is never properly initialized and MSAN aborts program execution upon the use of the uninitialized memory.

When analyzing the top 50 predicates, we find that they are are all related to the bug. The 16th predicate pinpoints the location where the developer fix is inserted. It describes that crashes fail to pass the condition of the while loop and—as a consequence—leave the loop with the local variable being uninitialized. Another predicate we identify pinpoints if the condition allows skipping the initialization steps, stating that this is a characteristic inherent to crashing inputs. All other predicates highlight locations during or after the state machine. Note that the crash only occurs within MSAN; thus, the binary we trace does not crash. However, this does not pose a problem for our analysis, which efficiently pinpoints root cause and propagation until the crashing and non-crashing runs no longer differ. In this particular case, the uninitialized memory is used to calculate a value that is then returned. For instance, we see that the minimal memory value written is less than $0x1c$ at some address. Consequently, our analysis pinpoints locations between the root cause and the usage of the uninitialized value.

6.4.5 Case Study: Null Pointer Dereference in `NASM`

For `NASM` (#11, CVE-2018-16517), we analyze a logic flaw which results in a null pointer dereference that crashes the program. This happens because a pointer to a `label` is not properly initialized but set to `NULL`. The program logic assumes a later initialization within a state machine. However, this does not happen because of a non-trivial logic flaw. The developers fix this problem by a significant rewrite, changing most of the implementation handling labels (in total, 500 lines of code were changed). Therefore, we conclude that no particular line can be determined as the root cause; nevertheless, we investigate how our approach performs in such a scenario. This is a good example to demonstrate that sometimes defining the root cause can be a hard challenge even for a human.

Analyzing the top 50 predicates reported, we find that AURORA generates predicates pointing to various hotspots, which show that the label is not initialized correctly. More precisely, we identify a perfect edge predicate stating that the pointer is initially set to `NULL` for crashes. Subsequent predicates inform us that some function is called, which takes a pointer to the label as a parameter. They identify that for crashes the minimal value for `rdi` (the first function parameter in the calling convention) is smaller than $0xff$. Immediately before the function attempts to dereference the pointer, we see that the minimal value of `rax` is smaller than $0xff$, which indicates that the value was propagated. Afterward, a segmentation fault occurs as accessing the address 0 is illegal. In summary, we conclude that AURORA is useful to narrow down the scope even if no definite root cause exists.

7 Discussion

As our evaluation shows, our approach is capable of identifying and explaining even complex root causes where no direct correlation between crashing cause and root cause exists. Nevertheless, our approach is no silver bullet: It still reports some predicates that are not related to the root cause. Typically,

this is caused by the crash exploration producing an insufficiently diverse set of test cases. This applies particularly to any input that was originally found by a grammar-based fuzzer since AFL's type of mutations may fail to produce sufficiently diverse inputs for such targets [30]. We expect that better fuzzing techniques will improve the ability to generate more suitable corpora. Yet, no matter how good the fuzzer is, in the end, pinpointing a single root cause will remain an elusive target for automated approaches: even a human expert often fails to identify a single location responsible for a bug.

Relying on a fuzzer illustrates another pitfall: We require that bugs can be reproduced within a fuzzing setup. Therefore, bugs in distributed or heavily concurrent systems currently cannot be analyzed properly by our approach. However, this is a limitation of the underlying fuzzer rather than AURORA: Our analysis would scale to complex systems spanning multiple processes and interacting components; our statistical model can easily deal with systems where critical data is passed and serialized by various means, including networks or databases, where traditional analysis techniques like taint tracking fail.

In some cases, the predicates that we generate might not be precise enough. While this situation did not happen during our evaluation, hypothetically, there may exist bugs that can only be explained by predicates spanning multiple locations. For example, one could imagine a bug caused by using an uninitialized value, which is only triggered if two particular conditions are met: The program avoids taking a path initializing the value and later takes a path where the value is accessed. Our single-location predicates fail to capture that the bug behavior is reliant on two locations. We leave extending our approach to more complex and compound predicates as an interesting question for future work.

Last, our system requires a certain computation time to identify and explain root causes. In some cases, AURORA ran for up to 17 hours (including 12 hours for crash exploration). We argue that this is not a problem, as our system is used in combination with normal fuzzing. Thus, an additional 17 hours of fuzzing will hardly incur a significant cost for typical testing setups. Since it took us multiple person-days to pinpoint the root cause for some of the bugs we analyzed, making the integration of our fully automated approach into the fuzzing pipeline seems feasible.

An integration to fuzzing could benefit the fuzzer: Successful fuzzing runs often produce a large number of crashing inputs, many of which trigger the same crash. To save an analyst from being overwhelmed, various heuristics are deployed to identify equivalent inputs. Most rely on the input's *coverage profile* or *stack hashing* where the last n entries of the call stack are hashed [42]. Unfortunately, both techniques have been shown to be imprecise, i. e., to report far too many distinct bugs, while sometimes even joining distinct bugs into one equivalence class [42]. Given an automated approach capable of identifying the root cause such as ours, it is possible to bucket crashing inputs according to their root cause. To this end, one could pick some random crashing input, identify its root cause and then check for all remaining crashing inputs whether the predicate holds true. Each crashing input for which the predicate is evaluated to true is then collected in one bucket. For the remaining inputs, the process could be repeated until all crashing inputs have been sorted into their respective equivalence classes.

In some cases, such as closed-source binaries or a limited amount of developer time, manually implementing fixes may be impossible. An automated approach to providing (temporary) patches may be desirable. Our approach could be extended to patch the root cause predicate into the binary such that—at the point of the root cause—any input crashing the binary leads to a graceful exit rather than a potentially exploitable crash.

8 Related Work

In the following, we focus on works related closest to ours, primarily statistical and automated approaches.

Spectrum-based Fault Localization. Closest related to our work are so-called spectrum-based, i. e., code coverage-based, fault localization techniques [34]. In other words, these approaches attempt to pinpoint program elements (on different levels, e. g., single statements, basic blocks or functions) that cause bugs. To this end, they require multiple inputs for the program, some of which must crash while others may not. Often, they use test suites provided by developers and depend on the source code being available. For instance, Zhang et. al. [60] describe an approach to root cause identification targeting the Java Virtual Machine: first, they locate the non-crashing input from provided test suite whose control flow paths beginning overlaps the most with the one of the crashing input under investigation. Then, they determine the location of the first deviation, which they report as the root cause. Overall, most approaches either use some metric [26, 27, 41, 54, 55] to identify and rank possible root causes or rely on statistical techniques [43, 44, 46].

As a sub-category of spectrum-based fault localization, techniques based on statistical approaches use predicates to reason about provided inputs. Predicate-based techniques are used to isolate bugs [43] or to pinpoint the root cause of bugs [44, 46, 60]. These approaches typically require source code and mostly rely on inputs provided by test suites.

While our work is similar to such approaches with respect to sampling predicates and statistically isolating the root cause, our approach does not require access to source code since it solely works on the binary level. Furthermore, our analysis synthesizes domain-specific predicates tailored to the observed behavior of a program. Also, we do not require any test suite but rely on a fuzzer to generate test cases. This provides our approach with a more diversified set of inputs, allowing for more fine-grained analysis.

Reverse Execution. A large number of works [32, 33, 38, 45, 57] investigate the problem of analyzing a crash, typically starting from a core dump. To this end, they reverse-execute the program, reconstructing the data flow leading to the crash. To achieve this, CREDAL [56] uses a program's source code to automatically enrich the core dump analysis with information aiding an analyst in finding memory corruption vulnerabilities. Further reducing the manual effort needed, POMP requires a control-flow trace and crash dump, then uses backward taint analysis [57] to reverse the data flow, identifying program statements contributing to a crash. In a similar vein but for a different application scenario—core dumps sampled on an OS level—RETRACER [33] uses backward taint analysis without a trace to reconstruct functions on the stack contributing to a crash. Improving upon RETRACER, Cui et. al. [32] developed REPT, an *reverse debugger* that introduces an error correction mechanism to reconstruct execution history, thereby recovering data flow leading to a crash. To overcome inaccuracies, Guo et. al. [38] propose a deep-learning-based approach based on value-set analysis to address the problem of memory aliasing.

While sharing the goal of identifying instructions causing a crash, AURORA differs from these approaches by design. Reverse execution starts from a crash dump, reversing the data-flow, thereby providing an analyst with concrete assembly instructions contributing to a bug. While these approaches are useful in scenarios where a crash is not reproducible, we argue that most of them are limited to correctly identify bugs that exhibit a direct data dependency between root cause and crashing location. While REPT does not rely on such a dependency, it integrates into an interactive debugging session rather than providing a list of potential root cause predicates; thus, it is orthogonal to our approach. Moreover, AURORA uses a statistical analysis to generate predicates that not only pinpoint the root cause but also add an explanation describing how crashing inputs behave at these code locations. Furthermore, since we do not perform a concrete analysis of the underlying code, AURORA can spot vulnerabilities with no direct data dependencies.

9 Conclusion

In this paper, we introduced and evaluated a novel binary-only approach to automated root cause explanation. In contrast to other approaches that identify program instructions related to a program crash, we additionally provide semantic explanations of how these locations differ in crashing runs from non-crashing runs. Our evaluation shows that we are able to spot root causes for complex bugs such as type confusions where previous approaches failed. Given debug information, our approach is capable of enriching the analysis' results with additional information. We conclude that AURORA is a helpful addition to identify and understand the root cause of diverse bugs.

Acknowledgements

We would like to thank our shepherd Trent Jaeger and the anonymous reviewers for their valuable comments and suggestions. We also thank Nils Bars, Thorsten Eisenhofer and Tobias Scharnowski for their helpful feedback. Additionally, we thank Julius Basler and Marcel Bathke for their valuable support during the evaluation. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

References

[1] mruby heap buffer overflow (CVE-2018-10191). https://github.com/mruby/mruby/issues/3995.

[2] Lua heap buffer overflow. https://www.lua.org/bugs.html#5.0-2.

[3] Perl heap buffer overflow. https://github.com/Perl/perl5/issues/17384.

[4] screen heap buffer overflow. https://seclists.org/oss-sec/2020/q1/65.

[5] readelf heap buffer overflow (CVE-2019-9077). https://sourceware.org/bugzilla/show_bug.cgi?id=24243.

[6] mruby heap buffer overflow (CVE-2018-12248). https://github.com/mruby/mruby/issues/4038.

[7] objdump heap over flow (CVE-2017-9746). https://sourceware.org/bugzilla/show_bug.cgi?id=21580.

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

[9] Python heap buffer overflow (CVE-2016-5636). https://bugs.python.org/issue26171.

[10] tcpdump heap buffer overflow (CVE-2017-16808). https://github.com/the-tcpdump-group/tcpdump/issues/645.

[11] NASM nullpointer dereference (CVE-2018-16517). https://nafiez.github.io/security/2018/09/18/nasm-null.html.

[12] Bash segmentation fault. https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00044.html.

[13] Bash segmentation fault. https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00042.html.

[14] Python segmentation fault. https://bugs.python.org/issue31530.

[15] nm stack buffer overflow. https://sourceware.org/bugzilla/show_bug.cgi?id=21670.

[16] mruby type confusion. https://hackerone.com/reports/185041.

[17] Python type confusion. https://hackerone.com/reports/116286.

[18] Xpdf uninitialized variable. https://forum.xpdfreader.com/viewtopic.php?f=3&t=41890.

[19] mruby uninitialized variable. https://github.com/mruby/mruby/issues/3947.

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

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

[22] mruby use-after-free (CVE-2018-10199). https://github.com/mruby/mruby/issues/4001.

[23] NASM use-after-free. https://bugzilla.nasm.us/show_bug.cgi?id=3392556.

[24] Sleuthkit use-after-free. https://github.com/sleuthkit/sleuthkit/issues/905.

[25] Lua use-after-free (CVE-2019-6706). https://security-tracker.debian.org/tracker/CVE-2019-6706.

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

[27] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Localizing software faults simultaneously. In *International Conference on Quality Software*, 2009.

[28] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[29] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[30] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[31] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

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

[33] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering (ICSE)*, 2016.

[34] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.

[35] Michael Eddington. Peach fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/.

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

[37] Google. Announcing OSS-Fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[38] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *USENIX Security Symposium*, 2019.

[39] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. The implementation of Lua 5.0. *J. UCS*, 11(7):1159–1176, 2005.

[40] Intel Corporation. Pin – a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[41] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[43] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[44] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[45] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. POMP++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering*, 2019.

[46] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[47] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[48] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing, 2018.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, February 2017.

[50] Jukka Ruohonen and Kalle Rindell. Empirical notes on the interaction between continuous kernel fuzzing and development. *arXiv preprint arXiv:1909.02441*, 2019.

[51] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[52] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[54] Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. Isolating suspiciousness from spectrum-based fault localization techniques. In *International Conference on Quality Software*, 2010.

[55] Jian Xu, Zhenyu Zhang, W. K. Chan, T. H. Tse, and Shanping Li. A general noise-reduction framework for fault localization of Java programs. *Information & Software Technology*, 55(5), 2013.

[56] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. CREDAL: towards locating a memory corruption vulnerability with your core dump. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

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

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

[59] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[60] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.