Valkyrie: Improving Fuzzing Performance Through Deterministic Techniques

Yuyang Rong[1,*], Chibin Zhang[2], Jianzhong Liu[2], and Hao Chen[1]
[1]University of California, Davis, CA, USA
[2]ShanghaiTech University, Shanghai, China
PtrRong@ucdavis.edu, zhangchb1@shanghaitech.edu.cn, liujzh@shanghaitech.edu.cn, chen@ucdavis.edu
*corresponding author

***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. Even fuzzers with a solver rely on incorrect assumptions, limiting their ability to solve branches and forcing them to turn to randomness as a last resort.**

We believe deterministic techniques deliver consistent, predictable, 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 using randomness. 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 shows little to no variance across ten trials and is the fastest to trigger half of the bugs. 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, predictability, reproducibility, and performance.

Keywords—fuzzing; dynamic analysis; vulnerability detection

I. 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 [6, 27], Android apps [19], kernels [16, 29, 30], and application software [2, 4, 7, 13, 21].

Many state-of-the-art greybox fuzzers are based on American Fuzzy Lop (AFL) [2]. 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 [3, 4, 7, 13, 14, 21].

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 [7, 13]. The increased unique branches brought by this new context information exacerbate branch collision problem.

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. [14], 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 [26]. 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" [32] are added to the code. These heuristics caused uneven performances across trials. Therefore, [17] 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 [3]. QSYM uses a modified concolic solver to solve the target predicate, but these solvers cannot solve constraints with complicated forms such as nonconvexity [31]. Angora converts the predicate to an objective function $f(\mathbf{x})$ to optimize using gradient descent, where $\mathbf{x}$ represents sections of input bytes [7]. 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.

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. 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 instrumentations 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.

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.

II. 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 the 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 [7, 13] 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 [14]. 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 [13]. 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 [3], concolic solvers [31], and gradient methods [7]. 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,

```
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 }
```

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

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 can't 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.

III. 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.

A. 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 don't 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.

1) Calculate the number of context-sensitive branches: Let F be the set of all functions in the program, $f \in F$ be a function, $\textit{branch_count}(f)$ be the number of branches in f, and $\textit{context_count}(f)$ be the number of different calling contexts of f. Then the amount of branches is $n = \sum_{f \in F} \textit{context_count}(f) \cdot \textit{branch_count}(f)$.

We calculate $\textit{branch_count}(f)$ through the control flow graph of f. Calculating $\textit{context_count}(f)$ 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.

2) 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 [12, 25].

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

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

$$\textit{offset}(f) + \textit{ID}_f(c) \cdot \textit{branch_count}(f) + \textit{ID}(b)$$

4) 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. Therefore, the removed instrumentation saves runtime by not reporting some edges' status.

However, we would like to find a minimal set of edges to instrument without affecting the distinguishability of different paths. Here, we formally define path and distinguishability:

Definition III.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 III.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 $P' = Q'$.

We do not need to instrument an edge if whether it is taken does not distinguish two different paths. This means there are two requirements for our instrumentation removal. First, for each loop, at least one edge needs to be instrumented. Otherwise, we wouldn't 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 Figure 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*.

[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 markded as delete then
10:             ∀e ∈ E, mark e as keep
11:             mark e1 as delete
12:     Instrument all edges marked with 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` doesn't 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`.

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.

Sufficiency (the left direction): Prove by contradiction. Assume $P \neq Q$ but $P' = Q'$. Let $P = (A, p_1, \cdots)$, $Q = (A, q_1, \cdots)$, 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 $P' = (A', p_1, \cdots)$ and $Q' = (A', q_1, \cdots)$ 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 $P' = (A', p_1, \cdots)$. The assumption $P' = Q'$ implies that $Q = (A, q_1, B, p_1, \cdots)$, 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. □

B. 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 feasible input space are difficult for them to solve, especially when the predicate is an equality comparison. As shown in Listing 1, there are only 256 feasible inputs out of all 256^3 possible inputs to satisfy the comparison. On the other hand, even state-of-the-art fuzzers with a solver may fail because their assumptions are not true.

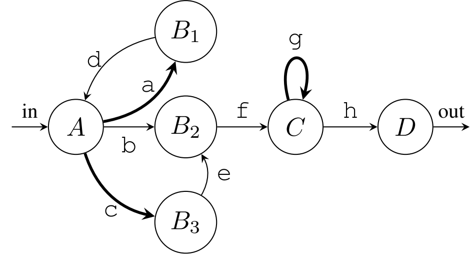

Figure 1: Examples of branches that do not require instrumentation. Only thickened edges need instrumenting.

TABLE I: 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 $f > 0$

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 $\mathbf{x}$ by dynamic taint analysis tools like DataFlowSanitizer [10]. 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 I. 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.

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:

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

If we apply the carry amount $\mathbf{c}_i$ to the ith component, we should have changed the objective function value by $\mathbf{c}_i\mathbf{g}_i$, where $\mathbf{g}_i$ is the partial derivative of dimension i. Since $\mathbf{c}_i$ is a fractional value that cannot be applied, when we carry this amount over to dimension j, To move the function value by the same amount, we should add $\mathbf{x}_j$ by another $\frac{\mathbf{c}_i\mathbf{g}_i}{\mathbf{g}_j}$. We can write the compensation process in Equation 1:

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

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 Equation 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 don't 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 Equation 1:

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

Algorithm 2 Compensated step

```
1:  function COMPENSATEDSTEP(x ∈ Z^n, Δx, g ∈ R^n)
2:      P ← Permutation matrix s.t. ∀i < j, |Pg_i| ≥ |Pg_j|
3:      x ← Px, Δx ← PΔx, g ← Pg        ▷ Sort dimensions in the
        descending order of the absolute value of the gradient
4:      c_0 ← 0, g_0 ← 1
5:      for i in 1..n do
6:          r_i ← Δx_i + (c_{i-1} g_{i-1}) / g_i
7:          Δx'_i = ⌊r_i⌋
8:          if x_i + Δx'_i > max_i then
9:              Δx'_i = max_i − x_i
10:         else if x_i + Δx'_i < max_i then
11:             Δx'_i = min_i − x_i
12:         else if r_i − Δx'_i is too large for the rest dimensions then
13:             Δx'_i = ⌈r_i⌉
14:         c_i = r_i − Δx'_i
15:     return P^{-1} Δx'
```

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 won't 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 Equation 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 Equation 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.

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 i-th 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 i-th dimension, $\mathbf{g}_i$, using Equation 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 $||\Delta\mathbf{x}||_\infty < \epsilon$, we have $f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{g}^T\Delta\mathbf{x}$ where $\mathbf{g}$ is the gradient.

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 Equation 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 may encounter the following problems when descending:

1) The execution path changes and the predicate is unreachable. In this case, we stop descending and use the value from the previous step as a result.
2) The function value drops less than expected or even increases. In this case, we test if the new function value is still descending; if not, we return the previous step.
3) The function value drops 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 case 1 or 2.

Algorithm 3 Descent routine

Require: f
1: **function** DESCENT($\mathbf{x}, \mathbf{g} \in \mathbb{R}^n$)
2: $\quad v \leftarrow max(1, \min_{i \text{ s.t. } \mathbf{g}_i \neq 0} |\mathbf{g}_i|)$
3: $\quad \alpha \leftarrow v/\mathbf{g}^T\mathbf{g}$
4: $\quad \mathbf{x}_{prev} \leftarrow \mathbf{x}$, $f_{prev} = f(\mathbf{x}_{prev})$,
5: **loop**
6: $\quad \Delta\mathbf{x}' \leftarrow$ COMPENSATEDSTEP($\mathbf{x}_{curr}, -\alpha\mathbf{g}, \mathbf{g}$)
7: $\quad \mathbf{x}_{curr} \leftarrow \mathbf{x}_{curr} + \Delta\mathbf{x}'$, $f_{curr} = f(\mathbf{x}_{curr})$
8: $\quad$ **if** $f_{curr} = \infty$ **or** $|f_{prev}| \leq |f_{curr}|$ **then** ▷ Next step doesn't exist or the function is not descending.
9: $\quad\quad$ **return** $\mathbf{x}_{prev}$
10: $\quad$ **else if** ISSOLVED(f_{curr}) **then**
11: $\quad\quad$ **return** $\mathbf{x}_{curr}$
12: $\quad \alpha \leftarrow 2\alpha$, $\mathbf{x}_{prev} \leftarrow \mathbf{x}_{curr}$, $f_{prev} \leftarrow f_{curr}$

The overall modified gradient algorithm is listed in Algorithm 3. We start by calculating the step size using Equation 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) 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 can't decrease more. Thus we carry all the $\mathbf{r}_1$ to the next dimension, i.e. $\mathbf{c}_1 = \mathbf{r}_1, \Delta\mathbf{x}'_1 = 0$.

$\mathbf{c}_1$ is then applied to the next dimension, thus $\mathbf{r}_2 = \Delta\mathbf{x}_2 + \frac{\mathbf{c}_1\mathbf{g}_1}{\mathbf{g}_2} = \frac{1}{3}(1 - \frac{1}{\mathbf{g}^T\mathbf{g}})$. $\mathbf{r}_2$ is again floored to 0, leaving $\mathbf{c}_2 = \mathbf{r}_2, \Delta\mathbf{x}'_2 = 0$.

Interestingly, we have $\mathbf{r}_3 = \frac{\mathbf{c}_2\mathbf{g}_2}{\mathbf{g}_3} - \Delta\mathbf{x}_3 = -1 + \frac{1}{\mathbf{g}^T\mathbf{g}} - \frac{1}{\mathbf{g}^T\mathbf{g}} = -1$, which counters all the fractional value we had. Therefore $\Delta\mathbf{x}'_3 = \lfloor \mathbf{r}_3 \rfloor = -1$ 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 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 the descend using $v = 1$. The final step would give us $\mathbf{x}_{eq} = [0, 1, 11]$, $f(\mathbf{x}_{eq}) = 0$.

IV. EVALUATION

We implemented Valkyrie to conduct a series of experiments to analyze the effectiveness of the entire fuzzer and individual components. We have open-sourced Valkyrie in Github[2]. Due to space limits, we also open-sourced the fuzzers' and softwares' versions we used in this section. We also open-sourced all build scripts, test settings, initial seeds, and docker images we used.

[2]https://github.com/ValkyrieFuzzer

We are interested to know how well Valkyrie works in practice. Thus 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?

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 [15], then on real-world programs. We intend to test Valkyrie on a more robust benchmark FuzzBench [22], 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 [18] to test real-world programs. Each fuzzer runs in a containerized environment with *one core*. Each experiment lasted 24 hours and was repeated ten times, as suggested by [15]. 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 AFL++, which has merged many improvements and function enhancements developed for AFL. We enabled llvm_mode, with AFLfast's power scheduling [4], MOpt's mutator [21], 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 [8]. However, the tool is not available to us.

A. 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 [15]. 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 [11] 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 isn't designed to trigger bugs. We borrow ideas from [24, 20], 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.

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

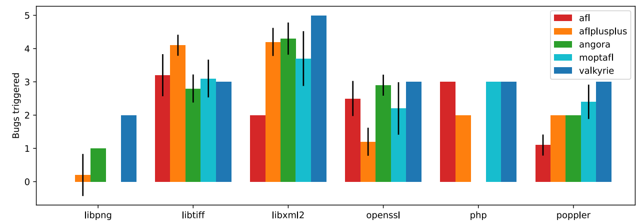

Figure 2: Arithmetic mean of number of integer and memory bugs triggered per trial per day. The black line shows 95% confidence interval. Valkyrie's performance is the same across ten trials.

MoptAFL is also reported to be the best in the benchmark [15], 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 Figure 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 II. 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 hours 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 23h across 10 trials.

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. Of 7 libraries,

```
// AAH001
size_t row_factor_l = 1 + (png_ptr->interlaced? 6: 0)
 + (size_t)png_ptr->width
 * (size_t)png_ptr->channels
 * (png_ptr->bit_depth > 8? 2: 1);
size_t row_factor = (png_uint_32)row_factor_l;
if (png_ptr->height > PNG_UINT_32_MAX/row_factor) {...}
```

Listing 2: Two seemingly easy bugs in Magma. With Valkyrie's solver these bugs can be triggered yet it would took randomized fuzzers hours to trigger.

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. [15] 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. When 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 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 hours 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 [15]'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 [15]. 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.

B. 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 [1].

The results of these experiments are shown in Figure 3. 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).

> 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.

C. 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 Figure 3, 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 III. 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 III 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

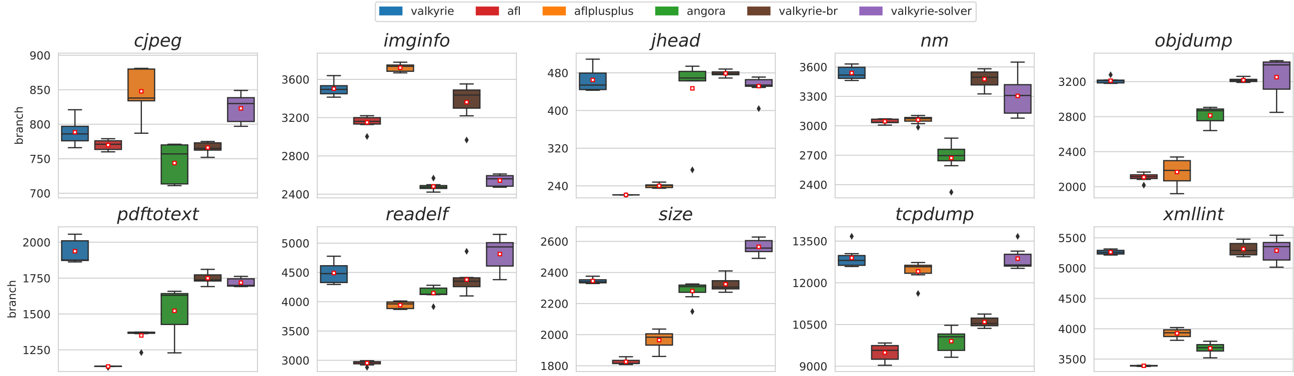

Figure 3: Branch coverage of six fuzzers in 24 hours time. Valkyrie-br is Valkyrie with only branch coverage improvement, Valkyrie-solver is Valkyrie with only solver improvement. Both design increased branch coverage compared with Angora in all programs. Overall, Valkyrie ranked #1 on geometric mean number of branches reached.

TABLE III: 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 IV: 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 8K bitmap.

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

64K buffer. Angora uses 1M to allow context-sensitivity. The utilization rate is shown in Columns 2 and 4 in Table IV. 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 IV. We list the utilization rate under such sizes Column 3 and 5 in Table IV. 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 Figure 3.

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

D. *Effectiveness of deterministic solver*

In Figure 3, 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-sover 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.

E. Discussion

We further studied the reason why Valkyrie-solve can surpass Valkyrie in some cases. Although the only difference between Valkyrie-solver and Valkyrie is that our branch counting method, it 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, and vice versa. These two-sided effects are neither predictable nor desirable, which further justifies our motivation to eliminate branch collisions.

F. 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.

V. Related work

Since AFL, much work has been devoted to strike a balance between branch counting sensitiveness and the probability of colliding. Angora updated AFL's method by adding a function context to the branch counting table [7]. CollAFL proposed replacing AFL's random ID generation with a non-random algorithm that attempts to mitigate branch collisions [14]. However, unlike Valkyrie, neither Angora nor CollAFL is collision-free. Wang et al. formalized branch counting methods and discussed the trade-offs between granularity and performance [26].

Many work focuses on predicate solving. REDQUEEN solves hashes and checksums through input-to-state-correspondance [3]. KLEE uses symbolic execution to solve predicates in the program to generate seeds [5], but symbolic execution can be ineffective when program path is deep and nested. Angora solves branch predicates using principled methods such as gradient descent [7], yet in this paper we show that without continuous assumption Angora's solver may fail some simple cases. Matryoshka proposes procedural methods for solving nested constraints in real-world situations [8].

Ever since AFL family [2, 4], fuzzers have evolved into two main streams: solver-based fuzzer and targetted fuzzer. solver-based fuzzers aim at better solvers on branch predicates to reach high code coverage. However, Angora cannot effectively solve branches that are nested together. Matryoshka [8] solved this problem by a slightly modified gradient solver, which is ad hoc and unjustifiable.

Targeted fuzzers attempt to target the potentially buggy code. ParmeSan use a spectrum of sanitizers to help it find possible buggy codes and put most energy in exploring and exploiting those codes [23]. However, only identifies potential bugs and use branch counting as an incentive to reach bugs instead of using a solver to trigger it like Valkyrie does. Savior also targets potential bugs, but it focuses on using seed scheduling to find those that lead to potential buggy code [9]. TOFO 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 [28]. None of the tools try to target the buggy code by using a new solving method.

VI. 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.

Acknowledgement

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

References

[1] *afl-cov*. URL: https://github.com/mrash/afl-cov.

[2] *American fuzzy lop*. URL: http://lcamtuf.coredump.cx/afl/.

[3] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[4] M. Böhme, V. Pham, and A. Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[6] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing." In: *NDSS*. 2018.

[7] Peng Chen and Hao Chen. "Angora: efficient fuzzing by principled search". In: *IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, 2018.

[8] Peng Chen, Jianzhong Liu, and Hao Chen. "Matryoshka: Fuzzing Deeply Nested Branches". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 499–513. ISBN: 9781450367479.

[9] Yaohui Chen et al. "Savior: Towards bug-driven hybrid testing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1580–1596.

[10] *DataFlowSanitizer*. URL: https://clang.llvm.org/docs/dataflowsanitizer.

[11] Brendan Dolan-Gavitt et al. "Lava: Large-scale automated vulnerability addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 110–121.

[12] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "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. Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 242–256. ISBN: 089791662X.

[13] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[14] S. Gan et al. "CollAFL: Path Sensitive Fuzzing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 679–696.

[15] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A ground-truth fuzzing benchmark". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.

[16] Dae R Jeong et al. "Razzer: Finding kernel race bugs through fuzzing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 754–768.

[17] George Klees et al. "Evaluating fuzz testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 2123–2138.

[18] Yuwei Li et al. "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers". In: *30th USENIX Security Symposium (USENIX Security 21). USENIX Association*. 2021.

[19] Baozheng Liu et al. "{FANS}: Fuzzing Android Native System Services via Automated Interface Analysis". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.

[20] *LLVM Undefined Behavior Sanitizer*. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[21] Chenyang Lyu et al. "MOPT: Optimized Mutation Scheduling for Fuzzers". In: *28th USENIX Security Symposium (USENIX Security 19)*.

[22] Jonathan Metzman et al. "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*. 2021.

[23] Sebastian Österlund et al. "ParmeSan: Sanitizer-guided Greybox Fuzzing". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.

[24] Yuyang Rong, Peng Chen, and Hao Chen. "Integrity: Finding Integer Errors by Targeted Fuzzing". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2020, pp. 360–380.

[25] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 32–41. ISBN: 0897917693.

[26] Jinghan Wang et al. "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)*. 2019, pp. 1–15.

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

[28] Zi Wang, Ben Liblit, and Thomas Reps. "TOFU: Target-Orienter FUzzer". In: *arXiv preprint arXiv:2004.14375* (2020).

[29] M. Xu et al. "Krace: Data Race Fuzzing for Kernel File Systems". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1643–1660.

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

[31] Insu Yun et al. "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 745–761. ISBN: 978-1-931971-46-1.

[32] Andreas Zeller. *When Results Are All That Matters: The Case of the Angora Fuzzer*. Oct. 2019. URL: https://andreas-zeller.info/2019/10/10/when-results-are-all-that-matters-case.html.