The Journal of Systems and Software 209 (2024) 111886

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Valkyrie: Improving fuzzing performance through deterministic techniques ☆

Yuyang Rong [a,*], Chibin Zhang [b], Jianzhong Liu [b], Hao Chen [a]

[a] University of California, Davis, CA, USA
[b] ShanghaiTech University, Shanghai, China

ARTICLE INFO

Keywords:
Fuzzing
Dynamic analysis
Vulnerability detection

ABSTRACT

Greybox fuzzing has received much attention from developers and researchers due to its success in discovering bugs within many programs. However, randomized algorithms have limited fuzzers' effectiveness. First, branch coverage feedback that is based on random edge ID can lead to branch collision. Besides, state-of-the-art fuzzers heavily rely on randomized methods to reach new coverage. Finally, some state-of-the-art fuzzers only employ heuristics-based bug exploitation methods, which are not effective in triggering those that require non-trivial triggering conditions.

We believe deterministic techniques deliver consistent and reproducible results. We propose Valkyrie, a greybox fuzzer whose performance is boosted primarily by deterministic techniques. Valkyrie combines collision-free branch coverage with context sensitivity to maintain accuracy while introducing an instrumentation removal algorithm to reduce overhead. It also pioneers a new mutation method, compensated step, allowing fuzzers that use solvers to adapt to real-world fuzzing scenarios without randomness. Additionally, Valkyrie proactively identifies possible exploit points in target programs and utilizes solvers to trigger actual bugs. We implement and evaluate Valkyrie's effectiveness on the standard benchmark Magma, and a wide variety of real-world programs. Valkyrie triggered 21 unique integer and memory errors, 10.5% and 50% more than AFL++ and Angora, respectively. Valkyrie reached 8.2% and 12.4% more branches in real-world programs, compared with AFL++ and Angora, respectively. We also verify that our branch counting and mutation method is better than the state-of-the-art, which shows that deterministic techniques trump random techniques in consistency, reproducibility, and performance.

1. Introduction

Greybox fuzzing has achieved much progress over the past few years, becoming more accepted in industry applications while receiving much attention in academia. Fuzzing's scalability and soundness have led security researchers to find a multitude of vulnerabilities in a wide variety of software, including IoT devices (Chen et al., 2018; Wang et al., 2019b), Android apps (Liu et al., 2020), kernels (Jeong et al., 2019; Xu et al., 2020; Xu et al., 2019), and application software (Anon, 2014; Böhme et al., 2019; Chen and Chen, 2018; Fioraldi et al., 2020; Lyu et al., 2019).

Many state-of-the-art greybox fuzzers are based on American Fuzzy Lop (AFL) (Anon, 2014). AFL is a classic mutation-based greybox fuzzer offering a versatile and robust architecture that allows developers to port its design to numerous platforms and operate on vastly different fuzzing targets. This has sparked interest in the research community, conceiving a number of AFL-derived fuzzers with numerous improvements (Aschermann et al., 2019; Böhme et al., 2019; Chen and Chen, 2018; Fioraldi et al., 2020; Gan et al., 2018; Lyu et al., 2019).

However, their respective strategies are limited by randomized algorithms. For example, AFL-based fuzzers obtain program feedback in the form of branch coverage by recording the hit counts of each branch in a fixed-size bitmap called branch count table. Branches' IDs are determined randomly at static time to index the table. Randomly assigned IDs result in potential collisions where two branches correspond to the same ID, also known as the branch collision problem. On the other hand, the importance of context-sensitive branch counting can be corroborated by its extensive implementation in newer fuzzers (Chen and Chen, 2018; Fioraldi et al., 2020). The increased unique branches brought by this new context information exacerbate branch collision problem.

☆ Editor: Prof W. Eric Wong.

* Corresponding author.
E-mail addresses: PtrRong@ucdavis.edu (Y. Rong), zhangchb1@shanghaitech.edu.cn (C. Zhang), liujzh@shanghaitech.edu.cn (J. Liu), chen@ucdavis.edu (H. Chen).

https://doi.org/10.1016/j.jss.2023.111886
Received 9 February 2023; Received in revised form 31 August 2023; Accepted 25 October 2023
Available online 10 November 2023

An intuitive solution to mitigate this problem is to increase the branch count table's size, which is state-of-the-art fuzzers' approach. However, during our tests with programs such as *tcpdump*, the utilization rate of bitmaps can reach up to 36.6% even when enabling context-sensitivity using an enlarged 1MiB bitmap. As shown by Gan et al. (2018), such utilization rates can induce very high collision rates, while an enlarged buffer reduces execution throughput by 30% on some programs. AFL++'s LTO mode statically assigns each branch a unique ID to achieve collision-free. However, its design does not accommodate for context-sensitivity, which is important for the fuzzer to detect subtle but important changes in a program's execution state.

Therefore, fuzzer developers have to face a trade-off between fine-grained but slow feedback or a fast but inaccurate one. Such trade-off has been carefully studied in Wang et al. (2019a). Thus, there is a need for a better solution that takes a principled approach towards providing detailed, accurate, and efficient branch counting.

On the other hand, little effort is put into mutators. AFL-based fuzzers generally use heuristic methods, most of which are based on randomization. Even fuzzers with solvers have unrealistic assumptions, which often lead to failure and force the fuzzer to turn to randomization as a last resort. For example, in Angora, lots of "odd heuristics and parameters" (Zeller, 2019) are added to the code. These heuristics caused uneven performances across trials. Therefore, Klees et al. (2018) proposes a series of methods including repeated trials to guarantee the comparison is fair. However, real-world bugs are far and rare. Even ten repeated trials cannot guarantee a bug being found.

We carefully study the state-of-the-art fuzzers with embedded solvers and find these fuzzers generally work in the following fashion. First, the fuzzer picks an unsatisfied branch predicate to solve. Then, it identifies the input sections that can affect the predicate's outcome through techniques such as dynamic taint analysis. Next, the fuzzer uses the solver to identify and exploit certain features of the predicate to solve it. The fuzzer continues to solve the target predicate until either the predicate is satisfied or the solver has exhausted its time budget. It then picks another predicate and repeats the process mentioned above. For instance, REDQUEEN attempts to identify and tackle checksums and hashes through techniques similar to magic byte matching, but it cannot solve general arithmetic predicates (Aschermann et al., 2019). QSYM uses a modified concolic solver to solve the target predicate, but these solvers cannot solve constraints with complicated forms such as nonconvexity (Yun et al., 2018). Angora converts the predicate to an objective function $f(\mathbf{x})$ to optimize using gradient descent, where $\mathbf{x}$ represents sections of input bytes (Chen and Chen, 2018). Using numerical differentiation, Angora approximates the objective function's gradient and performs descent by mutating the corresponding input sections.

Some solvers fall back to random mutation when their assumptions do not hold for scenarios in real-world programs. Mathematical methods such as gradient descent are designed to work on functions in the real domain, which renders these solving methods ineffective against real-world constraints where many are in the bounded integer domain. Therefore, fuzzers that utilize these methods can only solve a subset of the predicates for the following reasons. (1) They believe the mutation amount $\Delta\mathbf{x}$ is *always* an integer, and (2) the predicate may overflow when the mutation amount derived from an integer $\Delta\mathbf{x}$ is too large. Therefore we need to find a way to allow solvers assuming real domain to work with branch predicates in real-world programs, allowing the fuzzer to release its full potential instead of rolling a dice and hoping for the best.

The fuzzing process involves two distinct tasks: exploration and exploitation. During the exploration phase, the fuzzer generates seeds to achieve broader coverage of the target codebase. Conversely, the exploitation phase focuses on uncovering bugs by generating seeds that lead to program crashes or other notable behaviors. While state-of-the-art fuzzers employ various strategies for exploration, many rely on heuristic-based methods for bug exploitation. For instance, popular fuzzers like AFL and AFL++ employ random mutation techniques to blindly trigger bugs. In contrast, more advanced fuzzers like Angora identify specific instructions and attempt to insert values such as `NULL` or `INT_MAX` to trigger out-of-bounds memory access bugs. However, these approaches may fall short in detecting bugs that require non-trivial triggering conditions, such as specific input values or particular triggering mechanisms. Consequently, the effectiveness of the fuzzer may be compromised in such cases.

These problems are the current blocking issues when we hope to improve fuzzing effectiveness. A collision-prone and imprecise branch coverage feedback mechanism will cancel out the benefits of improved mutation methods, as the fuzzer would likely miss the resulting increased program states. A more sophisticated mutator cannot deliver its promise unless the fundamental assumptions hold under most circumstances. Ineffective exploitation methods render the fuzzer incapable of triggering bugs within code that has been already explored. We believe deterministic algorithms produce more consistent, predictable, reproducible results. Therefore, we wish to eliminate the randomness used in these two components. After re-evaluating these methods, we design techniques that address each aspect of the issues mentioned above:

First, we combine the best of two worlds by designing a branch coverage feedback mechanism that is collision-free *and* context-sensitive. We use static analysis to identify all possible branches present within the program. Instead of assigning each branch a static ID like current approaches, we give each branch a relocatable, function-local incremental ID. Additionally, we statically determine all possible *first-order* function contexts, i.e., function contexts are determined solely by the call site. For each function, we identify its direct call sites at static time. For indirect function calls, we assume any function with the same signature may be called at runtime. Thus, each branch's context-sensitive ID at runtime is determined by its function-local ID and the current function context. Furthermore, we develop an algorithm to remove unnecessary instrumentation while maintaining accuracy to reduce the table size. We also prove the correctness of the algorithm. To adapt to more extensive programs, we statically determine the required size for the branch counting table and negotiate a suitably-sized buffer automatically with the fuzzer at initialization. This approach allows for more fine-grained feedback while reducing overhead, improving the fuzzer's ability to observe execution state changes in the program.

Next, we design a *compensated step* method that adapts solver algorithms developed for values in the real domain to integer domains, where many real-world programs run. To demonstrate the effectiveness of this approach, we use a gradient descent solver and apply our modifications. The high-level idea of this method is to clip the fractional values that could not be applied to integer values and *compensate* them to other components of the input vector. We denote the input vector as $\mathbf{x}$, the original mutation amount as $\Delta\mathbf{x} \in \mathbb{R}^n$, where n is the dimensions of the input vector, i.e., the number of input bytes of a predicate. Our target is to find a compensated mutation amount $\Delta\mathbf{x}' \in \mathbb{Z}^n$, such that $f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}+\Delta\mathbf{x}')$. We also make some modifications to the original gradient descent solver such that *compensated mutation* can perform well in real-world situations. Specifically, we first modify the initial step size such that it is set to the smallest possible value by which the predicate can change, then doubling the step size value upon each successful descent step. We also used a different differential approach to get a more precise gradient.

Finally, we propose proactive exploitation that augments the fuzzer's bug detection capabilities. This method works by first identifying *exploit points*, i.e., locations where bugs may be present, in the target program during static analysis. We identify values according to its exploitation type that may trigger a bug, including divide by zero, out-of-bounds memory access, and memory allocation. In contrast with filling the input with interesting or randomized values, we utilize the solver to change the specific input values such that each exploit point will possibly trigger a bug during fuzzing. In order to maintain fuzzing

throughput, we prioritize conventional exploration, i.e., we first solve as many branch predicates as possible, thus covering as much code as possible, then for all exploit points found in the covered code, we attempt to trigger bugs within the program. Additionally, we devise a way to lower the runtime costs due to the added instrumentation code based on the observation that most instrumented code during one execution of the fuzzed program does not require execution. We clone each function into copies that are instrumented with different instrumentation types, such as one copy for exploration feedback, another two for memory-related exploitation and integer-related exploitation, respectively. At runtime, we only execute the corresponding instrumented function if it contains the target predicate or exploit point.

We implement a prototype fuzzer Valkyrie to deliver better performance through our improvements. We evaluate Valkyrie's effectiveness on standard dataset Magma and real-world programs. On Magma, Valkyrie found 21 unique integer and memory errors with no need for any randomization methods, 10.5% and 50% more than AFL++ and Angora, respectively. We also examine the performance of Valkyrie on real-world programs. First, our tests show that Valkyrie increased branch coverage by 8.2% compared with AFL++, and 12.4% compared with Angora. Second, we demonstrate that Valkyrie's branch counting mechanism allows for collision-free branch counting. At the same time, when using a bitmap with comparable size to Valkyrie's, AFL and Angora result in significant bitmap utilization rates, leading to high occurrences of collisions. Finally, we compared Valkyrie's solver with Angora's to show that even without any heuristics, our compensated step mutation can still do better than Angora.

This paper, makes the following contributions:

(1) We propose a collision-free branch counting method and an algorithm to reduce branch count table size.
(2) We propose an efficient mutation method for predicate solver. With the new solver, we can effectively target some memory and integer bugs during fuzzing.
(3) We implement a prototype fuzzer Valkyrie using these deterministic techniques and evaluated its effectiveness and performance.
(4) We demonstrate Valkyrie delivers a more stable and uniform performance than other commonly seen fuzzers on benchmarks and real-world programs.

2. Background and motivation

AFL is a classic mutation-based greybox fuzzer. AFL monitors the program state by inserting light instrumentation and monitoring branch coverage states. It then uses a series of heuristics and randomized methods to mutate existing seeds. The instrumented program is executed using the mutated seed. AFL will save the new seed if a new branch state is triggered.

Most fuzzers in the AFL family inherit these techniques with some modifications. For instance, fuzzers in the AFL family generally use a fixed-sized bitmap to record branch coverage information, allowing the fuzzer to identify new triggered states and save the mutated input as a seed for further mutation. During program execution, the instrumentation code increments the branch's bitmap entry whenever a new branch is executed. Some AFL-derived fuzzers implement context-sensitive branch counting (Chen and Chen, 2018; Fioraldi et al., 2020) to assist in discerning more unique states.

However, since the branch ID is determined randomly during instrumentation, it is not unique and can lead to branch collision. Gan et al. demonstrated that collisions are non-trivial and increase with the number of branches present within a program (Gan et al., 2018). Paired with context-sensitivity, which significantly increases the number of unique branches observable by the fuzzer. Branch collisions pose a significant challenge when improving fuzzing effectiveness.

There are several attempts to mitigate the problem. For instance, Angora defaults to a larger bitmap size, which has been proved ineffective by Gan et al. since it does not eliminate collisions and slows down execution speed significantly. Gan et al. proposed CollAFL, which assigns IDs using non-random algorithms that greatly reduces collisions. AFL++ offers an optional *LTO mode* that provides collision-free branch counting (Fioraldi et al., 2020). However, the former cannot adjust to programs automatically, while the latter is experimental and buggy. Besides, both approaches lack context-sensitivity.

Fuzzers in the AFL family randomly mutate the entire input. Random mutation becomes somewhat ineffective after the "easy" branches are solved. More recent developments focus on using solvers to solve branch predicates to dive deeper into the code. It is guaranteed to alter the control flow once the predicate is solved and possibly yield a new path. Many solvers have been proposed, including input-to-state-correspondence (Aschermann et al., 2019), concolic solvers (Yun et al., 2018), and gradient methods (Chen and Chen, 2018). These fuzzers generally operate using the following workflow: (1) it identifies the corresponding input sections of the target predicate, (2) then it derives relevant properties of the predicate, such as the gradient, and (3) it mutates the input sections with its predicate solver using the above information.

However, these methods are limited in real-world scenarios. For example, the gradient method used in Angora assumes the input domain to be continuous when it is discrete in most cases. This limits its ability to solve many real-world predicates, which becomes difficult and almost impossible to solve using continuous domain assumptions. Listing 1 is an example code copied from libjpeg, where three input bytes are involved, two of which describes the buffer length and the other is the number of components in the buffer. There is a sanity check before the program consumes the buffer. Angora may convert the check into an objective function $f(\mathbf{x}) = |\mathbf{g}\mathbf{x}^T - 8|$ where $\mathbf{g} = [256, 1, -3]$ is the gradient. Then Angora tries to minimize it using classic gradient descent, where one can move input in arbitrarily small steps. Suppose the initial point is $\mathbf{x}_{init} = [0, 12, 1]$. When trying to take a small step $-\alpha\mathbf{g}$, say $\alpha = 0.1$, $-\alpha\mathbf{g} = [0, -0.1, 0.3]$, later two dimensions will find it unable to accept a fractional value and thus floored step to $[0, -1, 0]$ and result to $\mathbf{x} = [0, 11, 0]$. Angora would stagnate at this point. Since the first and the second dimensions are going in opposite directions, and all dimensions must be positive, Angora cannot find a next step.

One may argue that in this situation, we can use ceiling or rounding to solve this problem. However, we can always find code snippets where one operation works and the other two fail. The root cause is not clipping operations we choose to use, but that the assumption Angora made is not true in real-world programs, as each byte is bounded to $[0, 255]$ and the smallest step by which one input byte can change is either 1 or -1.

```
1 static unsigned int NEXTBYTE (void);
2 static void process_SOFn (...) {
3     unsigned int length = (NEXTBYTE() << 8) +
          NEXTBYTE();
4     unsigned int num_components = NEXTBYTE();
5     if (length != 8 + num_components * 3)
6         ERREXIT("Bogus SOF marker length");
7     ...
8 }
```

Listing 1: Code snippet copied from libjpeg-9d. The program requires the length to be a specific amount to continue.

Bug exploitation in programs is another area that state-of-the-art fuzzers cannot perform as well as expected. In this paper, exploitation refers to the process where the fuzzer tries to detect bugs from *exploit points* by generating a seed that crashes the program at the exploit point. State-of-the-art fuzzers generally employ heuristics-based bug exploitation methods. Popular fuzzers such as AFL and AFL++ generally use random mutation and, in some cases, dictionary values in an attempt to trigger bugs within the program. Angora only targets

a small subset of possible exploitation points, for example, buffer indexing operations, and sets pre-defined values based on heuristics, such as `INT_MAX` or `NULL`. While these methods have been able to find numerous bugs in the history of these tools, more have been overlooked due to their non-trivial triggering conditions. Thus, we need a *proactive* bug exploitation method that allows the fuzzer to find these bugs within the explored program code.

3. Design

To overcome the limitations of state-of-the-art fuzzers, we propose the following improvements:

(1) a branch counting mechanism that combines collision-free and context-sensitivity, with an instrumentation removal algorithm to reduce memory overhead while maintaining accuracy,
(2) a predicate solver that adapts traditional optimization techniques designed for the real domain to bounded integer domain.

3.1. Collision-free context-sensitive branch counting

Following the common practice in fuzzing, we record the visit counts of branches and use them to approximate the state of the program. We designed our mechanism to be both context-sensitive and collision-free to improve the accuracy of the branch counting feedback. In current collision-free branch counting techniques, each branch is given a static unique ID b. Context-sensitive branch counting techniques generally use a context identifier c to differentiate between branches when appearing in different function contexts. Thus, we denote the tuple (c, b) as the context-sensitive branch. Our mechanism ensures that we record the visit count of each unique context-sensitive branch separately.

In contrast to AFL-derived branch counting mechanisms which use fixed-size branch counting tables, we wish to find the minimal space required for storing all the visit counts, allowing the fuzzer to adapt to any given program automatically. We achieve this in three steps. First, we identify all the unique context-sensitive branches. Then, in each function, we find the branches that do not need to be instrumented. Finally, for those branches that need instrumentation, we assign a unique sequential ID to each context-sensitive branch. This ID serves as the index of the branch in the branch-counting table.

3.1.1. Static branch edge ID generation

In contrast with AFL++'s approach of assigning a globally unique ID for each branch, we give each branch a relocatable function-local ID by visiting every function in the program, traversing the branch edges, and generating an incremental sequential ID statically for each branch. Then we collect the number of branches in each function. We also maintain an additional global *function offset* variable during runtime that is updated when calling or exiting a function. Thus the offset for each branch can be calculated by taking the function-local branch ID plus the function offset. Thus, we can dynamically calculate the unique branch ID using branch relocation depending on the specific function context.

3.1.2. Calculate the number of context-sensitive branches

Let F be the set of all functions in the program, $f \in F$ be a function, $branch\_count(f)$ be the number of branches in f, and $context\_count(f)$ be the number of different calling contexts of f. Then the amount of branches is

$$n = \sum_{f \in F} context\_count(f) \cdot branch\_count(f)$$

We calculate $branch\_count(f)$ through the control flow graph of f. Calculating $context\_count(f)$ is more involved:

To avoid the explosion of the number of calling contexts (e.g., caused by recursion), we consider only one-level context, i.e., the context is determined by the call site only. Thus, we can determine explicit call sites easily.

3.1.3. Indirect function call context generation

To assign function context offsets for indirect function calls, we must identify all possible functions an indirect call site can call. To determine implicit call sites precisely, we would need precise points-to analysis. However, that is both difficult and expensive (Emami et al., 1994; Steensgaard, 1996).

Therefore, to find possible function contexts within a reasonable amount of time, we employ our method of approximating all candidate values of function pointers in indirect call sites. First, we determine the number of branch table entries that are required for each function in each context by taking the maximum number of branches of all functions. Then, we iterate over all function declarations in the program or library and classify them according to their function prototypes. Next, we find all operations that take the address of any function and add the respective functions to the candidate list. Finally, we find all candidate functions for each indirect function call site with the same function prototype. We reserve the amount of branch table entries required for each context. The function base offset is resolved at runtime by matching the actual pointer value with all possible candidate values.

3.1.4. Calculate the ID of each context-sensitive branch

Conceptually, for each function, we reserve a contiguous region of IDs that can store all the context-sensitive branches in the function.

To implement this, during instrumentation,

- In each function f
  - for each branch b, we sequentially assign a *function-local ID*, $ID(b)$, starting from 0.
  - for each potential call site c, we sequentially assign a context ID, $ID_f(c)$, starting from 0.
- We arbitrarily assign an order to all the functions, and assign an *ID offset*, $offset(f)$, to each function in the following way: for each function f_i, we set its offset $offset(f_i) = offset(f_{i-1}) + context\_count(f_{i-1}) \cdot branch\_count(f_{i-1})$. We initialize $offset(f_0)$ as 0.

At runtime, the ID of the context-sensitive branch (c, b) in function f is:

$$offset(f) + ID_f(c) \cdot branch\_count(f) + ID(b)$$

3.1.5. Redundant branch instrumentation removal

The benefit of instrumentation removal is twofold. First, it allows us to shrink the branch count table's size, reducing the memory overhead. Besides, branch counting is a time-consuming job where the program has to calculate the offset, fetch the entry, and save the result. If we could reduce the number of reported branches without affecting the distinguishability, then we can use the reduced branch counting to same runtime by not reporting these edges. To that end, we wish the path after instrumentation removal to be distinguishable from a different path after compression. Here, we formally define path and distinguishability:

Definition 3.1 (*Path*)**.** For a program with a CFG, the set of all edges are E. A complete path is a sequence of edges between basic blocks that represents one execution of a program. A compressed path is a subsequence of a complete path where only edges in $E' \subset E$ are kept.

Definition 3.2 (*Distinguishability*)**.** Suppose we have two complete paths P and Q, and their compressed paths P' and Q'. P' and Q' are said to be distinguishable when $P = Q$ if and only if $P' = Q'$.

Table 1
Examples of path compressions in Fig. 1. Gray areas in column five means that we did not allocate memory for those edges. Notice how edge `f` and `h` *must* be executed, thus there is no need to instrument them.

Path	Compressed path	Compression rate	Uncompressed Matcher table								Compressed Matcher table							
			a	b	c	d	e	f	g	h	a	b	c	d	e	f	g	h
bfh	-	100%		1				1		1								
cefh	c	75%			1		1	1		1			1					
adbfgh	ag	66%	1	1		1		1	1	1	1						1	
cefggh	cgg	50%			1		1	1	2	1			1				2	
adadadbfggh	aaagg	55%	3	1		3		1	2	1	3						2	
adcefgh	acg	57%	1		1	1	1	1	1	1	1		1				1	
adcefggggh	acgggg	40%	1		1	1	1	1	4	1	1		1				4	

We do not need to instrument an edge if whether it is taken does not distinguish two different paths. This introduced two requirements for our instrumentation removal. First, for each loop, at least one edge needs to be instrumented. Otherwise, we would not distinguish how many times the loop has been executed. We use LLVM's definition of the loop[1] here and assume each loop has one and only one header block. Besides, for any basic block, exactly one of its outgoing edges needs no instrumentation. Because we can infer the status of that edge from other edges' status. For a basic block, if none of its instrumented outgoing edge is executed, then the only one that is not instrumented must be executed, and vice versa, if any instrumented edge is executed, then the edge without instrumentation is not executed. To satisfy both properties, we put labels on the edges before we instrument them. Algorithm 1 shows the algorithm.

For instance, in Fig. 1, we only need to instrument and record the visit counts of branches `a`, `c` and `g` to sufficiently distinguish different paths. Our algorithm would work in the following fashion to achieve this result. Initially, all edges are labeled as *delete*. We iterate over all loops' header block(A and C) first and mark the loops' outgoing edges(`a` and `g`) as *keep*. Then for each basic block, we have exactly one outgoing edge labeled as *delete* and mark others as *keep*. Thus only `c` is kept. Notice that whether we keep `c` or `b` does not change the branch table's size, nor the distinguishability of the instrumentation. We will prove this property in Theorem 1. Finally, we instrument all edges marked as *keep*, including branches `a`, `c` and `g`.

Table 1 shows how paths are compressed after our instrumentation optimization. Column four of the table shows the branch counting table if no compression is used. Each edge needs a counter to record how many time it has been executed. On the other hand, with our path compression we observe that only three edge need instrumentation, shortening our table size from eight entries to three entries, saving memory usage. Because many edges does not have a corresponding counter, they do not report their execution at runtime, lowering our runtime overhead. We find that the compression rate (column three) can be as high as 50%, that means we can save more than half of the runtime overhead. Finally, as we will prove in the following sections, the distinguishability is not affected by compression, which removes the necessity to convert the compressed path to the full path, since we can directly compare the compressed path and differentiate two execution traces.

We formally prove the algorithm's correctness:

Theorem 1. *Let P and Q be two paths. Let E be the set of all edges in the CFG, and $E' \subset E$ be the set of edges kept by Algorithm 1. Let P' and Q' be the compressed path of P and Q, respectively, generated from E'. Then $P = Q$ if and only if $P' = Q'$.*

Proof. **Necessity** (the right direction): Since $P = Q$, their subsequence on E' must also be equal.

[1] https://llvm.org/docs/LoopTerminology.html

Algorithm 1 Procedure for determining which branches to instrumentation in a function.

```
1:  function FindEdgesToInstrument(CFG)
2:      Mark all edges as delete.
3:      for Loop l ∈ CFG do
4:          h ← l's header block
5:          for Edge e = (h, b) ∈ h's outgoing edge do
6:              Mark edge (h, b) as keep.
7:      for Block b ∈ CFG do
8:          E = set of all outgoing edges of b
9:          if ∃e1 ≠ e2 ∈ E, both are marked as delete then
10:             ∀e ∈ E, mark e as keep
11:             mark e1 as delete
12:     Instrument all edges marked with keep
```

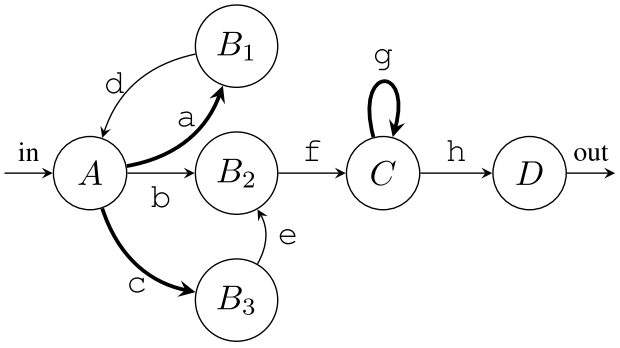

Fig. 1. Examples of branches that do not require instrumentation. Only thickened edges need instrumenting.

Sufficiency (the left direction): Prove by contradiction. Assume $P \neq Q$ but $P' = Q'$. Let $P = (A, p_1, \ldots)$, $Q = (A, q_1, \ldots)$, where A is the longest common prefix of P and Q. Therefore, p_1 and q_1 are different but they start from the same basic block B, so B must have $n > 1$ outgoing edges. Line 7–11 of Algorithm 1 guarantees that at least $n-1$ of the edges are marked *keep*, so at least one of p_1 and q_1 is marked as *keep*.

If both p_1 and q_1 are marked as *keep*, then they both appear in E', so $P' = (A', p_1, \ldots)$ and $Q' = (A', q_1, \ldots)$ where A' is the compressed path of A. Since $p_1 \neq q_1$, $P' \neq Q'$, but this contradicts our assumption.

If only one of p_1 and q_1 are marked as *keep*. Without loss of generality, let p_1 be marked as *keep*. So $P' = (A', p_1, \ldots)$. The assumption $P' = Q'$ implies that $Q = (A, q_1, B, p_1, \ldots)$, i.e., Q contains a cycle (q_1, B) and no edge in the cycle is marked as *keep*. But line 3–6 of Algorithm 1 prevented this. □

3.2. Compensated mutation assisted solver

While random mutation operators generally used by the AFL family of fuzzers can quickly solve "easy" predicates, predicates with a small

Table 2
Conversion table between branch predicate expressions, their corresponding objective functions and solver targets. δ represents the smallest possible positive value that the numerical type can represent. For integers, $\delta = 1$.

Predicate	Objective	Angora's constraint	Valkyrie's constraint
$a > b$	$f = b - a$	$f < 0$	$f < 0$
$a < b$	$f = a - b$	$f < 0$	$f < 0$
$a = b$	$f = a - b$	$\|f\| \leq 0$	$f = 0$
$a \geq b$	$f = b - a - \delta$	$f < 0$	$f < 0$
$a \leq b$	$f = a - b - \delta$	$f < 0$	$f < 0$
$a \neq b$	$f = a - b$	$-\|f\| < 0$	$f < 0$ or $f > 0$

feasible input space are difficult for them to solve, especially when the predicate is an equality comparison. In Listing 1, `num\_components` has 256 possibilities, thus 256 possible inputs to satisfy the comparison. However, there are 256^3 possible inputs for three bytes, making it difficult for fuzzers that randomly generate inputs. On the other hand, even state-of-the-art fuzzers with a solver may fail because their assumptions are not true.

Therefore, we need a new solver that properly handles the bounded integer domain that largely exists in real-world programs.

We use the notation $f(\mathbf{x})$ to represent its objective function for each predicate. $\mathbf{x}$ is a vector determined by a subset of the input bytes. The fuzzer maps input bytes to $\mathbf{x}$ using dynamic taint analysis tools like DataFlowSanitizer (Anon, 2023a). The range of each dimension of $\mathbf{x}$ is determined by its type, bit width, and signs, which Valkyrie computes by static analysis. For simplicity, we refer to the maximum and minimum value that can be represented by $\mathbf{x}_i$ as min_i and max_i.

f is a blackbox function determined by the predicate, as shown in Table 2. When the predicate becomes unreachable because a new input alters the program path, we set $f(\mathbf{x})$ to a value that violates the objective. For example, when the objective is $f(\mathbf{x}) < 0$, then we set $f(\mathbf{x}) = +\infty$.

The effectiveness of state-of-the-art predicate-solving fuzzers implies that many predicates in the program are solvable using principled methods. For example, Angora assumes that the objective functions of predicates are continuous, therefore it uses a gradient-descent-derived solver. However, program inputs usually take the form of byte values that are bounded and discrete. Therefore, solvers developed with a continuous range assumption require modifications to adapt to real-world situations.

We design a compensated mutation technique that mitigates this problem. The main idea of compensated mutation is when given a target step $\Delta\mathbf{x} \in \mathrm{R}^n$ that the solver wants to apply to the input, we find a $\Delta\mathbf{x}' \in \mathbb{Z}^n$ such that $f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}+\Delta\mathbf{x}')$. To do this, we clip the fractional values that could not be applied to integer values and *compensate* them to other components of the input vector. To demonstrate how this approach works and its effectiveness, we apply this technique to a gradient descent solver, albeit with some modifications.

3.2.1. Compensation from real domain to integer domain

Current methods resort to integer flooring when given a vector of fractional numbers $\Delta\mathbf{x} \in \mathbb{R}^n$ to apply to a vector of integer numbers. However, we cannot guarantee that the floored value $\lfloor\Delta\mathbf{x}\rfloor$ will result in a similar function value, especially when components have large coefficients in the function. To avoid precision loss due to rounding techniques of any kind, we wish to find an integer vector $\Delta\mathbf{x}' \in \mathbb{Z}^n$ such that $f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}+\Delta\mathbf{x}')$. The main idea behind the *compensated step* is that for a $\Delta\mathbf{x}$ as well as its *gradient* on the function, we traverse through each component, apply a suitable integer mutation value, and *compensate* the fractional values that were not applied into other components. We denote $\mathbf{r}_i$ as the amount that we intend to add to $\mathbf{x}_i$ and $\Delta\mathbf{x}'_i$ for the actual integer value that is added. The difference between $\mathbf{r}_i$ and $\Delta\mathbf{x}'_i$ is the value that needs to be compensated to another component of the input vector. We call this difference *carry amount* and use notation $\mathbf{c}_i$. Thus we have:

$$\mathbf{c}_i = \mathbf{r}_i - \Delta\mathbf{x}'_i$$

However, in many cases, it is not possible for the last dimension to take fractional values or reach its upper bound. To ensure that the function value remains the same, we introduce the concept of a "carry amount" ($\mathbf{c}_{i-1}$) to be added to the next dimension. If we applying the carry amount ($\mathbf{c}_i$) to the ith component, the objective function value should change by $\mathbf{c}_i\mathbf{g}_i$, where $\mathbf{g}_i$ represents the partial derivative of dimension i. However, since $\mathbf{c}_i$ is a fractional value that cannot be directly applied, we need to carry this amount over to dimension j. In order to maintain the same change in the function value, we should add another term of $\frac{\mathbf{c}_i\mathbf{g}_i}{\mathbf{g}_j}$ to the original value $\mathbf{x}_j$. As a result, the representation $\mathbf{r}_i$ of the ith dimension consists of two parts. The first part is the original value $\mathbf{x}_i$, and the second part is the carry amount from the last dimension, which is $\frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i}$. We can write the compensation process in Eq. (1):

$$\begin{aligned}
\mathbf{c}_0 &= 0 \\
\mathbf{r}_1 &= \Delta\mathbf{x}_1 \\
\mathbf{r}_i &= \Delta\mathbf{x}_i + \frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i} \\
\mathbf{c}_i &= \mathbf{r}_i - \Delta\mathbf{x}'_i
\end{aligned} \tag{1}$$

Finally, to obtain the integer value $\Delta\mathbf{x}'_i$, most of the time we use $\Delta\mathbf{x}'_i = \lfloor\mathbf{r}_i\rfloor$. This is different than $\lfloor\Delta\mathbf{x}_i\rfloor$. As shown in Eq. (1), $\mathbf{r}_i$ is the sum of the target value $\Delta\mathbf{x}_i$ and the amount carried over from the previous component $\mathbf{c}_{i-1}$ corrected by the fraction of gradients $\frac{\mathbf{g}_{i-1}}{\mathbf{g}_i}$. There are few exceptions where we do not floor $\mathbf{r}_i$:

(1) $\mathbf{x}_i + \lfloor\mathbf{r}_i\rfloor > max_i$. This means we could overflow this dimension, thus we set $\Delta\mathbf{x}'_i = max_i - \mathbf{x}_i$.
(2) $\mathbf{x}_i + \lfloor\mathbf{r}_i\rfloor < min_i$. Similarly, we set $\Delta\mathbf{x}'_i = min_i - \mathbf{x}_i$.
(3) The carry amount $\mathbf{c}_i$ is so large that all the dimensions will be overflown by it. In this case we try $\Delta\mathbf{x}'_i = \lceil\mathbf{r}_i\rceil$.

It is not hard to derive the following relation using calculus and Eq. (1):

$$\begin{aligned}
f(\mathbf{x}+\Delta\mathbf{x}') \approx f(\mathbf{x}) + \mathbf{g}^T\Delta\mathbf{x}' &= f(\mathbf{x}) + \sum_i \mathbf{g}_i(\mathbf{r}_i - \mathbf{c}_i) \\
&= f(\mathbf{x}) + \sum_i [(\mathbf{g}_i \cdot \Delta\mathbf{x}_i + \mathbf{g}_i \cdot \frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i}) - \mathbf{g}_i\mathbf{c}_i] \\
&= f(\mathbf{x}) + \sum_i \Delta\mathbf{x}_i\mathbf{g}_i - \mathbf{g}_n\mathbf{c}_n \\
&= f(\mathbf{x}+\Delta\mathbf{x}) - \mathbf{g}_n\mathbf{c}_n
\end{aligned} \tag{2}$$

Therefore, the loss of our method can be as low as $|\mathbf{g}_n\mathbf{c}_n|$. In practice, we use a permutation matrix to sort the components in the descending order of the absolute value of their gradients for the following reasons:

(1) Since in most cases $\mathbf{c}_{i-1} < 1$, we need $\frac{\mathbf{g}_{i-1}}{\mathbf{g}_i} > 1$, otherwise the compensation will not affect $\mathbf{r}_i$ too much.
(2) We also want $\frac{\mathbf{g}_{i-1}}{\mathbf{g}_i}$ to be as small as possible, so it would not amplify $\mathbf{r}_i$ too much that we have to push $\mathbf{x}'_i$ to its bound.
(3) As shown in Eq. (2), a small $|\mathbf{g}_n|$ would reduce the error incurred by compensated step.

The whole process is described in Algorithm 2. First, we sort the inputs based on the gradient. Then we calculate $\mathbf{r}_i$ for each dimension based on Eq. (1). We then choose $\Delta\mathbf{x}'$ based on $\mathbf{r}_i$ as described before.

This method is applicable to any solver that can obtain the *gradient* of each input component. The gradient can be obtained using a variety of methods, such as using white-box analysis and receiving an explicit expression, or through numerical methods to approximate the gradient. In our approach, we use a numerical estimation. In the following section, we describe our modifications for improved numerical differentiation in real-world fuzzing scenarios in the following part.

Algorithm 2 Compensated step

function CompensatedStep($\mathbf{x} \in \mathbb{Z}^n, \Delta\mathbf{x}, \mathbf{g} \in \mathbb{R}^n$)
  $P \leftarrow$ Permutation matrix s.t. $\forall i < j, |P\mathbf{g}_i| \geq |P\mathbf{g}_j|$
  $\mathbf{x} \leftarrow P\mathbf{x}$
  $\Delta\mathbf{x} \leftarrow P\Delta\mathbf{x}$
  $g \leftarrow Pg$ ▷ Sort dimensions in the descending order of the absolute value of the gradient
  $\mathbf{c}_0 \leftarrow 0, \mathbf{g}_0 \leftarrow 1$
  for i in 1..n **do**
    $\mathbf{r}_i \leftarrow \Delta\mathbf{x}_i + \frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i}$
    $\Delta\mathbf{x}'_i = \lfloor \mathbf{r}_i \rfloor$
    if $\mathbf{x}_i + \Delta\mathbf{x}'_i > max_i$ **then**
      $\Delta\mathbf{x}'_i = max_i - \mathbf{x}_i$
    else if $\mathbf{x}_i + \Delta\mathbf{x}'_i < max_i$ **then**
      $\Delta\mathbf{x}'_i = min_i - \mathbf{x}_i$
    else if $\mathbf{r}_i - \Delta\mathbf{x}'_i$ is too large for the rest dimensions **then**
      $\Delta\mathbf{x}'_i = \lceil \mathbf{r}_i \rceil$
    $\mathbf{c}_i = \mathbf{r}_i - \Delta\mathbf{x}'_i$
  return $P^{-1}\Delta\mathbf{x}'$

3.2.2. Compensated gradient descent

With the compensated step, here we modify the traditional gradient descent solver to tackle real-world scenarios. Although compensated step can be applied to any solvers, we find gradient descent better suited for our needs. Compensated step heavily rely on a gradient to work, which is the same for gradient descent.

Modified differentiation for a more accurate gradient. Since the predicates' mathematical expressions are unknown and we treat them as black-box functions, we cannot derive a gradient symbolically. However, the traditional differentiation method lack accuracy since a valid gradient's absolute value may be less than 1. For example $x = 5, f(x) = \lfloor x/4 \rfloor$, where flooring the result is the semantic of integer division in C programs. In this case, we find $f(x+1) = f(x) = f(x-1) = 1$ and end up with zero gradient. We need an approximated gradient instead of a zero gradient to keep the algorithm going.

Therefore, to obtain the partial gradients of a particular predicate, we use a modified numerical differentiation method on each dimension to derive a partial gradient. When calculating differentiation for dimension i, we create a unit vector $\mathbf{e_i} \in \mathbb{R}^n$ where only the ith element is 1 and all other elements is 0. We add and subtract $\mathbf{x}$ with this $\mathbf{e_i}$ and observe f's value change to derive a gradient.

Furthermore, we introduce amplifiers β_+ and β_- to increase the unit step size. β_+ and β_- starts with 1. We keep doubling β_+ and β_- until we find a non-zero $f(\mathbf{x} + \beta_+\mathbf{e_i}) - f(\mathbf{x})$ or $f(\mathbf{x}) - f(\mathbf{x} - \beta_-\mathbf{e_i})$. Then we can compute the gradient in the ith dimension, $\mathbf{g}_i$, using Eq. (3):

$$\mathbf{g}_i = \frac{f(\mathbf{x} + \beta_+\mathbf{e_i}) - f(\mathbf{x} - \beta_-\mathbf{e_i})}{\beta_+ + \beta_-} \tag{3}$$

If the amplifier β grows very significant without finding a practical value, we consider the gradient to be zero. β is considered large if $\beta > \frac{1}{2}(max_i - min_i)$. If both directions turn out to be zero, we assume this direction to have zero gradient. By repeating this process on all dimensions, we get a differentiation vector $\mathbf{g}$.

Determine the step size in descent. In the state-of-the-art solver, it takes a step $\Delta\mathbf{x} = -\alpha\mathbf{g}$ to descend in each iteration. However, it is challenging to set α. If we set it too small, $\mathbf{x}$ may move slowly or even stagnate. For example, $f(x) = \lfloor x/4 \rfloor$, if we move x by 1, $f(x)$ will not change. But if we set it too large, we may overshoot, causing the function to descend more than intended.

Therefore, we take the advantage of the fact that given a small step $\Delta\mathbf{x}$, f is approximately linear. There is an ϵ ball $B_\epsilon(\mathbf{x})$ such that for small enough $\epsilon \in \mathbb{R}$ such that given $\|\Delta\mathbf{x}\|_\infty < \epsilon$, $\mathbf{g}$ being the gradient, we have

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{g}^T\Delta\mathbf{x}$$

Algorithm 3 Descent routine

Require: f
function Descent($\mathbf{x}, \mathbf{g} \in \mathbb{R}^n$)
  $v \leftarrow max(1, \min_{i \text{ s.t. } \mathbf{g}_i \neq 0} |\mathbf{g}_i|)$
  $\alpha \leftarrow v/\mathbf{g}^T\mathbf{g}$
  $\mathbf{x}_{prev} \leftarrow \mathbf{x}$, $f_{prev} = f(\mathbf{x}_{prev})$,
  loop
    $\Delta\mathbf{x}' \leftarrow$CompensatedStep($\mathbf{x}_{curr}, -\alpha\mathbf{g}, \mathbf{g}$)
    $\mathbf{x}_{curr} \leftarrow \mathbf{x}_{curr} + \Delta\mathbf{x}'$, $f_{curr} = f(\mathbf{x}_{curr})$
    if $f_{curr} = \infty$ **or** $|f_{prev}| \leq |f_{curr}|$ **then** ▷ Next step does not exist or the function is not descending.
      return $\mathbf{x}_{prev}$
    else if IsSolved(f_{curr}) **then**
      return $\mathbf{x}_{curr}$
    $\alpha \leftarrow 2\alpha$, $\mathbf{x}_{prev} \leftarrow \mathbf{x}_{curr}$, $f_{prev} \leftarrow f_{curr}$

We select an α such that $f(\mathbf{x})$ will change approximately by the smallest possible increments or decrements.

$$\begin{aligned} v &= \max(1, \min_{\mathbf{g}_k \neq 0}(|\mathbf{g}_k|)) \\ \alpha &= \frac{v}{\mathbf{g}^T\mathbf{g}} \end{aligned} \tag{4}$$

If v is small, $f(\mathbf{x} - \alpha\mathbf{g}) - f(\mathbf{x}) \approx -v$ by Eq. (2). We introduced a minimum non-zero gradient $\mathbf{g}_k$ because if $|\mathbf{g}_k| > 1$, the minimal change possible to $f(\mathbf{x})$ is $|\mathbf{g}_k|$ instead of 1, since $f(\mathbf{x})$ is a discrete function. In each iteration, we double the step size to descend quicker. We revert the descent parameters to the initial state when we can no longer descend.

For non-linear functions, we could recalculate gradient in every step. However, since each dimension's gradient calculation requires us run the program under test a few times, calculating gradient is very expensive. We recognize that there are three possible outcomes for non-linear predicates. First, the execution path changes and the predicate is unreachable. In this case, we have no choice but to stop descending and use the value from the previous step as a result, a new gradient will be calculated later. Secondly, the function value may drop less than expected or even increase. We test if the new function value is still descending; if not, we return the previous step and recalculate the gradient. Finally, the function value may drop more than expected. Since our goal is to do gradient descent instead of keeping the function linear, we are fine with this step and keep going until we run into previous two cases.

The overall modified gradient algorithm is listed in Algorithm 3. We start by calculating the step size using Eq. (4). Then we would decide whether to ascend or descend based on the current status of the function. Once the actual step $\Delta\mathbf{x}$ is determined, we calculate the compensated step using Algorithm 2. Finally we apply the integer step.

3.2.3. Solving motivating example

In the case of the example in Listing 1, we first formalize it as "given $f(\mathbf{x}) = \mathbf{g}^T\mathbf{x} - 8$, find $\mathbf{x}_{eq}$, s.t. $f(\mathbf{x}_{eq}) = 0$" Suppose the input has been sorted by gradient, thus $\mathbf{g} = [256, -3, 1]$ and the initial point is $\mathbf{x}_{init} = [0, 1, 13]$, $f(\mathbf{x}_{init}) = 2$.

We start with $v = 1, \alpha = \frac{v}{\mathbf{g}^T\mathbf{g}}$, i.e. we try to decrease function's value by only 1. The first dimension will have $\mathbf{r}_1 = \Delta\mathbf{x}_1 = -\frac{\mathbf{g}_1}{\mathbf{g}^T\mathbf{g}}$. We find $\mathbf{x}_1$ is already 0 and cannot decrease more.

Thus we carry all the $\mathbf{r}_1$ to the next dimension, i.e. $\mathbf{c}_1 = \mathbf{r}_1 = -\frac{\mathbf{g}_1}{\mathbf{g}^T\mathbf{g}}, \Delta\mathbf{x}'_1 = 0$.

$\mathbf{c}_1$ is then applied to the next dimension, thus we have $\mathbf{r}_2$:

$$\begin{aligned} \mathbf{r}_2 &= \Delta\mathbf{x}_2 + \frac{\mathbf{c}_1\mathbf{g}_1}{\mathbf{g}_2} = -\frac{\mathbf{g}_2}{\mathbf{g}^T\mathbf{g}} - \frac{\mathbf{g}_1}{\mathbf{g}^T\mathbf{g}}\frac{\mathbf{g}_1}{\mathbf{g}_2} \\ &= -\frac{1}{\mathbf{g}_2}\frac{1}{\mathbf{g}^T\mathbf{g}}(\mathbf{g}_1^2 + \mathbf{g}_2^2) = -\frac{1}{\mathbf{g}_2}\frac{1}{\mathbf{g}^T\mathbf{g}}(\mathbf{g}^T\mathbf{g} - \mathbf{g}_3^2) \\ &= \frac{1}{3}(1 - \frac{1}{\mathbf{g}^T\mathbf{g}}) \end{aligned}$$

$\mathbf{r}_2$ is again floored to 0, leaving $\mathbf{c}_2 = \mathbf{r}_2, \Delta\mathbf{x}'_2 = 0$.

Interestingly, we have

$$\mathbf{r}_3 = \Delta\mathbf{x}_3 + \frac{\mathbf{c}_2\mathbf{g}_2}{\mathbf{g}_3} = -\frac{1}{\mathbf{g}^T\mathbf{g}} + \frac{\mathbf{g}_2}{\mathbf{g}_3}\frac{1}{3}(1 - \frac{1}{\mathbf{g}^T\mathbf{g}})$$
$$= -\frac{1}{\mathbf{g}^T\mathbf{g}} - (1 - \frac{1}{\mathbf{g}^T\mathbf{g}}) = -1$$

Therefore $\Delta\mathbf{x}'_3 = -1$ and we end up with $\Delta\mathbf{x}' = [0, 0, -1]$. This would give us $\mathbf{x} = [0, 1, 12]$, $f(\mathbf{x}) = 1$. Since the descent is successful, we would double the step size, i.e. set $v = 2$ and descent again. Following a similar process would give us $\mathbf{x} = [0, 1, 10]$, $f(\mathbf{x}) = -1$. Because the absolute value is not descending, we would abort the descent instead of taking the step. We calculate the gradient again and restart using $v = 1$. The final step would give us $\mathbf{x}_{eq} = [0, 1, 11]$, $f(\mathbf{x}_{eq}) = 0$.

3.3. Proactive bug exploitation

Current exploitation methods employed by state-of-the-art fuzzers such as AFL++ are merely best efforts, which are generally based on heuristics and magic byte insertions. These methods are ineffective when exploiting bugs that require non-trivial triggering conditions. Instead, we extend the predicate solver which is designed for program exploration into the domain of bug exploitation. We identify and designate *exploit predicates*, which are possible exploit points transformed into a predicate that the solver can handle. When exploit predicates are solved, they trigger a bug instead of explore a new path. The proactive bug exploitation process is divided into exploit point identification through static analysis and exploit predicate solving during fuzzing.

3.3.1. Exploit point identification

During static analysis, we identify susceptible instructions that have a probability of triggering a crash as *exploitation points*. For each possible exploitation site, we identify an *exploitable value*, i.e., a value that will trigger a crash at this point, where we instrument predicates for the solver to try triggering these bugs. Here, we select three exploitation cases where an architecture failure will be triggered:

Divide by zero. The relevant susceptible instruction takes the form of a division operation, specifically `result = dividend/divisor`. We instrument a predicate $divisor == 0$ so that the solver will try to move the divisor to zero. In practice, we find programmers will most likely check if divisor is zero, while forgetting the possibility that the divisor can *overflow* to zero. Therefore, we instrument another predicate $divisor == \text{MAX} + 1$.

Memory indexing. If the memory index is larger than the size of the buffer, there may be a buffer overflow. Although buffer size is hardly known at runtime, all we need is to guide the solver to push the index to a higher value. The solver either finds it impossible due to index checks in the program, or trigger a buffer overflow. For each indexing, we instrument a predicate idx > MAX.

Memory allocation. If the size of memory allocation is not sanitized properly, a memory corruption can happen. There are two possibilities. If the allocated memory is less than desired, then it may result in a future buffer overflow, which may happen when the argument overflows to a small value. For example, `malloc((uint8_t)(257))` only allocates 1 byte of memory instead of 257 due to integer overflow. On the other hand, large allocation size requests may result in resource exhaustion, leading to the program being killed or returning a null pointer that may not be properly sanitized. This can happen when malformed or malicious inputs are processed without proper checks within the program or an integer underflow, for example `malloc(10 - 12)`. Therefore, we instrument two predicates: size > MAX and size < 0. One predicate targets a very large allocation size while the other targets an integer underflow.

3.3.2. Exploration prioritized scheduling

We observe that if the buggy code cannot be reached by the fuzzer, then the bug cannot be triggered regardless of the resources spent on exploitation. Therefore, we prioritize exploration over exploitation so that we will have a better chance of triggering bugs. In each round of fuzzing, we always try to solve exploration predicates first, we only start trying exploitation predicates after we exhausted exploration predicates. We achieve this by always setting exploitation predicates with lower initial priority. Since state-of-the-art fuzzers are using priority queue to do scheduling, our approach brings no overhead to the fuzzer. What is more, this approach guarantees that the solver will attempt to solve exploration predicates before proceeding to trigger exploit points. In our experience fuzzing Magma dataset (Section 4.1), the solver will attempt on all exploit predicates at least once except for a few trials.

On the other hand, we realize that most exploit predicates are infeasible since each one of them represents a bug. Therefore, spending too many resources on exploitation in a fuzzing campaign is unwise. Because our solver is deterministic, we will discard an exploit predicate after one attempt, as we have no reason to believe the second attempt will work. One exception is that we may find a predicate with different initial points. Since some predicates can be non-convex, the solver will try to solve the same predicate with different initial points.

4. Evaluation

We are interested to know how well Valkyrie works in practice. We implemented Valkyrie to conduct a series of experiments to analyze the effectiveness of the entire fuzzer and individual components. We borrowed from Angora the dynamic taint tracking framework and instrumentation base code. We used the LLVM compiler framework for program analysis and instrumentation. However, the branch counting algorithm and the solver are independent of Angora's. The implementation of our branch counting mechanism uses *gllvm* (Anon, 2022) to consolidate the program's compiled LLVM IR into one module, allowing for full-program analysis. We have open-sourced Valkyrie, including all docker images and seeds used in evaluation to Github: https://github.com/organizations/ValkyrieFuzzer.

We propose the following research questions to help us understand the results and implications of our designs:

- **RQ1**: Is Valkyrie state-of-the-art? How does it fare on benchmarks such as Magma?
- **RQ2**: How does Valkyrie perform against similar fuzzers on real-world open-source programs?
- **RQ3**: Is our branch counting mechanism a better trade-off than that of AFL++ or Angora?
- **RQ4**: Is the solver assisted with compensated step better?
- **RQ5**: Can branch counting and solver contribute to bug finding in real world applications?

To answer these questions, we designed experiments to examine Valkyrie's performance on Magma and a select group of open-source programs. We then conducted two close examinations to address the latter two questions adequately.

First, we test Valkyrie on benchmark Magma v1.1 (Hazimeh et al., 2020), then on real-world programs. We intend to test Valkyrie on a more robust benchmark FuzzBench (Metzman et al., 2021), but Angora is not provided in the benchmark. The reason is that FuzzBench only allows programs to be compiled once, but Angora requires two compilations to generate two versions of binaries. For fairness of the testing, we borrow the framework from Unifuzz (Li et al., 2021) to test real-world programs. Each fuzzer runs in a containerized environment with *one core*. Each experiment lasted 24 h and was repeated ten times, as suggested by Hazimeh et al. (2020). In both experiments, we select AFL, AFL++, and Angora for comparison. We choose AFL as the reference fuzzer since it is a source of inspiration for many others. We also include

Table 3
The list of fuzzers we used in our evaluation. Included are their respective versions and the arguments we provided to invoke the fuzzer.

Package	Version	Arguments
afl	2.57b	`-m 2048 -t 1000+`
MoptAFL	commit `339a21e`	`-m 2048 -t 1000+`
aflplusplus	3.01a	`-m 2048 -t 1000+`
angora	commit `3cedcac`	`-M 2048 -T 1`

Table 4
The list of projects we used in our evaluation. Included are their respective versions, the binary we used and the arguments we provided to invoke the binaries.

Package	Version	Program	Arguments
libjpeg-ijg	v9d	*cjpeg*	@@
jasper	2.0.12	*imginfo*	-f @@
jhead	3.04	*jhead*	@@
binutils	2.35	*nm*	-C @@
binutils	2.35	*objdump*	-x @@
xpdf	4.00	*pdftotext*	@@
binutils	2.35	*readelf*	-a @@
binutils	2.35	*size*	@@
libpcap/tcpdump	1.9.1/4.9.3	*tcpdump*	-e -vv -nr @@
libxml	2.9.10	*xmllint*	@@

AFL++, which has merged many improvements and function enhancements developed for AFL. We enabled llvm_mode, with AFLfast's power scheduling (Böhme et al., 2019), MOpt's mutator (Lyu et al., 2019), and non-colliding branch counting for AFL++. Angora is also a solver-based fuzzer with similar design goals to Valkyrie. We intend to compare to one of Angora's successors Matryoshka (Chen et al., 2019). However, the tool is not available to us. Table 3 shows the fuzzers' versions and arguments, Table 4 shows the versions and arguments of targets we fuzzed.

4.1. Magma benchmark

To test whether Valkyrie is state-of-the-art, we would like to work on a benchmark with ground truth first. We examined Valkyrie's performance against other popular fuzzers on Magma v1.1 (Hazimeh et al., 2020). Magma is a collection of targets with real-world environments. It contains seven libraries and 16 binaries. Magma manually forward-ported these bugs in older versions to the latest versions. Unlike LAVA-M (Dolan-Gavitt et al., 2016) where all bugs are synthetic and magic byte comparison, Magma has a spectrum of bugs covering most categories in Common Weakness Enumeration (CWE). Magma contains 118 bugs in total. There are 15 integer errors, six of which are divide-by-zero, and 58 memory overflows. The rest 45 bugs include use-after-free, double-free, 0-pointer dereference, etc.

However, Angora is a coverage-guided fuzzer that is not designed to trigger bugs. We borrow ideas from Rong et al. (2020), Anon (2023b), for each potential bug, e.g. buffer overflow, we would insert a branch `if (ptr > buf_len)report();` so that Angora can see and solve the predicate. Therefore, for a fair comparison, we only tested on 15 integer errors and 58 memory bugs that can be converted to a predicate.

MoptAFL is also reported to be the best in the benchmark (Hazimeh et al., 2020), therefore we included MoptAFL in this evaluation. We used the version provided in the benchmark. We want to see how Valkyrie compares with the state-of-the-art fuzzers.

We list Valkyrie's performance on Magma in Fig. 2. We calculate the arithmetic mean number of bugs found per trial per day. However, state-of-the-art fuzzers rely on randomized methods, a bug found in one trial may not be triggered in the another. Therefore, we also list all the unique bugs found, including bug id and the time used to trigger it in Table 5. The time shown is the arithmetic mean time to trigger a bug. If the fuzzer did not trigger a bug, then the time to trigger is set to 24 h for that fuzzer. Therefore, for non-deterministic fuzzers, the mean time to trigger a bug becomes large when the bug is triggered only a few times. For example, AFL++ triggered the bug `AAH001` in a few minutes in only one trial, so the mean is 23 h across 10 trials.

Table 5
Average time used to trigger a bug in Magma. Bolded text shows the fastest to trigger a bug.

Bug ID	Valkyrie	angora	aflplusplus	moptafl	afl
AAH037	**15 s**	15 s	39 s	20 s	20 s
AAH041	**15 s**	15 s	1 m	33 s	21 s
JCH207	5 m	16 m	3 m	1 m	**53 s**
AAH055	4 h	8 h	27 m	**4 m**	43 m
AAH015	7 h	6 h	4 m	**1 m**	1 h
MAE016	**20 s**	–	1 m	1 m	3 m
AAH020	8 h	11 h	2 h	**23 m**	3 h
MAE008	**20 s**	–	6 h	27 m	5 m
AAH024	**15 s**	15 s	1 m	16 h	–
AAH045	49 s	**15 s**	15 h	3 h	–
MAE014	**20 s**	–	23 h	2 h	2 h
AAH032	5 h	21 h	1 h	**28 m**	–
MAE104	3 m	**2 m**	22 h	13 h	16 h
AAH014	20 h	5 h	**21 m**	21 h	14 h
AAH026	46 s	**40 s**	22 h	22 h	–
AAH007	**1 m**	2 m	22 h	–	–
MAE115	**9 h**	15 h	–	19 h	12 h
AAH017	**7 h**	–	21 h	10 h	20 h
JCH201	**4 h**	–	–	19 h	21 h
AAH001	**1 h**	–	23 h	–	–
AAH010	22 h	–	**9 h**	–	–

```
1  // AAH001
2  size_t row_factor_l = 1 + (png_ptr->interlaced? 6: 0)
3   + (size_t)png_ptr->width
4   * (size_t)png_ptr->channels
5   * (png_ptr->bit_depth > 8? 2: 1);
6  size_t row_factor = (png_uint_32)row_factor_l;
7  if (png_ptr->height > PNG_UINT_32_MAX/row_factor)
       {...}
8  // MAE014
9  char *dir_start = value_ptr + maker_note->offset;
10 int NumDirEntries = php_ifd_get16u(dir_start,
       ImageInfo->motorola_intel);
```

Listing 2: Two seemingly easy bugs AAH001 and MAE014 in Magma. Valkyrie can trigger this bug in seconds while other fuzzers can take hours.

Valkyrie finds 21 unique integer and memory errors in Magma, while AFL, ALF++, MoptAFL, and Angora found 14, 19, 18, and 14 errors, respectively. Overall, Valkyrie ranked #1 and found 10.5% and 50% more errors compared with AFL++ and Angora, respectively. We conduct the Mann–Whitney U test to obtain p-value between each pair of fuzzers and list the significant plot of Valkyrie in Fig. 3. Of 7 libraries, Valkyrie ranked #1 on libpng, libxml2, and poppler ($p < 0.001$ compared with #2); tied #1 on openssl and php ($p < 0.01$ compared with #3); tied #2 on libtiff. No fuzzer found any integer or memory errors on sqlite3. We want to emphasize that Valkyrie achieved the result with no randomization design.

Bug `AAH001` demonstrates that not only randomness is not required in certain bugs, but also that compensated steps can be effective in predicate solving. `AAH001` is a divide-by-zero in libpng. We listed the code snippet of `AAH001` in Listing 2. To trigger it the mutator must change `png_ptr->width` to `0x5555_5555` and `png_ptr->channels` to 3, and the later two conditions to false. Hazimeh et al. (2020) proved that it is hard for the randomized method to trigger it and claimed that only a fuzzer with a solver could trigger this easily. However, Angora failed to trigger it. When Angora mutates the value close to `0x5555_5555`, even a small step in `png_ptr->channels` will overshoot and overflow the result. For example, when setting `png_ptr->channels` to 4, the result will be a small value due to overflow; when setting to 2, the result will be a large value. Angora may conclude that this variable has negative gradient and start moving it to a smaller value. When

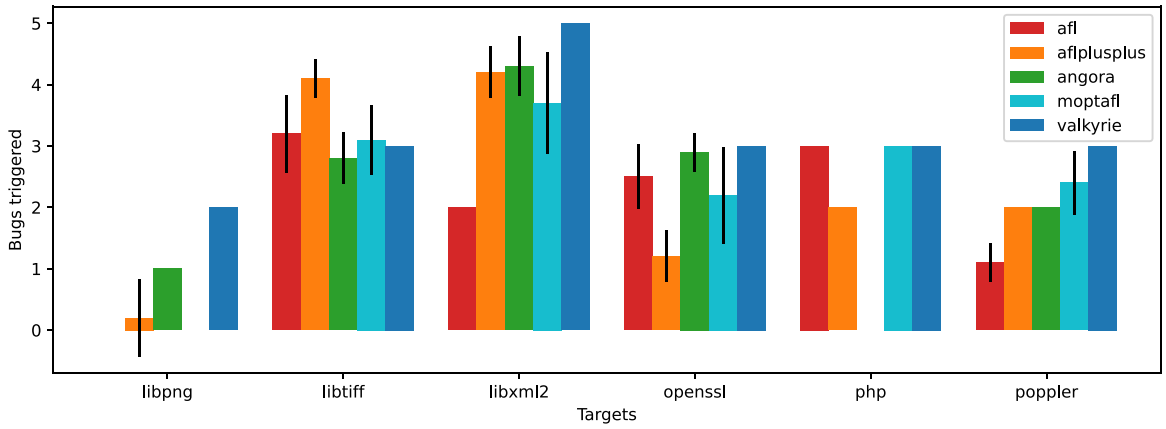

Fig. 2. Arithmetic mean of number of integer and memory bugs triggered per trial per day. The black line shows 95% confidence interval. Valkyrie's performance is the same across ten trials.

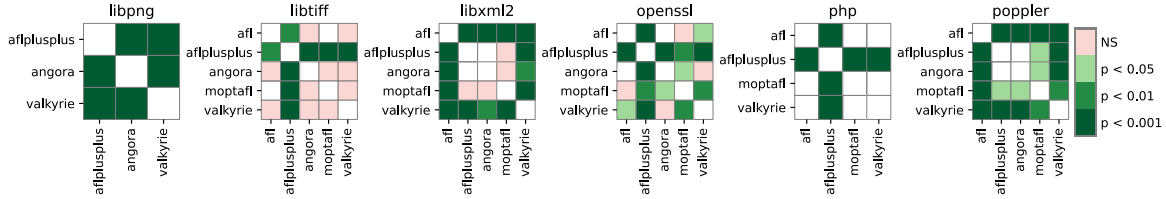

Fig. 3. Significant plot of Valkyrie. Valkyrie is superior than state-of-the-art on libpng, libxml2, and poppler.

it happens, Angora may get the wrong gradient and cannot progress correctly. However, Valkyrie knows the upper bound of the unsigned value and forces the solver not to exceed it using compensated steps. Thus Valkyrie is able to solve it and triggered this bug within the hour in all ten trials.

Valkyrie is also the first to find many of the bugs compared with its peers. The reason is that solver-based fuzzers work on predicates in a more orderly manner. Randomized methods can be choked by a predicate, not knowing if it is a hard one or just infeasible, wasting its time budget. However, Valkyrie can report with confidence whether the predicate can be solved and explore a new path or report it unsolvable. One example is `MAE014` as shown in Listing 2. `dir_start` is a pointer to the buffer and `php_ifd_get16u` tries to get a `u16` from the buffer. However, it does not check whether `dir_start` is pointing to the last byte of the buffer, causing the code to over-read one byte from the buffer. In the case of Valkyrie, it will try to increase the index of the read by setting up a predicate `dir_start+1 > MAX`, thus triggering the bug in 20 s. However, it generally takes fuzzers in AFL family hours to trigger it. Furthermore, these two examples demonstrate that our design in Section 3.3 is effective.

Valkyrie found four unique errors on libtiff (`AAH010`, `AAH014`, `AAH015`, and `AAH020`), the same number as other state-of-the-art fuzzers. However, on average, only three errors are triggered per trial because 24 h timeout is not enough for Valkyrie. The seeds corresponding to `AAH010` and `AAH014` are scheduled with the same priority. There is no guarantee which one is taken out first. In any trial, if one seed was taken, the other would not be taken before timeout. Thus the mean time to trigger these two bugs are both 20+ hours.

We want to comment on another interesting finding regarding MoptAFL and AFL++. MoptAFL is reported to be the best fuzzer in this benchmark, however, in our experiment, MoptAFL found fewer bugs than AFL++. We carefully compared Hazimeh et al. (2020)'s result with ours and find that, in our experiment, AFL++ found several bugs that were reported as untriggered. Some examples include `AAH001`, `AAH007` in libpng, both of which are only triggered once by AFL++ across ten trials. The difference is surprising considering we used the same configuration provided by Hazimeh et al. (2020). This further proves that randomized methods are volatile and unstable, while our deterministic approach is simpler and more reliable.

> In summary, Valkyrie found 21 unique integer and memory errors on Magma, the most compared with other state-of-the-art fuzzers. Also, Valkyrie had little to no variance across ten trials, while others showed unstable performance. Therefore, we can answer **RQ1** with confidence that Valkyrie is state-of-the-art on Magma.

4.2. Real-world open-source programs

While performing well on Magma is sufficient to claim Valkyrie is state-of-the-art, we would like to evaluate on real-world programs and see the branch coverage data. Therefore, to demonstrate Valkyrie's effectiveness on real-world programs already in production, we selected a series of open-source programs to evaluate Valkyrie and demonstrate the effectiveness of its methods and techniques in real-world situations. Of these open-source programs, there are image processors (*jhead*, *imginfo*), binary file processing programs (*nm*, *objdump*, *size*, *readelf*), structured text parsing utilities (*xmllint*), pdf parsers(*pdftotext*), network utilities(*tcpdump*). Because different tools count branches differently, for fairness of comparison, all branch coverage reported are generated by afl-cov (Anon, 2018).

The results of these experiments are shown in Fig. 4. We obtain p-value between each pair of fuzzers using Mann–Whitney U test. Valkyrie ranked #1 on seven out of ten applications ($p < 0.01$ compared with #2), #1 tied with Angora ($p = 0.0011$ compared with #3) on *jhead*, #2 on *cjpeg* and *imginfo* ($p < 0.05$ compared with #3).

We also plotted the branch coverage against time in Fig. 5. Valkyrie is the fastest fuzzer in all programs except *imginfo*, i.e., Valkyrie spends less time to reach the same branch coverage compared with other fuzzers. This trend is clearest in *objdump*, *readelf*, and *size*. It further demonstrated the effectiveness of deterministic algorithms we introduced in branch counting and solver. While state-of-the-art fuzzers are mutating randomly without knowing the detail of the program, Valkyrie can flip a predicate within several steps.

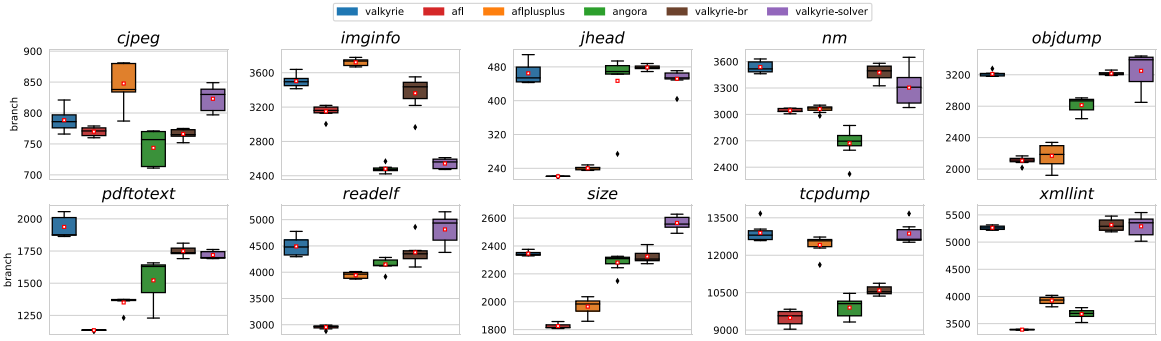

Fig. 4. Branch coverage of six fuzzers in 24 h time. Valkyrie-br is Valkyrie with only branch coverage improvement, Valkyrie-solver is Valkyrie with only solver improvement. Both design increased branch coverage compared with Angora in all programs. Overall, Valkyrie ranked #1 on geometric mean number of branches reached.

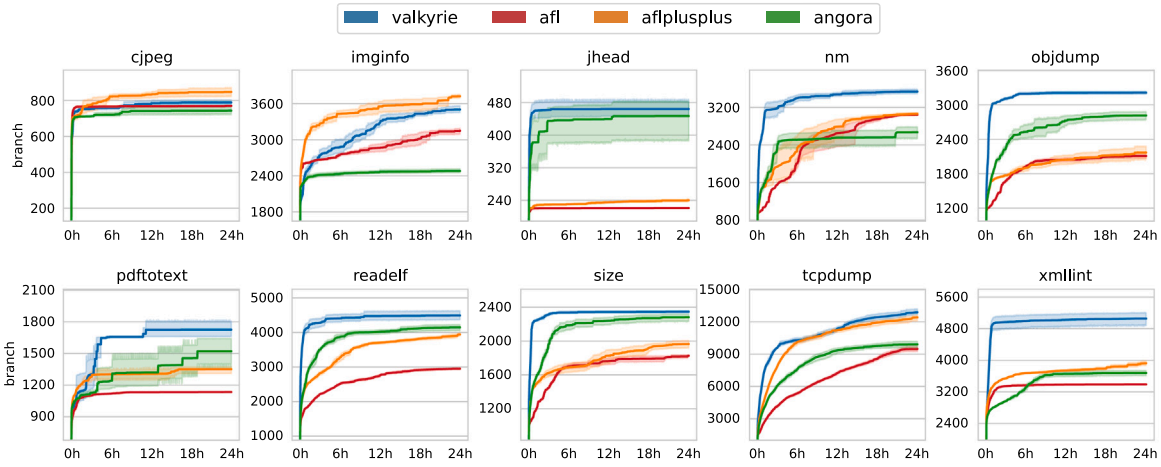

Fig. 5. Branch coverage of four fuzzers in 24 h time. Valkyrie not only finds more branch coverage but also is the fastest one on eight of ten applications thanks to deterministic algorithms.

In summary, the geometric mean number of branches Valkyrie reached per target is 2452, 8.2% and 12.4% more than AFL++ (2266) and Angora (2181), respectively. We can answer **RQ2** with confidence that Valkyrie is the state-of-the-art on real-world open-source programs.

4.3. Effectiveness of deterministic branch counting

We wish to understand the advantages of Valkyrie's branch counting mechanism quantitatively. We first controlled the variable to see how much improvement collision-free context-sensitive branch counting design contributes. Therefore, we disabled our improved solver and compared it with Valkyrie and Angora. The result is shown in Fig. 4, the modified version is labeled as Valkyrie-br. We find that Valkyrie-br outperformed Angora in all cases, proving that this design is effective. Our study shows the improvement is contributed by two designs: branch instrument optimization and context-sensitive collision-free branch counting.

We first examined the effectiveness of our branch table optimization strategies by obtaining the buffer sizes required by Valkyrie, as shown in Column 2–4 in Table 6. We observe that our optimization strategies can reduce the bitmap size by 69% on average. We used seeds generated by AFL++ to evaluate how much runtime is reduced. Column 5–7 in Table 6 show that we reduced runtime by 28% on average. Thus, given the same amount of time, Valkyrie can test the program more.

We then analyzed the buffer utilization rates of AFL and Angora under the evaluated programs. By default, AFL uses a 64 K buffer. Angora uses 1M to allow context-sensitivity. The utilization rate is shown in Columns 2 and 4 in Table 7. Many programs' utilization rates exceed the recommended limit of 4%, even ranging up to nearly 34%, indicating that a newly found branch has a nearly 34% chance of colliding with existing branches. Under the default settings, many instances have a high potential for branch collisions, as evidenced by the high bitmap utilization rate of up to around 36%. Therefore, the default buffer sizes are too small for ordinary programs.

We further resized their bitmaps according to the size required by Valkyrie to achieve collision-free branch counting and analyzed their utilization rates. Their bitmap sizes should be a strict power of 2, so we found the closest value possible for each program, as listed in Column 7 in Table 7. We list the utilization rate under such sizes Column 3 and 5 in Table 7. The utilization rates have dropped to under 4% since we increased AFL's buffer size for most programs. However, AFL lacks context-sensitivity and can potentially lose the capability to identify branches that increase the overall coverage. Angora, on the other hand, still exceeds the recommended limit greatly in many cases, resulting in significant accuracy loss. In comparison, Valkyrie guarantees accuracy while maintaining context-sensitivity, which is the second reason why the branch coverage increased in Fig. 4.

Therefore, we can answer **RQ3** with confidence that Valkyrie's branch counting mechanism is a better trade-off and outperforms that of comparable fuzzers.

Table 6
Bitmap size for Valkyrie before and after optimization. On average we reduced 69% of all instrumentations and 28% of runtime.

Program	Valkyrie bitmap size (B)			Valkyrie bitmap runtime (µs)		
	Original	Optimized	Reduction	Original	Optimized	Reduction
cjpeg	254 874	74 576	70.74%	10 331	7918	23.35%
imginfo	133 010	34 690	73.92%	20 769	12 583	39.41%
jhead	13 620	4396	67.72%	1124	776	30.92%
nm	1 758 594	542 688	69.14%	1491	1270	14.84%
objdump	2 196 528	691 048	68.54%	1405	1374	2.24%
pdftotext	400 858	112 808	71.86%	6312	5663	10.29%
readelf	353 222	132 352	62.53%	1229	902	26.57%
size	1 750 206	540 180	69.14%	1687	1359	19.44%
tcpdump	1 554 400	506 468	67.42%	1278	972	23.93%
xmllint	3 323 032	996 220	70.02%	1439	1115	22.52%
Total	11 738 344	3 635 426	69.03%	47 065	33 932	27.90%

Table 7
Bitmap utilization for AFL and Angora on open-source programs. We evaluated their respective utilizations under default sizes and adjusted sizes. "*" indicates failure, AFL refuses to run *jhead* with only 8 K bitmap.

Program	AFL utilization		Angora utilization		Bitmap size (B)	
	Default (64 K)	Adjusted	Default (1.0 M)	Adjusted	Valkyrie	Adjusted
cjpeg	2.11%	1.06%	0.24%	1.88%	74 K	128 K
imginfo	10.23%	10.30%	1.68%	23.94%	34 K	64 K
jhead	0.45%	*	0.54%	49.51%	4.2 K	8.0 K
nm	7.92%	0.49%	33.14%	33.14%	542 K	1.0 M
objdump	5.26%	0.33%	24.98%	24.96%	691 K	1.0 M
pdftotext	3.30%	0.83%	18.88%	56.67%	112 K	256 K
readelf	10.92%	2.73%	4.05%	15.24%	132 K	256 K
size	4.49%	0.28%	14.75%	14.72%	540 K	1.0 M
tcpdump	20.85%	2.59%	34.64%	57.13%	506 K	512 K
xmllint	6.51%	0.41%	18.30%	18.29%	996 K	1.0 M

4.4. Effectiveness of deterministic solver

In Fig. 4, we evaluated Valkyrie with only solver enabled. The modified version is tagged as Valkyrie-solver. Since Valkyrie-solver and Angora have the same scheduling algorithm and branch counting method, comparing them will tell us how much improvement our solver had.

The result shows that we improved branch coverage compared with Angora in *all* open source programs. We obtained *p*-value for each program using Mann–Whitney U test, all of them showing less than 0.02 except for *jhead*, where branch coverage is statistically insignificant. On geometric mean, Valkyrie-solver reached 2608 branches, 19.5% more than Angora (2181 branches). On *readelf* and *size*, Valkyrie-solver even ranked #1 compared with all other fuzzers.

> On average, Valkyrie-solver can execute more branches than Angora. This gives us a positive answer to **RQ4**, the compensated step does improve the solver performance.

4.5. Bug finding ability of valkyrie

In Section 4.1 we find that Valkyrie can find most memory and divide by zero errors compared to the state-of-the-art. Although examples like `AAH001` have demonstrated solver's effectiveness, we wish to carry out a more detailed study to understand each components individual contribution to bug finding in real-world settings. We first instrument programs in Unifuzz (Li et al., 2021) using the approach described in Section 3.3, then compiled these programs using Angora, Valkyrie, Valkyrie-br, and Valkyrie-solver. Similar to previous evaluations, we run each fuzzer for 24 h and ten times. After fuzzing we collect *all* errors the fuzzers found, deduplicate them and found the following bugs in three programs. The detailed bugs are listed in Table 8.

We find that Valkyrie is able to find six bugs while Angora only found three. Valkyrie-br and Valkyrie-solver found four and five bugs respectively. The main difference comes from three assertion failures in *imginfo*. The first and second assertion failure are `qmfbid == JPC_COX_RF` and `absstepsize >= 0`, which can be solved by magic byte matching quickly, thus all fuzzers triggered it. However, the third assertion failure is `!((expn + (numrlvls - 1) - (numrlvls - 1 - ((bandno > 0) ? ((bandno + 2) / 3) : (0)))) & (~0x1f))`, which involves three variables and a nested condition. Such predicate requires that the last five bits of the result are not all zeros. Tradition solvers like Angora's struggle to calculate the gradient when it reaches the boundaries, thus unable to solve it. On the other hand, only Valkyrie and Valkyrie-br triggered an out of bound read. After some study we find that the program attempt to access a buffer without checking the index, Listing 3 We find that using Angora's branch counting, the loop back edge collided with other edge. When the other edge is executed repeatedly, Angora had no motivation to increase the iteration of the loop, since to Angora's eyes this has already been executed. Since there is no collision in our branch counting, Valkyrie will try to increase the number of iteration until its greater than 128, a heuristics number set by Anon (2014), Fioraldi et al. (2020).

```
1  int Catalog::countPageTree(Object *pagesObj) {
2    for (i = 0; i < kids.arrayGetLength(); ++i) {
3      kids.arrayGet(i, &kid);    // Access without check
4      n2 = countPageTree(&kid);
5      if (n2 < INT_MAX - n) {
6        n += n2;
7      } else {
8        error(errSyntaxError, -1, ...);
9        n = INT_MAX;
10     }
11     kid.free();
12   }
13 }
```

Listing 3: Code snippet copied from xpdf. The program accesses the array without checking the bound.

> In summary, Valkyrie found six unique errors in three programs, ranked number one compared to Angora and other variants of Valkyrie. We can answer **RQ5** with confidence that both deterministic branch counting and solver contributed to Valkyrie's bug finding capability.

4.6. Summary

In the previous sections, we have addressed all research questions. Our results show that Valkyrie triggers 21 unique integer and memory errors, 10.5% and 50% more than AFL++ and Angora, respectively. In real-world programs, Valkyrie reached 2431 branches per target on average, 8.2% and 12.4% more compared with AFL++ and Angora, respectively. We demonstrated that our branch counting mechanism is a better solution for efficient and accurate feedback. Finally, we demonstrated that our predicate solving algorithms works effectively on real-world branch predicates, allowing Valkyrie to perform better than the other fuzzers we use for evaluation. Thus we claim that Valkyrie, which utilizes accurate and efficient feedback and effective predicate solving, is principled and reliable.

5. Discussion

5.1. Unsolved predicates

Previous sections demonstrated the effectiveness of our solver. However, there are scenarios where our solver experience difficulty when solving branch predicates.

Table 8
Bugs found by Valkyrie and Angora. Valkyrie found six bugs in three programs while Angora only found three.

Program	Description	Bugs found by each fuzzer			
		Angora	Valkyrie	Valkyrie-br	Valkyrie-solver
cjpeg	Floating point exception	✔	✔	✔	✔
imginfo	Assertion failure-1 (`qmfbid == JPC_COX_RF`)	✔	✔	✔	✔
imginfo	Assertion failure-2 (`absstepsize >= 0`)		✔	✔	✔
imginfo	Assertion failure-3 (Check Section 4.5 for predicate details.)		✔		✔
pdftotext	Throwing `GMemException`	✔	✔		✔
pdftotext	Out of bound read		✔	✔	

The solver is designed to solve single predicates. One possible cause would be unsolvable predicates guarding dead code, such as redundant error checks. The solver would also have difficulty solving some non-convex predicates, i.e., its local minima are not the global minimum.

If the predicate is nested, the solver may mutate the input and modify the outcome of its parent predicate(s), rendering the target predicate itself unreachable. We can solve this by applying Matryoshka's framework for solving nested branch predicate(s) (Chen et al., 2019).

On the other hand, Valkyrie relies on DFSan (Anon, 2023a), which can be slow when input space is large. While deterministic methods are more predicable and stable, the extra workload may prevent it from scale to large applications. Therefore, we would suggest a combined approach where we can use Valkyrie to explore hard to trigger predicates, and use AFL++ to explore the code.

5.2. Bug detection

We may have missed bugs in Magma due to the following reasons. Apart from the aforementioned issues, one main reason is that our work focus on increasing program coverage, our exploitation instrumentation only targets a small subset of bugs. Many common problems such as null pointer dereference, double free, use after free, etc. are not in the scope of this paper. However, we managed to find more bugs in 3 libraries in Magma and improved branch coverage by 12.4% compared to Angora. This fact further proved that Valkyrie is a reliable tool.

5.3. Branch counting effectiveness

In Fig. 4, we find that Valkyrie-solver can reach more branches than Valkyrie in rare cases, e.g. *readelf*. Although the only difference between Valkyrie-solver and Valkyrie is the our branch counting method, this does not suggest our method is less effective. Valkyrie-solver performed better because branch counting with branch collisions may miss many branches. These missed branches have two-sided effects. On the one hand, there may be key branches that lead to more coverage, thus limiting solver's ability. On the other hand, some difficult conditions are not generated in the first place, thus saving fuzzer's time. When the former effect is in dominance, Valkyrie will outperform Valkyrie-solver, vice versa. These two-sided effects are neither predictable nor desirable, which further justifies our motivation to eliminate branch collisions.

6. Related work

6.1. Branch counting methods

Since AFL, much work has been devoted to strike a balance between branch counting sensitiveness and the probability of colliding. Angora (Chen and Chen, 2018) updated AFL's method by adding a function context to the branch counting table. CollAFL (Gan et al., 2018) proposed replacing AFL's random ID generation with one that would largely prevent duplicate edge IDs from occurring. However, its method is subject to the bitmap size exceeding the number of branches in the target program and cannot integrate context-sensitivity easily like Valkyrie did. Recent work (Wang et al., 2019a) points out that branch counting is a trade off. Fuzzers benefit from sensitive branch counting algorithms, yet the more sensitive it is, the more computing budget is consumes. Angora uses an enlarged branch counting table to allow context sensitivity. However, that brings substantial memory overhead to the fuzzing process. This problem is sloved by Valkyrie by making branch counting collision free and reducing the size of matcher table.

6.2. Predicate solving methods

Many work focuses on predicate solving. Solver-based fuzzers aim at better solvers on branch predicates to reach high code coverage. REDQUEEN (Aschermann et al., 2019) solves hashes and checksums through input-to-state-correspondence. KLEE (Cadar et al., 2008) uses symbolic execution to solve predicates in the program to generate seeds, but symbolic execution can be ineffective when program path is deep and nested. This made KLEE ineffective compared to Valkyrie. Angora (Chen and Chen, 2018) solves branch predicates using principled methods such as gradient descent, yet in this paper we show that without continuous assumption Angora's solver may fail some simple cases. Angora's method is largely based on mathematical optimization through gradient descent. However, Angora cannot effectively solve branches that are nested together. Matryoshka (Chen et al., 2019) proposes procedural methods for solving nested constraints in real-world situations. Matryoshka solves nesting branches by a slightly modified gradient solver, which is ad hoc and unjustifiable.

6.3. Targeted fuzzers

Targeted fuzzers attempt to target the potentially buggy code. AFLGo (Böhme et al., 2017) proposed to reach the target code by giving priority to those close to the target. IJON (Aschermann et al., 2020) tries to manually take out some "important" program points and ask the fuzzer to put more time on it. However, unlike Valkyrie which can identify potential buggy code automatically, both AFLGo and IJON requires a human expert to label the target. IntEgrity (Rong et al., 2020) specifically target on integer errors and automatically identifies them using static analysis. Unlike Valkyrie who targets integer errors and memory errors, IntEgrity only targets integer errors, limiting its scope. Savior (Chen et al., 2020) also targets potential bugs, but it focuses on using seed scheduling to find those that lead to potential buggy code. TOFO (Wang et al., 2020) proposed a method to calculate the distances between all basic blocks in seed and target basic block and reaches its target by always selecting the closest seed. Both Savior and TOFO focuses on scheduling to reach targeted code faster and neglected exploration part of the fuzzing. Therefore, their tool is useful when reaching target for exploitation is more important than exploration. Besides, all the tools use randomized approach to reach a target, while only Valkyrie deals this by using a new solving method.

6.4. Machine learning based fuzzers

Machine learning has become more and more popular in various areas. There have also been many attempts to incorporate it into fuzzers. She et al. (2018) attempts to use a neural network to smooth the predicate to predicate a gradient when it cannot be calculated. Wang et al. (2017) uses data driven method to learn the input format from valid seeds to generate other seeds. Liu et al. (2019) uses generative model to generate C programs to fuzz compilers. However, their work only tests C compiler, which is a very narrow scope. Large language models (LLM) have also started to play a role in the fuzzing community. (Deng et al., 2023; Xia et al., 2023; Zhao et al., 2023; Hu et al., 2023) seek to use LLM to generate or mutate inputs for fuzzing. On the other hand, (Wang et al., 2023) attempts to reach more code by using an LLM to select program arguments. However, we believe machine learning are black boxes that cannot be reasoned. While these approaches trump in fuzzing, a more deterministic base line is needed to provide a reasonable baseline.

7. Conclusion

In this paper, we identify the challenges that state-of-the-art mutation-based greybox fuzzers face when finding vulnerabilities in real-world scenarios and propose our solution to address these issues. State-of-the-art fuzzers cannot achieve better performance mainly due to the following reasons: (1) they lack accurate and fine-grained branch counting feedback, and (2) their respective mutation strategies are not well-suited to real-world scenarios. We propose Valkyrie, a prototype fuzzer to address these issues. First, Valkyrie implements collision-free context-sensitive branch counting, which eliminates branch collision while capable of preserving context-sensitivity. Second, Valkyrie implements a predicate solver for fuzzing that adapts optimization algorithms for the real domain to the integer domain. Finally, we use the solver to help us trigger bugs by converting potentially exploitable code into predicates.

We evaluated Valkyrie on the Magma benchmark as well as real-world programs. Our results show that Valkyrie triggers 21 unique integer and memory errors, 10.5% and 50% more than AFL++ and Angora, respectively. In real-world programs, Valkyrie's branch counting mechanism proved effective by eliminating branch collisions and keeping context-sensitivity, while AFL and Angora incur high bitmap utilization rates, indicating significant branch collision probabilities. For coverage statistics, Valkyrie reached 8.2% more branches on average compared with AFL++, and 12.4% compared with Angora.

CRediT authorship contribution statement

Yuyang Rong: Methodology, Software, Validation, Data analysis, Writing – original draft, Writing – review & editing, Visualization. **Chibin Zhang:** Validation, Data analysis, Visualization. **Jianzhong Liu:** Methodology, Software, Validation, Data analysis, Writing – original draft. **Hao Chen:** Conceptualization, Methodology, Writing – review & editing, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Yuyang Rong reports financial support was provided by National science Foundation.

Data availability

Data will be made available on request.

Acknowledgment

This material is based upon work supported by the National Science Foundation, United States under Grant No. 1801751 and 1956364.

References

Anon, 2014. American fuzzy lop, URL http://lcamtuf.coredump.cx/afl/.
Anon, 2018. afl-cov, URL https://github.com/mrash/afl-cov.
Anon, 2022. gllvm: Whole Program LLVM in Go, URL https://github.com/SRI-CSL/gllvm.
Anon, 2023. DataFlowSanitizer, URL https://clang.llvm.org/docs/dataflowsanitizer.
Anon, 2023. LLVM Undefined Behavior Sanitizer, URL https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
Aschermann, Cornelius, et al., 2019. REDQUEEN: Fuzzing with input-to-state correspondence. In: NDSS, Vol. 19. pp. 1–15.
Aschermann, Cornelius, et al., 2020. Ijon: Exploring deep state spaces via fuzzing. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1597–1612.
Böhme, M., Pham, V., Roychoudhury, A., 2019. Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. 45 (5), 489–506. http://dx.doi.org/10.1109/TSE.2017.2785841.
Böhme, Marcel, et al., 2017. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344.
Cadar, Cristian, Dunbar, Daniel, Engler, Dawson R., et al., 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, Vol. 8. pp. 209–224.
Chen, Peng, Chen, Hao, 2018. Angora: efficient fuzzing by principled search. In: IEEE Symposium on Security and Privacy. SP, San Francisco, CA.
Chen, Peng, Liu, Jianzhong, Chen, Hao, 2019. Matryoshka: Fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. CCS '19, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450367479, pp. 499–513. http://dx.doi.org/10.1145/3319535.3363225.
Chen, Jiongyi, et al., 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In: NDSS.
Chen, Yaohui, et al., 2020. Savior: Towards bug-driven hybrid testing. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1580–1596.
Deng, Yinlin, et al., 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. arXiv:2212.14834 [cs.SE].
Dolan-Gavitt, Brendan, et al., 2016. Lava: Large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 110–121.
Emami, Maryam, Ghiya, Rakesh, Hendren, Laurie J., 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. PLDI '94, Association for Computing Machinery, New York, NY, USA, ISBN: 089791662X, pp. 242–256. http://dx.doi.org/10.1145/178243.178264.
Fioraldi, Andrea, et al., 2020. AFL++: Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies. WOOT 20, USENIX Association.
Gan, S., et al., 2018. CollAFL: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy. SP, pp. 679–696. http://dx.doi.org/10.1109/SP.2018.00040.
Hazimeh, Ahmad, Herrera, Adrian, Payer, Mathias, 2020. Magma: A ground-truth fuzzing benchmark. Proc. ACM Meas. Anal. Comput. Syst. 4 (3), 1–29.
Hu, Jie, Zhang, Qian, Yin, Heng, 2023. Augmenting greybox fuzzing with generative AI. arXiv:2306.06782 [cs.CR].
Jeong, Dae R., et al., 2019. Razzer: Finding kernel race bugs through fuzzing. In: 2019 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 754–768.
Klees, George, et al., 2018. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 2123–2138.
Li, Yuwei, et al., 2021. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: 30th USENIX Security Symposium. USENIX Security 21, USENIX Association.
Liu, Xiao, et al., 2019. DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing. In: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence. In: AAAI'19/IAAI'19/EAAI'19, AAAI Press, ISBN: 978-1-57735-809-1, http://dx.doi.org/10.1609/aaai.v33i01.33011044.
Liu, Baozheng, et al., 2020. {FANS}: Fuzzing android native system services via automated interface analysis. In: 29th {USENIX} Security Symposium. {USENIX} Security 20).
Lyu, Chenyang, et al., 2019. MOPT: Optimized mutation scheduling for fuzzers. In: 28th USENIX Security Symposium. USENIX Security 19, USENIX Association, Santa Clara, CA, ISBN: 978-1-939133-06-9, pp. 1949–1966, URL https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.
Metzman, Jonathan, et al., 2021. FuzzBench: An open fuzzer benchmarking platform and service. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA.

Rong, Yuyang, Chen, Peng, Chen, Hao, 2020. Integrity: Finding integer errors by targeted fuzzing. In: International Conference on Security and Privacy in Communication Systems. Springer, pp. 360–380.

She, Dongdong, et al., 2018. Neuzz: Efficient fuzzing with neural program learning. arXiv preprint arXiv:1807.05620.

Steensgaard, Bjarne, 1996. Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '96, Association for Computing Machinery, New York, NY, USA, ISBN: 0897917693, pp. 32–41. http://dx.doi.org/10.1145/237721.237727.

Wang, Zi, Liblit, Ben, Reps, Thomas, 2020. TOFU: Target-orienter fuzzer. arXiv preprint arXiv:2004.14375.

Wang, Junjie, et al., 2017. Skyfire: Data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy. SP, pp. 579–594. http://dx.doi.org/10.1109/SP.2017.23.

Wang, Jinghan, et al., 2019a. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses. {RAID} 2019), pp. 1–15.

Wang, Xueqiang, et al., 2019b. Looking from the mirror: evaluating IoT device security through mobile companion apps. In: 28th {USENIX} Security Symposium. {USENIX} Security 19, pp. 1151–1167.

Wang, Dawei, et al., 2023. CarpetFuzz: Automatic program option constraint extraction from documentation for fuzzing. In: 32nd USENIX Security Symposium. USENIX Security 23, USENIX Association, Anaheim, CA, ISBN: 978-1-939133-37-3, pp. 1919–1936, URL https://www.usenix.org/conference/usenixsecurity23/presentation/wang-dawei.

Xia, Chunqiu Steven, et al., 2023. Universal fuzzing via large language models. arXiv:2308.04748 [cs.SE].

Xu, Wen, et al., 2019. Fuzzing file systems via two-dimensional input space exploration. In: 2019 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 818–834.

Xu, M., et al., 2020. Krace: Data race fuzzing for kernel file systems. In: 2020 IEEE Symposium on Security and Privacy. SP, pp. 1643–1660.

Yun, Insu, et al., 2018. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium. USENIX Security 18, USENIX Association, Baltimore, MD, ISBN: 978-1-931971-46-1, pp. 745–761.

Zeller, Andreas, 2019. When results are all that matters: The case of the angora fuzzer | andreas zeller. URL https://andreas-zeller.info/2019/10/10/when-results-are-all-that-matters-case.html.

Zhao, Jianyu, et al., 2023. Understanding programs by exploiting (fuzzing) test cases. arXiv:2305.13592 [cs.LG].

Yuyang Rong received his Bachlor degree at ShanghaiTech University in 2019. He joined University of California, Davis to further pursuit a Doctoral degree. His research interest includes fuzzing, software analysis, compiler analysis, and software security.

Chibin Zhang received his Bachelor degree at ShanghaiTech University. He is now a Ph.D. student at Swiss Federal Institute of Technology Lausanne (EPFL). His research focuses on fuzzing, software security.

Jianzhong Liu received his Bachelor degree at ShanghaiTech University. He is pursuing a Ph.D. degree at Tsinghua University. His research interests are in dynamic testing and systems security.

Hao Chen is a professor at the University of California, Davis. He received his Ph.D. at the University of California, Berkeley. His research focuses on security, software engineering, and machine learning. He is a fellow of IEEE.