PANGOLIN: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction

Heqing Huang†, Peisen Yao†, Rongxin Wu‡, Qingkai Shi†, Charles Zhang†
†The Hong Kong University of Science and Technology, China
‡Xiamen University, China
{hhuangaz, pyao, qshiaa, charlesz}@cse.ust.hk, wurongxin@xmu.edu.cn

***Abstract*—Hybrid fuzzing, which combines the merits of both fuzzing and concolic execution, has become one of the most important trends in coverage-guided fuzzing techniques. Despite the tremendous research on hybrid fuzzers, we observe that existing techniques are still inefficient. One important reason is that these techniques, which we refer to as non-incremental fuzzers, cache and reuse few computation results and, thus, lose many optimization opportunities. To be incremental, we propose "polyhedral path abstraction", which preserves the exploration state in the concolic execution stage and allows more effective mutation and constraint solving over existing techniques. We have implemented our idea as a tool, namely PANGOLIN, and evaluated it using LAVA-M as well as nine real-world programs. The evaluation results showed that PANGOLIN outperforms the state-of-the-art fuzzing techniques with the improvement of coverage rate ranging from 10% to 30%. Moreover, PANGOLIN found 400 more bugs in LAVA-M and discovered 41 unseen bugs with 8 of them assigned with the CVE IDs.**

***Index Terms*—Fuzzing, constraint solving, program analysis, sampling**

I. INTRODUCTION

Hybrid fuzzing, combining the merits of both fuzzing and concolic execution to better explore program states [1], [2], [3], [4], [5], [6], has recently become one of the most important trends in the coverage-guided fuzzing techniques. Hybrid fuzzing leverages concolic execution [7] to generate seed inputs that drive the program execution to hard-to-cover branches, and resorts to mutation to speed up the exploration of trivial input spaces. The efficacy of such combination has been well demonstrated in various well-known benchmarks, such as DARPA Cyber Grand Challenges binaries and LAVA-M dataset, as well as the real-world applications [1], [2].

Despite this tremendous progress in achieving high coverage rates, hybrid fuzzing is well known to still suffer from efficiency issues. For instance, Driller [2] could cause more than 100 times runtime overhead analyzing coreutils while the coverage is lower than 20% [1]. We have observed that one important reason for this deficiency is that hybrid fuzzing, as far as the state of the art is concerned, is not incremental when going deeper in nested path constraints, in the sense that little exploration state is preserved and reused in the successive epochs of concolic execution and mutation. However, intuitively, the search space of the previous epoch provides good guidance for exploring that of the next.

More specifically, in the current design of hybrid fuzzing [1], [2], [6], two successive exploration epochs are connected by two elements, the newly generated seed and the symbolic snapshot, which are, unfortunately, not state preserving and cause serious computational redundancies. Taking the program in Figure 1 as an example, an earlier concolic execution can generate a seed input ($v = 20, w = 5, x = 3, y = 4, z = 30$), which satisfies the predicate at Line 3. In the next epoch, the current approaches mutate this seed *randomly* to explore the nested branches at Line 4 and Line 8. This is not effective because, even with the help of taint analysis [8], [9], which, in this example, determines only z needs to be mutated, the probability of satisfying the constraint $0 < z < 200$ at Line 3 is still very low since z ranges over the entire integer domain. On the contrary, if the mutator retains not only the seed but also the solution intervals of the predicate at Line 3, i.e., $x \in [3, 3], y \in [4, 4], z \in [0, 200)$, it should be much faster to get pass the first branch at Line 3 and focus on exploring branches of Line 4 and Line 8.

The same drawback also happens at the concolic execution stage. The symbolic snapshot simply memorizes the enclosing path constraints and the symbolic values of the environment, which is to be set in conjunction with the nested branch conditions and sent to constraint solvers. It is obvious that the enclosing path constraints, e.g., $x = 3 \wedge y == 4 \wedge z * z < 40000$, in our example, are solved repetitively, for instance, at both Line 4 and Line 8. As a result, the concolic execution becomes even more sluggish as the fuzzing process goes deeper and deeper in the nested branches. However, if we replace the enclosing path constraint with the solution intervals, $x \in [3, 3], y \in [4, 4], z \in [0, 200)$, it becomes trivial for constraint solvers to find a new seed for Line 4 with the new constraint $z > 195$.

Our example illustrates that being incremental, i.e., retaining and reusing a summary about the solution space of a previous epoch, in hybrid fuzzing has tremendous benefits in both guiding the mutation of the seeds and efficiently solving nested path constraints. We construct this summary as the bounded ranges for the input variables and their linear expressions at the concolic execution stage. Specifically, in addition to finding a single feasible solution of the outer branch through constraint solvers, we also target at obtaining the bounded ranges of the related input variables for the target path constraint. We describe such bounded ranges by linearizing the path constraint as a polyhedron, denoted as the "*polyhedral path abstraction*", for guiding both the mutation and the constraint

DOI 10.1109/SP40000.2020.00063

```
1  int main() {
2    unsigned v,w,x,y,z = input();
3    if(x==3 && y==4 && z*z<40000){
4      if(z > 195){
5        ......
6        //crash 1
7      }
8      if(15<=z+v && z+v<=25){
9        ......
10       //crash 2
11     }
12   }
13   // other instructions related to w
14 }
15
```

Fig. 1: Motivating Example

solving procedure in hybrid fuzzing:

- The polyhedral path abstraction of a path constraint enables us to convert the problem of mutating the seeds into the problem of sampling over a polyhedron, which has been well studied and has many efficient solutions [10], [11], [12], [13], [14], [15]. In this work, we adopt a state-of-the-art technique, the Dikin walk algorithm [15], which can achieve the polynomial time complexity, to efficiently generate a large number of inputs while still respecting the target path constraints.
- Moreover, we adopt the polyhedral path abstraction to speed up the constraint solving in the concolic execution from two aspects. First, since the polyhedral abstraction is a sound and simplified form of the solved constraint, to prove the infeasibility of a new path concatenated using the constraint from the previous epochs, i.e., the prefix, we can safely use the polyhedral abstraction of this prefix instead to dramatically reduce the constraint solving complexity. Second, we can narrow down the solution space of the path constraint for a feasible path, by using the polyhedral abstraction of the prefix.

To evaluate the efficiency and effectiveness of our proposed technique, we implemented a tool, PANGOLIN, and compared it to the state-of-the-art fuzzing frameworks using both the well-known benchmarks and the real-world systems. Our evaluation results show that, PANGOLIN not only improves the coverage by 10% to 38% than the state of the art, but also finds more bugs in the benchmark dataset and discovers previously unknown vulnerabilities. For instance, PANGOLIN has found 500+ more bugs than the state-of-the-art techniques in the LAVA-M dataset. For the real-world systems, although they have been extensively examined by the state-of-the-art fuzzers, we still can discover 41 unseen bugs with 8 of them assigned with the CVE IDs. In summary, this work makes the following contributions:

- We are the first to propose the concept of incremental fuzzing, in which we compute the polyhedral path abstraction of an earlier path constraint to guide the later fuzzing processes.
- We design an efficient input generation method based on the polyhedral path abstraction, which accelerates both the concolic execution stage and the mutation stage in hybrid fuzzing.
- We evaluate our approach and the experimental results demonstrate that it outperforms the state-of-the-art fuzzers in terms of both achieving a high coverage rate and finding previously unseen bugs.

II. BACKGROUND

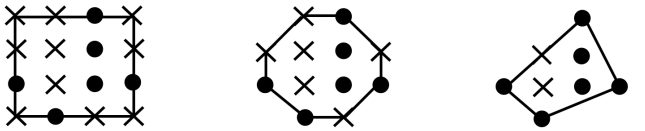
A. Coverage-guided Fuzzing

Coverage-guided fuzzing is one of the most powerful vulnerability detection techniques [1], [16], and has been widely adopted to detect various kinds of software security issues in the industry. The general workflow of coverage-guided fuzzing consists of three steps: 1) determining the uncovered path; 2) generating a seed input satisfying the path predicates of the uncovered path; 3) mutating the seed to generate a large number of inputs to further increase the coverage rate. Despite tremendous research progresses [1], [3], [9], [17], [18], [19], [20], [21], [22], [23], [24], [25], [25], [26], [27], [28], [29], we observe that all the existing fuzzing approaches spend the majority of the efforts on finding a limited number of seed inputs satisfying those hard-to-cover conditions, but neglecting these efforts when it comes to mutating the seeds. Thus, the search space for mutating the seed is always too large to find an input for new coverage, especially for those complex and tight constraints.

The majority of fuzzing frameworks [1], [18], [19], [20], [21], [24], [27], [30], [31] mutate every seed randomly using some simple heuristics. Although randomly mutating the seed is fast to generate a large number of inputs, it is not efficient to produce inputs satisfying tight and complex path constraints. Take the program in Figure 1 as an example. To take the true branch at Line 3, the inputs need to satisfy the constraint ($x = 3 \wedge y = 4 \wedge z*z < 10000$). However, it is very hard for random mutation to generate such inputs, because the search space of the inputs is around 2^{32*5} and the probability of generating the feasible inputs is around 90 / $(2^{32+32+32}) \approx 10^{-27}$.

To reduce the large search space of inputs, taint analysis [9], [25], [26] is used to determine the input offsets that are correlated with the path constraints in the target conditional statements. As illustrated in Figure 1, the taint analysis can determine that the variables z, i, k are irrelevant to the conditional statements at Line 3. Thus, a fuzzer can focus on mutating the variables x and y. This greatly improves the efficiency of the mutation by reducing the search space to 2^{32*3}. However, as taint analysis does not give any hints on how to mutate the byte offsets of the inputs, these approaches still rely on the random mutation to generate seeds, which is inefficient. For example, the probability of generating the feasible inputs for the example in Figure 1 is still 10^{-27}.

Another category, hybrid fuzzing, leverages constraint solving [1], [2], [3], [4] to generate seed inputs to drive the program execution to pass tight and complex path constraints,



$x \in [a, b]$
$y \in [c, d]$
Interval

$k_1x + k_2y \leq b$
$k_i \in \{0, \pm 1\}$
Octagon

$k_1x + k_2y \leq b$
$k_i \in \mathbb{Z}$
Polyhedra

Fig. 2: Different types of path abstraction. In the figure, a, b, c, and d are constants.

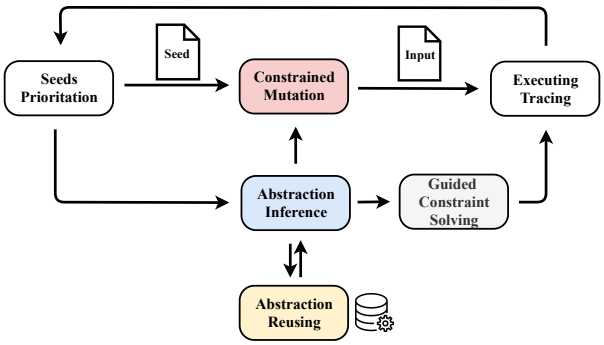


Fig. 3: Architecture of PANGOLIN

but, unfortunately, still resorts to inefficient mutation to speed up the exploration of the input space. The lack of effective guidance to mutation makes their approaches excessively rely on the heavy-weight constraint solving to achieve a high coverage rate. First, given a seed input discovered by solving constraints, randomly mutating it would easily invalidate its associated path constraints that have already been conquered by constraint solvers, wasting the computation in exploring the nearby nested branches. For example, suppose we have leveraged a solver to obtain a seed input ($v = 20, w = 5, x = 3, y = 4, z = 30$), which satisfies the predicate at Line 3. Any new inputs generated by mutating the variables x and y make it difficult to explore the successive branches at Lines 4 and 8. Second, with the growth of the nested level of conditional statements, the path constraints become increasingly restricted, which makes the mutation less and less efficient. For example, the predicate ($z > 195$) at the nested conditional statement of Line 4 is not difficult to be satisfied by mutation. However, the path constraint that conjuncts with the predicates at Line 3 and Line 4 (i.e., $x = 3 \wedge y = 4 \wedge 195 < z < 200$) becomes challenging for mutation. Even though we already have a seed input ($v = 20, w = 5, x = 3, y = 4, z = 30$) and only consider mutating the variable z, the probability of generating a feasible input to reach the condition at Line 4 is around $100/2^{32}$, and the probability to cover the true-branch of the predicate at Line 4 decreases to $4/2^{32}$.

B. Polyhedral Path Abstraction

In this work, we use the notion "path abstraction" to denote an approximation of a path constraint. In the existing literature, many different abstractions have been studied, such as interval [32], octagon [33], and polyhedral [34]. As illustrated in Figure 2, the interval abstraction only contains the value ranges of each variable. The octagon abstraction is in the form $k_1x + k_2y \leq b$, where x and y are the variables in the path constraints and $k_i \in \{0, \pm 1\}$. The polyhedral abstraction is in a more general form where $k_i \in \mathbb{Z}$. In Figure 2, the black dots represent all feasible values satisfying a given path constraint, whereas the crosses represent the infeasible ones. The path abstraction is the region bounded by the lines representing multiple linear inequalities.

All the path abstractions approximate non-linear formula in the given path constraints. All these abstractions are sound, because the regions cover all the black dots. However, they have different levels of the precision. According to the recent studies [35], [36], the polyhedral abstraction has the best precision. The example in the Figure 2 also demonstrates the best performance of the polyhedral abstraction, as only two infeasible values are confused by the polyhedron.

We further present the properties of the path abstraction required by a fuzzer and detail how to convert a path constraint into a polyhedral path abstraction in Section IV-A.

III. OVERVIEW

In this section, we describe the design philosophy of realizing PANGOLIN and illustrate the workflow of PANGOLIN shown in Figure 3. PANGOLIN is essentially a hybrid fuzzing technique. It shares the typical components as the conventional hybrid fuzzing techniques that have been briefly introduced in Section II, such as seeds prioritization and tracing. We focus on our discussion on the following ideas.

A. Path Abstraction Inference

We first identify the uncovered branches in fuzzing and deliver them to the concolic execution engine to proceed. Different from the concolic execution in the conventional hybrid fuzzing which invokes the constraint solver to directly obtain a feasible solution, PANGOLIN constructs a summary of these uncovered branches, which we denote as the polyhedral path abstraction. The polyhedral path abstraction describes a sound search space of the feasible inputs with respect to the path constraints, which is used to guide the mutation of the seeds and speed up the solving of the subsequent path constraints. We explain how to construct a polyhedral path abstraction in Section IV-A.

B. Constrained Mutation

As the polyhedral path abstraction renders a bounded range of the input variables with respect to the path constraint of a path prefix, by sampling from such bounded search space, we are able to quickly generate a large number of new inputs that still satisfy this path constraint and, meanwhile, to explore the subsequent paths sharing the same path prefix. Specifically, in this work, we adapt an existing sampling technique, the Dikin walk algorithm [15], to generate new inputs (See Section

IV-B). This sampling method guarantees that the new inputs uniformly cover the given search space, ensuring the diversity in exploring the program states.

C. Guided Constraint Solving

After sampling the inputs, PANGOLIN still preserves the polyhedral path abstraction offline for the successive fuzzing epochs. Unlike the heavy-weight memory-snapshot or fork-server method [2] that only stores the constraints themselves, the polyhedral path abstraction is a succinct and sound memento of the solution space of the path constraints we have solved so far, which dramatically simplifies the solving of the path constraint from the next epochs, since it uses the previous path constraints as its prefix. Compared to the incremental solving [37], PANGOLIN does not need to maintain a heavy-weight memory snapshot, but provides a light-weight caching mechanism that also guides input generation for hybrid fuzzing. We further explain this technical detail in Section IV-C.

IV. METHODOLOGY

In this section, we explain the technical details of the key components of PANGOLIN. Section IV-A presents the inference procedure of the polyhedral path abstraction. Section IV-B and Section IV-C discuss how the polyhedral path abstraction benefits the mutation and the constraint solving processes in hybrid fuzzing.

A. Path Abstraction Inference

We notice that retaining and reusing the solution space of a previous epoch have tremendous benefits in both guiding the mutation of the seeds and efficiently solving nested path constraints. This solution space is summarized as the *polyhedral path abstraction* (See Section II), which is a simplified and linear approximation of the path constraint. A polyhedral path abstraction should satisfy the following requirements to support incremental hybrid fuzzing:

1) The path abstraction needs to be sound. The path abstraction needs to include all feasible inputs for achieving new coverage. As an example, in Figure 1, if the path abstraction constructed for the variable z is an unsound interval, e.g., $z \in [0, 100]$, we then lose the opportunity to generate a value of the variable z within $[196, 199]$ to cover the true branch of the condition at Line 4 and, thus, cannot trigger the crash at Line 5.
2) The path abstraction can be inferred efficiently while containing as few false-positive values as possible. The precision of the path abstraction determines its effectiveness to guide input generation. Figure 4 demonstrates the two possible abstractions of the condition at Line 8 in Figure 1. Although inferring the interval abstraction is cheaper than the octagon abstraction, it contains over 60% more false-positive values.

Considering that the path abstraction should be built efficiently and easy to utilize in the subsequent fuzzing process, we take advantage of a sweet spot between precision and speed by producing the path abstraction as a summary of the current program states. To this end, the state-of-the-art method BWAI [38] is leveraged in our approach to linearize a path constraint to its polyhedral (or linear) form. Basically, it regards the abstraction inference as a typical SMT-based optimization (SMT-opt) problem [39], which computes the minimum upper-bound value (i.e., b in Figure 2) of an objective linear expression (i.e., $k_1x + k_2y$) subjected to the given path constraint. Moreover, SMT-opt ensures the abstraction is sound because it guarantees that the computed upper-bound value is the minimum subjected to the constraint.

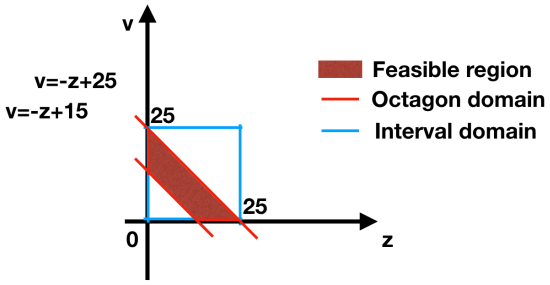


Fig. 4: Example of different path abstraction.

BWAI enumerates all possible relations between any two variables using the predefined templates: x_1, x_2, $x_1 + x_2$, and $x_1 - x_2$, as the objective linear expressions. It calculates the boundary values of these objectives through solving the SMT-opt problem. The outcomes, which are in the form of $k_1x + k_2y \leq b$ ($k \in \{0, \pm 1\}$), constitute the approximation of the given path constraint. As an example, for the path constraint

$$x \leq 2 \wedge y \leq 5 \wedge y \leq x^2 - 5x + 4 \tag{1}$$

the BWAI method produces the path abstraction including the following linear inequalities:

$$\begin{cases} 0 \leq x \leq 2 \\ 0 \leq y \leq 5 \end{cases} \quad \begin{cases} 1 \leq x + y \leq 7 \\ -5 \leq x - y \leq 2 \end{cases}$$

Since BWAI enumerates all possible relations between any two variables as exemplified above, the number of the linear inequalities in the path abstraction grows quadratically, on the order of $\mathcal{O}(n^2)$, along with the increase of the number, n, of variables. This hinders the scalability of our method. In practice, to reduce the overhead caused by computing the path abstraction, we do not generate all possible inequalities, as illustrated above. Instead, we generate a much smaller set of the equality formulas, which, however, can still demonstrate a better precision for hybrid fuzzing. We present our method in Algorithm 1.

Let us use the same path constraint in Equation (1) to illustrate the algorithm. We do not compute the boundary for the linear expressions $x + y$ and $x - y$ because, on one hand, they do not exist in the input path constraint and, on the other hand, the result $x + y \leq 7$ and $-5 \leq x - y \leq 2$ is a super set of $0 \leq x \leq 10 \wedge 0 \leq y \leq 3$, thus making no contribution to the path abstraction. Based on our algorithm, we compute the

Algorithm 1 Polyhedral path abstraction inference.

```
1:  procedure INFERENCE(pc ≜ σ1 ∧ σ2... ∧ σn)
2:      pc, path constraint. p̂c, polyhedral path abstraction.
3:
4:      p̂c ← true
5:      for all input variable vi in pc do
6:          min ← SMToptMin(vi, pc)
7:          max ← SMToptMax(vi, pc)
8:          p̂c ← p̂c ∧ min ≤ vi ≤ max
9:      end for
10:     for all atomic predicate σi in pc do
11:         if σi contains linear expression ιi then
12:             min ← SMToptMin(ιi, pc)
13:             max ← SMToptMax(ιi, pc)
14:             p̂c ← p̂c ∧ min ≤ ιi ≤ max
15:         end if
16:     end for
17:
18:     return φ̂
19: end procedure
```

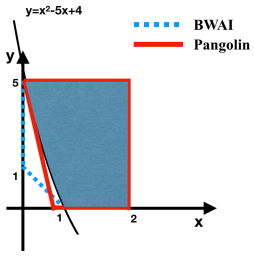


Fig. 5: Pangolin vs. BWAI.

boundary for every single variable (Lines 5 - 9), and compute the boundary of every linear expression existing in the input path constraint (Lines 10 - 16). As a result, it produces the following path abstraction:

$$\begin{cases} 0 \leq x \leq 2 \\ 0 \leq y \leq 5 \\ 4 \leq 5x + y \leq 15 \end{cases}$$

As shown in Figure 5, our path abstraction provides a more precise and, meanwhile, more concise approximation for the input path constraint. In practice, it is not necessary to compute the boundary values of each variable or linear expression individually, as shown by the for-loops in Algorithm 1. Instead, we can benefit from multi-objective optimization algorithms [40] to compute boundary values for multiple linear expressions at the same time. Such a multi-objective method can avoid repetitively computing many intermediate results shared by individual SMT-opt procedures, thus speeding up our approach.

Our approach to computing polyhedral path abstraction is efficient and effective because of the following two reasons. First, the linear expressions are not predefined templates, such as $x + y$ and $x - y$, but extracted directly from the original path constraints. Thus, they can better reveal the dependence

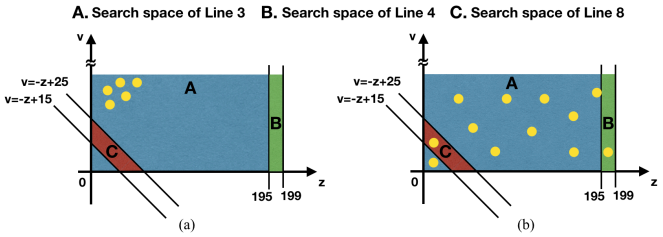


Fig. 6: The importance of uniform sampling. **A**, **B** and **C** are the search spaces of constraints at Lines 3, 4, and 8, respectively.

relations among program variables, thereby being capable of guiding the mutation and constraint solving procedures more effectively. Second, the number of the linear expressions is linear, rather than quadratic, to the size of path constraints. Meanwhile, there are equalities that can be extracted from the constraints directly without further computation such as $x < 2$ in Equation (1). Thus, our approach can generate the path abstraction more efficiently. Next, we detail how the polyhedral path abstraction benefits the mutation and constraint solving procedures in hybrid fuzzing.

B. Constrained Mutation

As mentioned in Section I, mutation is not effective even with the assistance from constraint solving. Specifically, this ineffectiveness obstructs a fuzzer to generate multiple inputs to sufficiently test a hard-to-cover branch. To tackle this problem, the mutation method should have two properties. First, the mutation cannot be completely random. The mutation results should be restricted by the path constraint, so that we can efficiently generate multiple inputs for a given branch. Second, the mutated inputs should be uniformly distributed over the solution space of the path constraint. It is well known that such a uniform distribution is more likely to trigger different program behaviors [41].

To illustrate these two properties, let us consider the example in Figure 1 again. When the concolic engine reaches Line 4, we generate a new seed satisfying the branch conditions. On the one hand, randomly mutating the seed may dissatisfy the branch conditions and, thus, is not effective for testing the behaviors in the branch. Therefore, we need to constrain the mutation process. On the other hand, as illustrated in Figure 6, if the inputs generated by mutation are not uniformly distributed, a fuzzer will have a much lower probability to cover some behaviors such as the regions described by the constraints at Lines 4 and 8, named as Region B and Region C.

1) How to generate: PANGOLIN uses the "Dikin walk" [15] to guarantee the efficient and uniform input generation. The Dikin walk algorithm allows the uniform sampling over a polyhedron described by a series of linear inequalities. The core idea of Dikin walk is to ensure that the distance between every two sampling results must be greater than a dynamic bound, which guarantees uniform sampling in a region.

Moreover, the complexity of the Dikin walk algorithm is $O(mn)$, where m and n are the numbers of inequalities and variables in the path abstraction, respectively. It is noteworthy that such polynomial time complexity is much lower than the NP-completeness of constraint solving. Thus, compared to the conventional approaches, we are more efficient to generate multiple inputs for a hard-to-cover branch.

2) How many to generate: To determine the number of generated inputs, we take two factors into consideration. First, the harder a branch to cover, the more inputs we need to generate using constrained mutation. This is because, if a branch is not hard to cover, it is easy to use the conventional random mutation to generate inputs for the branch. Second, the more paths that depend on a branch, the more inputs we need to generate to cover these paths.

For the first factor, given a branch and a seed that can reach the branch, we calculate p as the proportion of *the successful mutation times* over *the total mutation times of the seed*, to measure the difficulties of covering the branch. Here, a successful mutation means that the execution with the mutation result can reach the branch. For the second factor, to measure the potential number of paths depending on a given branch, we use the solution space of the polyhedral path abstraction at the branch to approximate. To this end, we calculate v as the volume of the polyhedron using a state-of-the-art method [42] defined as: $v = m(n+1) + \sum_{i=1}^{m} b_i + \sum_{i=1}^{m} \sum_{j=1}^{n} k_{ij}$, where m is the number of the inequalities, n is the number of the involved variables, b_i and k_{ij} are the parameters of the inequalities $k_{ij}x_{ij} < b_i$.

Putting the two factors together, we use the formula $\sqrt{vp^{-1}}$ to approximate the number of inputs to generate. Basically, this formula follows our intuition: the smaller the value of p, or the larger the value of v, the more inputs to generate.

3) Cooperating with random mutation: It is noteworthy that we do not give up random mutation because it is very cheap and can quickly cover most easy-to-cover branches. Only for hard-to-cover branches, we then create the path abstraction and utilize the constrained mutation to generate diversified seeds as discussed above.

C. Guided Constraint Solving

Not only is the path abstraction able to guide the mutation, it is also capable of speeding up the constraint solving process, which is notoriously expensive in practice. Specifically, the polyhedral path abstraction can speed up constraint solving from two aspects: quickly pruning infeasible paths and narrowing down the solution space of the path constraint for a feasible path.

To be clear, we use $pc(\pi)$ and $\hat{pc}(\pi)$ to represent the path constraint and its path abstraction of a path π, and assume that a path π is the concatenation of a prefix path π_1 and the remaining part π_2.

In PANGOLIN, before solving the path constraint $pc(\pi)$ of a hard-to-cover branch, we first compute and solve a simplified form of the constraint using the abstraction of the existing prefix path, i.e., $\hat{pc}(\pi_1) \wedge pc(\pi_2)$. Generally, the simplified form is less complex than the original one and, thus, is easy to solve. On the one hand, if $\hat{pc}(\pi_1) \wedge pc(\pi_2)$ is unsatisfiable, we can immediately prune the path π because $pc(\pi_1) \wedge pc(\pi_2)$ must be unsatisfiable. On the other hand, if $\hat{pc}(\pi_1) \wedge pc(\pi_2)$ is satisfiable, we continue to solve $\hat{pc}(\pi_1) \wedge pc(\pi_1) \wedge pc(\pi_2)$, i.e., $\hat{pc}(\pi_1) \wedge pc(\pi)$. Although this constraint looks more complex than $pc(\pi)$, it is also much easier to solve in practice because the path abstraction $\hat{pc}(\pi_1)$ actually narrows down the solution space of the path constraint. Since the path abstraction contains computed linear inequalities appeared in the path constraints, the conjunction of path constraint and abstraction directly narrows down the search space.

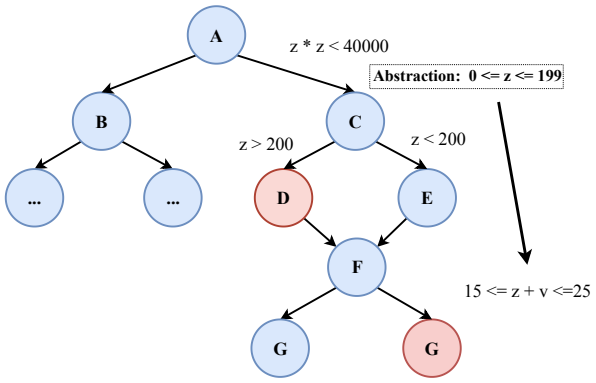


Fig. 7: A control flow graph to illustrate the path-abstraction guided constraint solving. The branch condition is labeled on the edge.

Let us use the example in Figure 7 to demonstrate the method. In this example, we assume that we have generated a path abstraction $\hat{pc}(AC) = 0 \leq z \leq 199$ for the path AC, of which the path constraint is $pc(AC) = z^2 < 40000$. When the concolic execution reaches the node D, PANGOLIN does not solve $pc(ACD)$ immediately, but reuses the abstraction to ease the burden of constraint solving. That is, we first compute $\hat{pc}(AC) \wedge pc(CD) = 0 \leq z \leq 199 \wedge z > 200$, which is apparently unsatisfiable. Thus, we save the cost of an expensive SMT solving process.

In the other case, let us assume that the concolic execution reaches the node G, where the path constraint is $pc(ACEFG) = z^2 < 40000 \wedge z < 200 \wedge 15 \leq z + v \leq 25$. Instead of solving the path constraint directly, we add the path abstraction $\hat{pc}(AC)$ to $pc(ACEFG)$ so that the search space of z can be narrowed down from $(-\infty, 200)$ to $(0, 199]$, thereby reducing the cost of SMT solving.

V. EVALUATION

We have built PANGOLIN on top of AFL (v.2.52b) [30] and QSYM [1], which are the state-of-the-art fuzzing and concolic execution frameworks, respectively. When fuzzing a program with AFL and encountering a hard-to-cover branch, we leverage QSYM for collecting the path constraints and use the SMT-opt algorithms in Z3 (v.4.8) [43] to compute the path abstractions. PANGOLIN is capable of running on both 32bit and 64bit platforms.

TABLE I: Baseline fuzzers.

Fuzzer	Technical Description
AFL [30]	Fuzzer baseline with evolutionary search
AFLFast [20]	AFL + power scheduling
QSYM [1]	AFL + concolic execution
Driller [2]	Demand-driven concolic execution
Angora [9]	Evolutionary search + taint analysis + gradient descent
T-Fuzz [46]	Program transformation + symbolic execution

TABLE II: Real-world benchmark programs.

Program	Version	Input format	Argument
readelf	2.33	ELF	-agteSdcWw –dyn-syms -D @@
nm-new	2.33	ELF	-C -a -l –synthetic @@
objdump	2.33	ELF	-D @@
libtiff	4.0.10	TIFF	@@
tcpdump	commit-b5046f	PCAP	-evvvnr @@
jhead	3.03	JPG	@@
libjpg	commit-ec5adb	JPG	@@
libpng	1.6.37.git	PNG	@@
bento	commit-cbebcc	MP4	@@

In this section, we design a series of experiments to evaluate the effectiveness of PANGOLIN by investigating the following research questions:

1) Can PANGOLIN detect more bugs in comparison with the state-of-the-art fuzzers? (Section V-A)
2) Can PANGOLIN achieve higher coverage rate in comparison with the state-of-the-art fuzzers? (Section V-B)
3) How effective are the path-abstraction-guided mutation and constraint solving? (Section V-C)

Benchmarks. First, we included the LAVA-M dataset [44], a widely-used benchmark containing artificial vulnerabilities with the given oracle, in our evaluation. LAVA-M is an artificial benchmark for bug detection. Each bug is identified with a unique identifier, which is printed when the corresponding bug is triggered. This allows us to know the exact number of bugs triggered by the fuzzers. The LAVA-M benchmark consists of four coreutils programs: *uniq*, *base64*, *md5sum*, and *who*. It has been widely evaluated by the majority of the state-of-the-art fuzzing techniques.

Second, we evaluated PANGOLIN on nine real-world programs shown in Table II. We used the newest version of each program. All of the programs have been widely-evaluated by the state-of-the-art fuzzers in both academia [1], [9], [31] and industry [45] and, thus, are expected to have high quality. These programs also have diverse functionalities and complexity, which demonstrate the usefulness of our approach in practice.

Baseline Approaches. Following the evaluation instructions provided by the previous work [1], [16], we compared PANGOLIN with the state-of-the-art fuzzers in different categories listed in Table I.

- AFL [30] is a well-known fuzzing framework with industrial strength. We used two instances of AFL (afl-master and afl-slave) during the comparison.
- AFLFast [20] optimizes AFL by powerful seed scheduling method. Similarly, we set up two instances of AFLFast (afl-master and afl-slave).
- QSYM [1] is one of the state-of-the-art hybrid fuzzing frameworks that optimize the performance of constraint emulation. For QSYM, we ran it with one concolic engine and one fuzzer engine, i.e., AFL.
- Driller [2], another state-of-the-art hybrid fuzzer that only solves path constraints in a demand-driven way. We ran the approach using the interface `shellphuzz` [47] provided by the authors, which launches one AFL instance and one Driller instance.
- Angora [9], which is one of the most recent greybox fuzzers, leverages an effective gradient descent method to quickly satisfy complex path conditions. For Angora, we used two runtime workers (`-j 2`).
- T-Fuzz [46] bypasses the hard-to-cover conditions by erasing them from the original programs. It wraps AFL and the symbolic execution engine Angr [48]. Since there are no parameter settings in T-Fuzz, we directly run T-Fuzz to collect the experimental results.

Experimental setup. The initial seed corpus plays an important role in the overall performance [22]. We prepared our initial seed corpus for each benchmark in the following ways. For the LAVA-M benchmark, we ran each fuzzer with the seed provided by the authors. For the real-world projects, we followed the standard instructions in the previous paper [16]. To construct this corpus, we first collected the seeds for *libtiff*, *libpng*, *libjpg*, and *jhead* from the official website of AFL [49]. The seeds for *tcpdump* and *bento4* were provided by their Github repositories[1,2]. The seed corpus for *binutils* come from its own testsuite[3]. We ran all the fuzzing techniques using the same initial seed corpus for each target program.

With the initial seed corpus, we then ran AFL to fuzz the target programs for an hour. This process generates inputs to cover the easy branches and, thus, we can focus on the hard-to-cover branches in the subsequent experiments. Note that, during the fuzzing process, we applied `afl-cmin` on the seeds to prune the duplicates.

We followed the setting of the existing studies [1], [9] to set the time limits for evaluating the benchmarks. We used twenty-four hours time budget for both LAVA-M and real-world projects. To avoid the influence brought by randomness, we ran each experiment ten times and used the average as the final results. In addition, we also employed the Mann-Whitney U Test [50] to demonstrate the statistical significance of the contribution made by each part of our framework.

All experiments were conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64GB of memory running Ubuntu 16.04 LTS.

[1]https://github.com/the-tcpdump-group/tcpdump/tree/master/tests
[2]https://github.com/axiomatic-systems/Bento4/tree/master/Test
[3]binutils-gdb.git/binutils/testsuite/

TABLE III: Average results of 24 hours experiment on LAVA-M for 10 times. The data is shown in the form of $+/*$, where $+$ stands for the bugs detected by each fuzzers and $*$ stands for the bugs listed by the authors.

	uniq	base64	md5sum	who
PANGOLIN	30/28	48/44	60/57	2021/2136
QSYM	28/28	44/44	57/57	1353/2136
AFL	11/28	2/44	0/57	2/2136
AFLFast	25/28	28/44	1/57	4/2136
Angora	29/28	48/44	59/57	1619/2136
T-Fuzz	26/28	40/44	49/57	61/2136
Driller	12/28	4/44	8/57	24/2136

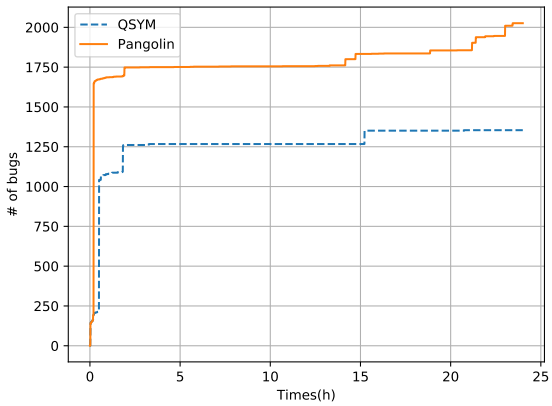


Fig. 8: Evaluation on *who* in 24 hours comparing with QSYM of 10 times repeated experiment with p-value $< 10^{-6}$.

A. Discovering Bugs

The main purpose of fuzzing is to detect bugs. Therefore, we first evaluated the bug detection capability of PANGOLIN.

1) The LAVA-M dataset: Within the time budget of twenty-four hours, we observed that PANGOLIN significantly outperforms the baseline fuzzers in terms of the number of bugs detected, as shown by Table III. Especially, for the subjects, *uniq*, *base64*, and *md5sum*, PANGOLIN detected all the injected bugs within five minutes and even found new bugs unlisted in the oracle. For the largest program *who*, AFL, AFLFast, Driller, and T-Fuzz detected only a few bugs while PANGOLIN found nearly all the listed bugs in the LAVA-M dataset. For instance, as shown in Figure 8, PANGOLIN found 600 more bugs than the state-of-the-art hybrid fuzzer QSYM. The improvement shows the effectiveness of PANGOLIN for satisfying hard-to-cover conditions, which benefits from our constrained mutation and guided constraint solving method.

We further analyzed the reason why PANGOLIN could not detect all the bugs in LAVA-M. Since PANGOLIN was built on top of the concolic engine of QSYM, PANGOLIN naturally inherits some of its limitations, such as the lack of modeling the low-level system calls and supporting the floating-point constraints. These limitations lead to the missing of detecting some bugs in the program *who*. For example, we noticed that *who* applies the low-level system function wrapper *x2nrealloc* to reallocate memory for parsing the input structure. Without the precise memory modeling of this function, we cannot generate the precise path constraint and, thus, cannot generate effective inputs either. However, these limitations are not the problem we attempt to address in this work and we leave them as our future work. Nevertheless, we still triggered far more unique bugs than all the baseline fuzzers.

2) The real-world benchmark: We evaluated PANGOLIN on the real-world projects and examine whether PANGOLIN can also find more bugs. These projects receive different kinds of inputs such as images, binaries, and network packages.

Despite that these projects have been extensively evaluated by many state-of-the-art fuzzers, PANGOLIN still can discover new bugs. For instance, the subjects *libtiff*, *libjpeg*, *libpng*, and *binutils* have been fuzzed by OSS-fuzz[4] with a tremendous amount of inputs. Still, with few hours of running, PANGOLIN found 41 new bugs including buffer overflow, null pointer dereference, and exhaustive memory usage. All of the bugs were confirmed by the original developers and eight of them have been assigned with CVE IDs due to their security impacts. The results are shown in Table IV.

We further provided the details of the comparison in terms of the number of detected bugs and the number of unique crashes triggered by different fuzzers. Simply speaking, since a bug often can be triggered in multiple ways, a bug may correspond to multiple unique crashes. The unique crashes provide a more comprehensive picture of a bug to help developers fix it. Thus, evaluating both the number of bugs and the number of unique crashes are necessary and meaningful, just like the evaluation in the previous literature [9], [20]. In the evaluation, we filtered duplicate crashes by `afl-cmin -C`, followed by a manual analysis with the following standard: a crash is unique if and only if the path, which consists of multiple edges, to the crash point has a unique edge sequence.

As the results shown in Table V, we detected 33 and 29 more bugs than the hybrid fuzzer QSYM and the gray-box fuzzer Angora, respectively. PANGOLIN can also detect more bugs than other fuzzers. In terms of unique crashes, PANGOLIN also detected more in comparison with other fuzzers. Some crashes lead to severe problems, demonstrating the effectiveness of our approach. For example, we have detected two buffer overflows in *readelf* related to the incomplete fix issue of CVE-2017-9038 (details in Section V-D). Although it had been patched for more than two years, PANGOLIN can still detect two different feasible paths that can trigger this bug. This shows the effectiveness of the optimized input generation method in PANGOLIN. Note that T-Fuzz finds the same number of bugs as PANGOLIN on *djpeg* but fewer bugs in other benchmarks.

[4]https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

TABLE IV: Bugs detected by PANGOLIN.

	CVE/bugs	Bug Type
tcpdump	Patched	buffer overread * 2
tiff2ps	Confirmed	buffer oveflow * 6
	Pending	memory leak
readelf	CVE-2018-20623	use after free
	Confirmed	buffer overflow * 3
	Patched	integer overflow
objdump	Patched	null pointer dereference * 4
nm	CVE-2019-9070	buffer overflow
	Confirmed	buffer overread
bento	CVE-2018-20186	buffer overflow
	CVE-2019-6132	buffer overflow
	CVE-2019-13238	buffer overflow
	CVE-2019-13959	buffer overflow
	Confirmed	buffer overflow * 7
libjpg	CVE-2019-13960	exhaustive memory usage
libpng	Pending	buffer overflow * 2
jhead	CVE-2019-1010302	buffer overflow
	Confirmed	buffer overflow * 5

TABLE V: Unique crashes/bugs detected by PANGOLIN.

	PANGOLIN	AFL	AFLFAST	QSYM	Driller	Angora	T-Fuzz
tiff2ps	216(7)	22(1)	20(1)	3(1)	0	71(2)	0
readelf	350(5)	0	0	0	0	(2)	0
nm	18(2)	0	0	2(1)	0	0	0
objdump	169(4)	0	0	2(1)	0	1(1)	0
tcpdump	22(2)	0	0	0	0	0	0
libpng	36(2)	0	0	0	0	0	0
jhead	853(6)	63(1)	42(1)	551(2)	31(1)	580(3)	39(1)
bento	132(11)	0	3(1)	40(3)	42(3)	62(4)	0
djpeg	12(1)	0	0	0	0	0	11(1)

B. Coverage Comparison

Existing studies on fuzzing techniques often use the edge coverage criterion to measure the proportion of the program behaviors examined by a fuzzer. We followed this convention to evaluate the coverage rate of PANGOLIN and other baseline fuzzers. Figure 9 illustrates the number of edges discovered by different fuzzers against the running time. Note that we discovered some scalability issues when applying T-Fuzz to large-scale programs. Thus, the results of T-Fuzz are not included in Figure 9. We confirmed with the authors of T-Fuzz that it is currently fine-tuned for the CGC and LAVA-M benchmark programs and not for the programs used in our evaluation. Overall, PANGOLIN outperforms QSYM, and Angora, with the average coverage improvement 21.4% and 16.8%, within twenty-four hours. We used the one-tailed hypothesis for the Mann-Whitney U test to calculate the p-value of the results. There are two significance levels of the test: 0.01 and 0.05. The majority of the p-values are smaller than 0.01 which shows the results of PANGOLIN are significant compared with the other fuzzers. The project *jhead* is rather small with fewer edges so that PANGOLIN achieved the edge coverage with a small difference. The results were also influenced by randomness which, sometimes, makes the p-value larger than 0.01 but still smaller than 0.05.

TABLE VI: Average increase of the edge coverage comparing the two configurations of PANGOLIN and QSYM in 10 repeated experiments. The columns PANGOLIN(P) and PANGOLIN($\hat{P}$) denote PANGOLIN with and without guided solving, respectively. The p_1 stands for the p-value between P and $\hat{P}$. The p_2 stands for the p-value between $\hat{P}$ and QSYM.

	PANGOLIN(P)	PANGOLIN($\hat{P}$)	p_1	QSYM	p_2
readelf	10624(+26%)	9358(+11%)	0.00009	8402	0.00009
nm	4026(+38%)	3609(+24%)	0.00009	2909	0.00009
objdump	7492 (+21%)	6850 (+10%)	0.00009	6205	0.00009
djpeg	6147 (+21%)	5571 (+8%)	0.00009	5169	0.00009
pngimge	1861 (+10%)	1701(+3%)	0.00139	1651	0.00453
tiff2ps	5703 (+18%)	5212 (+8%)	0.00009	4824	0.00009
bento	1889 (+31%)	1640 (+14%)	0.00009	1439	0.00009
tcpdump	8973 (+19%)	8289 (+10%)	0.00009	7506	0.00009
jhead	2490 (+10%)	2343 (+3%)	0.00078	2261	0.02680

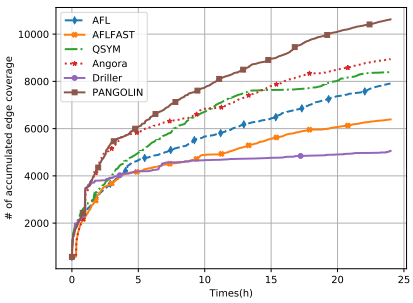
It should also be noted that PANGOLIN requires an initial warm-up time for generating the path abstractions, and, thus, may not perform the best at the very beginning. For example, for the program *objdump* in Figure 9c and *libjpg* in Figure 9d, Angora achieved better code coverage than PANGOLIN in the first five hours. However, with the path abstraction generated in the first few hours, PANGOLIN began to reap the benefits from the path-abstraction-guided input generation and achieved a higher coverage rate. We further monitored the fuzzing process and found out the reason behind it. These two programs divide the input into multiple sections. After loading the inputs, the programs start parsing these sections simultaneously. This design requires PANGOLIN to infer more path abstractions at the early stage, whereas, for later input generation, the fuzzer can take benefits from these generated path abstractions.

We remark that, compared with other hybrid fuzzing techniques, PANGOLIN takes advantage of the path-abstraction-based input generation method to achieve not only a higher coverage rate but also better bug detection capability. Specifically, the diverse inputs generated by the sampling method have more potentials to trigger the vulnerable program behaviors hidden behind complex path conditions. Taking QSYM as an example, it mainly aims to cover more branches and only generates one seed for each branch. This leads to some false negatives because covering a branch does not mean triggering the vulnerabilities in the branch.

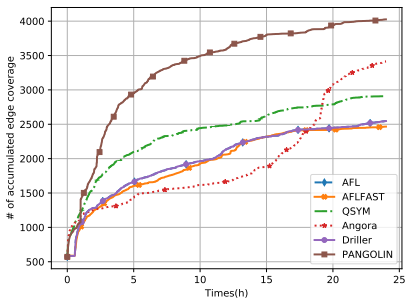
C. Key Feature Evaluation

To thoroughly understand PANGOLIN, we set up two more experiments to evaluate the two key techniques in PANGOLIN: the constrained mutation and the guided constraint solving.

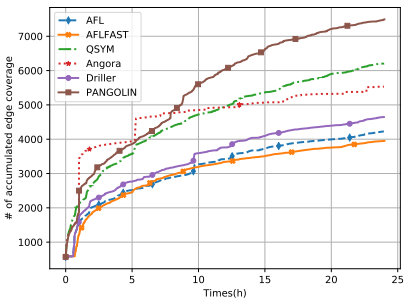
1) Guided constraint solving: To evaluate the effectiveness of guided constraint solving, we built an extra version of PANGOLIN without reusing the path abstraction to guide constraint solving and re-run the previous experiments. We compared with the original PANGOLIN using the following two metrics: the time cost of constraint solving; and the edge coverage rate.



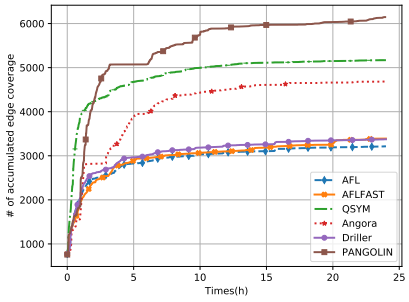
(a) readelf, $p_1 = 0.00009, p_2 = 0.00009$



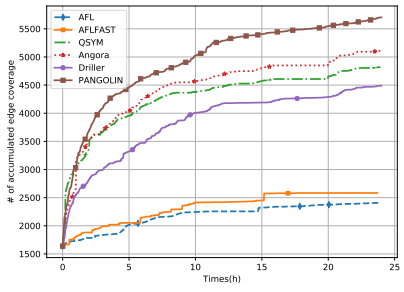
(b) nm, $p_1 = 0.00009, p_2 = 0.00009$



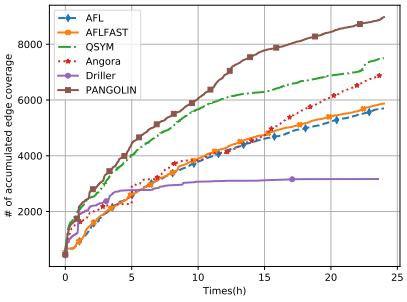
(c) objdump, $p_1 = 0.00009, p_2 = 0.00009$



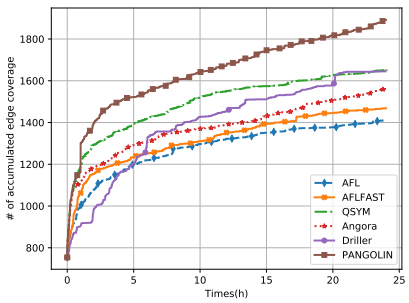
(d) libjpg, $p_1 = 0.00009, p_2 = 0.00009$



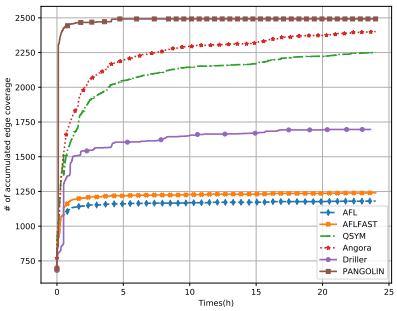
(e) tiff2ps, $p_1 = 0.00009, p_2 = 0.00009$



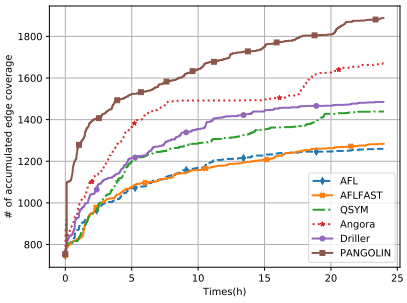
(f) tcpdump, $p_1 = 0.00009, p_2 = 0.00009$



(g) libpng, $p_1 = 0.00139, p_2 = 0.00009$



(h) jhead, $p_1 = 0.01578, p_2 = 0.03216$



(i) bento, $p_1 = 0.00009, p_2 = 0.00009$

Fig. 9: Average edge coverage(ten runs) in twenty four hours evaluation on real-world projects. The p_1 stands for the p-value compared with QSYM, the p_2 stands for the p-value compared with Angora.

Table VI demonstrates the overall comparison on the program coverage. PANGOLIN(P) and PANGOLIN($\hat{P}$) represent our tool with and without reusing the path abstraction in the constraint solving, respectively. Comparing PANGOLIN(P) with PANGOLIN($\hat{P}$), we observe that PANGOLIN(P) achieved 7% to 15% higher edge coverage rate. PANGOLIN($\hat{P}$) achieved 3% to 14% higher edge coverage rate than QSYM. All the p-values are smaller than 0.01 which shows the results are significant. Moreover, from Figure 10, it is clear that, reusing the path abstraction for constraint solving reduced the solving overhead from 20% to 30%. Similar to the coverage comparison, small projects such as $jhead$ with fewer constrains took fewer benefits (about 7%) from the guided constraint solving method.

Overall, the experimental results confirm that the path abstraction can successfully reduce the cost of constraint solving, and has a significant impact on the improvement of edge coverage rate. Since guided constraint solving can solve the path constraint efficiently, PANGOLIN(P) can reach the vulnerable program points earlier than conventional approaches and, thus, detect more bugs as shown in Figure 11.

2) The performance of constrained mutation: To evaluate the performance of constrained mutation using the path abstraction, we compared our implementation with the most recent SMT model sampling tool SMTSampler [51], which also aims to generate multiple solutions of a given path constraint by solving the constraint. For a fair comparison, we first exported the path constraints which were processed during the

TABLE VII: Evaluation results of constrained mutation. The column "N" stands for the average number of path constraints exported during fuzzing. The columns "T" stand for the average time costs of transforming the constraint into the path abstraction with different time budget. For each program, we calculated the average number of unique and feasible inputs n_t for all the constraints with different time budget $t = 3s, 5s, 10s$. The column "ratio" stands for the proportion of constraints which constrained sampling in PANGOLIN can generate more inputs than the SMTSampler in the program.

			n_3			n_5			n_{10}		
Programs	N	T	ratio	PANGOLIN	SMTSampler	ratio	PANGOLIN	SMTSampler	ratio	PANGOLIN	SMTSampler
readelf	721	1.3021	0.95	13889	9177	0.96	22288	14166	0.99	35689	22208
objdump	790	1.1682	0.96	3168	2020	0.96	5318	3123	0.99	9562	5173
nm	480	1.1262	0.96	12825	7827	0.98	18474	10016	1.00	29015	15542
libjpg	972	1.0415	0.98	2494	692	0.99	3508	1068	0.99	9226	2845
jhead	467	0.9074	0.97	7734	2067	0.98	10499	3974	0.99	22594	9424
bento	674	1.3009	0.87	12263	3400	0.93	18707	3870	0.94	22391	6649
tiff2ps	814	1.5493	0.96	9280	4542	0.96	11781	5549	0.99	13280	7030
libpng	460	1.1003	0.96	5883	3029	0.98	10507	5132	1.00	20662	14726
tcpdump	961	1.8821	0.97	3423	1720	0.98	5668	2935	0.99	11320	5902

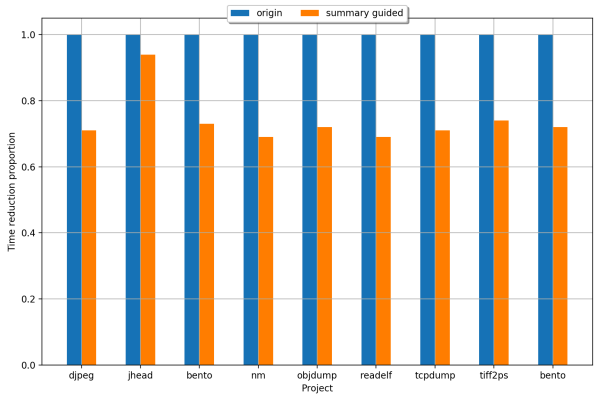


Fig. 10: Average proportion of time reduction comparing with the original solving in different real-world projects.

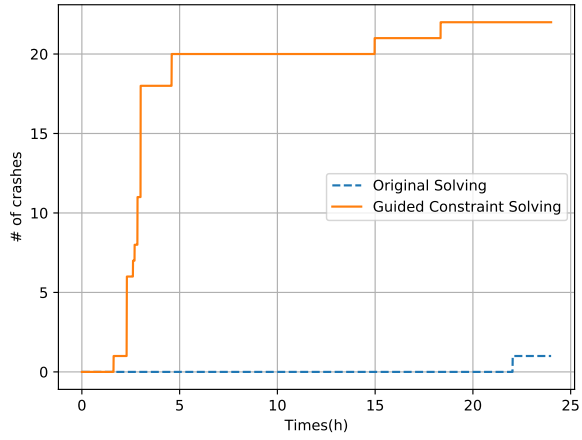


Fig. 11: Comparison of crash detection in *tcpdump* in 24 hours using original solving and guided constraint solving.

constrained mutation in PANGOLIN. Then, we leveraged our constrained mutation approach and SMTSampler to generate inputs that satisfy the exported constraints, respectively. We used the number of unique and feasible inputs generated within the given time budget to evaluate the performance of the two approaches. To be more specific, for each constraint, we gave the same time budget t for the two methods and evaluated the number of inputs n_t satisfying the constraint. We repeated each experiment five times and compared the average results using three sets of cut-off time t (3s, 5s, and 10s). If we could not infer the path abstraction within the given time, then we placed t for the solving time of this constraint.

The results are shown in Table VII. PANGOLIN outperforms SMTSampler almost in all the constraints. On average, PANGOLIN generated 93.78% more feasible inputs than SMTSampler. Meanwhile, our method performed better on more than 95% of the constraints within 3 seconds. This ratio increases along with the growth of the time budget since some constraints require a longer time for path abstraction inference. In our experiments on the nine real-world programs, we also found that over 99% of the constraints can be transformed into the path abstraction within 10 seconds and the average path abstraction inference time is 1.264s. This also shows the practical effectiveness of our method. Overall, our implemented constrained mutation achieves both efficiency and effectiveness compared with SMTSampler.

The key reason for this improvement is that our approach reduces the invocations of the constraint solver. Specifically, PANGOLIN only queries the constraint solver to obtain the path abstraction once, so as to narrow down the search space and offload the input generation to the sampling engine. Different from PANGOLIN, SMTSampler does not obtain the approximated search space of the feasible inputs and its mutation is based on the solutions generated by the solver. Therefore, SMTSampler has to frequently query the solver to generate multiple solutions. Moreover, the sampling algorithm in PANGOLIN (i.e., Dikin walk) guarantees the uniform distribution of the generated inputs while SMTSampler does not. The feature of uniform distribution improves the bug-finding capability [41].

```
1  //readelf binutils/drawf.c: processs_cu_tu_index
2  //sanitization, filter the malformed inputs
3  if ((size_t) nslots*8/8!=nslots
4    ||phash<phdr||phash>limit
5    ||pindex<phash||pindex>limit
6    ||ppool<pindex||ppool>limit
7    ....
8    version==2){
9      unsigned char*poffsets=ppool+ncols*4;
10   //sanitization, filter the malformed inputs
11   if(poffsets<ppool
12     ||(poffsets-ppool)/4!=ncols
13     ||type==1 ....) {
14   return;
15   }
16   // CVE-2017-9038 will be triggered if the
17   // value ncols is large.
18 }
19
```

Fig. 12: Incomplete fix of CVE-2017-9038.

D. Case Study

Among the bugs found by PANGOLIN, we pick an interesting and representative one which conveys our design intuitions. As mentioned above, the example shown in Figure 12 contains a buffer overflow vulnerability (CVE-2017-9038) related to the variables `ppool` and `ncols` at Line 16. The sanitization at Lines 11 - 13 can filter the majority of, but not all, malformed inputs that can trigger the vulnerability. Since covering a branch does not mean we can trigger a vulnerability, a bug-triggering input needs to satisfy a very complex path constraint to cover the branch and, meanwhile, satisfy the overflow constraint. Since it is expensive to solve complex constraints and it is also hard for a random mutation to generate inputs satisfying the complex bug-triggering constraint, conventional approaches like QSYM cannot detect it.

Our approach can quickly cover the vulnerable branch by leveraging the polyhedral abstraction of the path before Line 11 to accelerate the constraint solving process. After covering the branch, we leverage the constrained mutation to generate diversified inputs that can reach the vulnerable branch, thus having a high probability to trigger the bug. The evaluation results demonstrate that PANGOLIN successfully detected this bug very quickly.

VI. Related Work

To make a thorough comparison with related work, in this section, we discuss the differences between PANGOLIN and recent advances in gray-box fuzzing (Section VI-A), symbolic execution (Section VI-B), and hybrid fuzzing (Section VI-C).

A. Gray-box Fuzzing

Fuzzing becomes one of the most effective vulnerability detection methods nowadays. With the success of AFL, which can quickly explore a program through simple mutation, fuzzing has been developed and optimized to become more powerful. We discuss two optimization directions as below.

1) Seed prioritization and scheduling: Seed prioritization aims to find those seeds that are more likely to trigger new vulnerabilities. AFL [30] prioritizes seeds according to coverage information. The more uncovered branches a seed can cover, the higher priority the seed has. AFLFast [20] prioritizes the seeds based on the Markov chain. AFLGo [21] uses simulated annealing to calculate the distance of a seed towards the target program points. CollAFL [24] distinguishes the seeds with more precise coverage information to alleviate the collision problems. However, these scheduling methods are only based on the program structure but ignore the actual value space of related inputs. PANGOLIN not only uses the program structure information but also considers the value space with the path abstraction.

2) Mutation strategy: The other optimization direction is to take advantage of some advanced mutation strategies to generate inputs for complex path constraints. The basic idea is to only mutate the related inputs or input offsets to satisfy the uncovered branch conditions. Rather than random mutation, BuzzFuzz [52] uses a taint analysis to narrow down the mutation search space. It focuses on the input bytes that can flow into the potentially vulnerable points. Angora [9] adapts byte-level taint tracking to discover the related input bytes of the target condition, then applies a gradient-descent-based search strategy. To make the gradient-descent-based search more reasonable, Neuzz [28] proposes to use the neural network to smooth the search progress. There are also some techniques involving a lightweight program analysis and transformation to improve the effectiveness of mutation. Vuzzer [25] utilizes the so-called "magic-value-comparison" method to identify the input offsets where the values are not necessary to change and, thus, improves the efficiency of mutation. Steelix [53] splits a magic-value comparison into single-byte comparisons in order to increase the probability of generating proper inputs. Different from the mutation methods discussed above, the constrained mutation in our approach makes use of the path abstraction to generate diversified inputs for a covered branch, so that the branch is thoroughly tested. This method is orthogonal to the aforementioned existing approaches. We believe their combination will be more powerful.

B. Symbolic Execution

We discuss the incremental mechanisms in existing symbolic execution techniques from two aspects: the incremental mechanisms in SMT solvers and the incremental mechanisms specially designed for the symbolic execution engine.

1) Incremental mechanisms in SMT solvers: Modern SAT/SMT solvers have incremental solving facilities, which cache intermediate solving states such as the learned clauses [54] or final solutions such as the satisfying models or unsatisfiable cores [55] to accelerate the solving procedure of new constraints. However, these methods are not specially designed for fuzzing. Although the intermediate results they cache can accelerate constraint solving, it is hard to directly use the intermediate results to guide the mutation procedure. In comparison, the intermediate results in our approach, i.e.,

the path abstractions, can facilitate both the mutation and the constraint solving procedures.

2) Incremental mechanisms in symbolic execution engines: Symbolic executors can maintain different forms of cache explicitly. For instance, KLEE [56] caches the solved constraints to reduce unnecessary constraint solving for the same constraints. Instead of the original constraints, our approach caches a simplified form of the original path constraints, i.e., the path abstraction, to reduce the irrelevant mutation and accelerate the constraint solving procedure. The fork-server mode [56] aims to save the effort from reconstructing the system environment for a concolic engine. It cannot directly reuse previous solving results and needs to re-execute the program in a symbolic manner. Savior [6] and Driller [2] utilize the idea of fork-server mode to preserve the previous symbolic states. However, these methods aim to cache the states after the initialization of the concolic engine to avoid the time-consuming environment setup. They are orthogonal to PANGOLIN since path abstraction in our approach caches the intermediate results of the concolic execution.

C. Hybrid Fuzzing

Hybrid fuzzing combines the advantages of efficient mutation and precise constraint solving to evaluate the target programs. The original hybrid fuzzing [3] uses concrete values to aid the concolic execution so that the complexity of the path constraints can be reduced. With the development of fuzz testing, the majority of the path exploring demand offloads to the fuzzers to avoid the path explosion problem of concolic execution. In a word, the state-of-the-art hybrid fuzzing selectively solves the path constraints to improve the performance. For example, Driller [2] proposes to solve those uncovered paths for fuzzing rather than exploring all paths with concolic execution. Dowser [26] uses a constraint solver to generate input for the program paths potentially containing buffer-overflow vulnerabilities. T-Fuzz [46] uses symbolic execution to verify the infeasible path constraints and prune the related paths from the fuzzing scope.

However, how to effectively integrate concolic execution with fuzzing is always under consideration. For example, Pak [4] proposes to generate multiple inputs for the deepest branches that can be reached by symbolic execution and leverage fuzzing for the rest. DigFuzz [5] collects the execution frequency of each branch during fuzzing as the path prioritization metric for the constraint solver. SymFuzz [57] analyzes the path constraint to determine the number of bytes that a fuzzer needs to mutate each time. QSYM [1] solves part of the path constraint for a basis seed and leverages mutation for validated inputs satisfying the actual condition. These approaches are different from ours because none of them are incremental.

Using seeds to cache the progress may be adequate for fuzzing, but it cannot well-represent the progress in hybrid fuzzing. The fundamental factor in concolic execution, the analyzed path constraint, is not preserved for later fuzzing processes. The new seeds cannot represent the complex constraint, and the fuzzer still suffers from the inefficiency caused by the complex path constraint. Although there are some memory snapshot mechanisms attempting to preserve the program states [7], they are too expensive to improve the effectiveness of mutation in hybrid fuzzing [1]. In contrast, PANGOLIN optimizes this caching mechanism in hybrid fuzzing. We preserve a sustainable path abstraction, which is a simplified form of an analyzed constraint. The path abstraction can both guide the mutation in a fuzzer and reduce the computation in later concolic execution. Therefore, this cache-and-reuse mechanism optimizes the performance of hybrid fuzzing.

VII. CONCLUSION

We have presented PANGOLIN, the first incremental hybrid fuzzing framework using the polyhedral path abstraction to support effective constrained mutation and guided constraint solving. The incremental mechanism aids the fuzzer to effectively take the benefits from both worlds of fuzzing and concolic execution. Our evaluation results demonstrate that PANGOLIN outperforms the state-of-the-art fuzzers in terms of both the edge coverage rate and the bug-detection capability.

VIII. ACKOWNLEGEMENTS

The authors would like to thank the anonymous reviewers and the shepherd, Professor Mathias Payer, for their insightful comments. This work is partially funded by an MSRA grant, the Hong Kong GRF16230716, GRF16206517, ITS/215/16FP, ITS/440/18FP grants, and the NSFC Project No. 61902329. Qingkai Shi is the corresponding author.

REFERENCES

[1] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[2] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[3] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2007.41

[4] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," 2012.

[5] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[6] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: towards bug-driven hybrid testing," *CoRR*, vol. abs/1906.07327, 2019. [Online]. Available: http://arxiv.org/abs/1906.07327

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 380–394. [Online]. Available: https://doi.org/10.1109/SP.2012.31

[8] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 475–485. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238176

[9] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, May 2018, pp. 711–725. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00046

[10] Y. Chen, R. Dwivedi, M. J. Wainwright, and B. Yu, "Fast mcmc sampling algorithms on polytopes," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 2146–2231, 2018.

[11] R. Kannan and H. Narayanan, "Random walks on polytopes and an affine interior point method for linear programming," *Mathematics of Operations Research*, vol. 37, no. 1, pp. 1–20, 2012.

[12] H. Narayanan and A. Rakhlin, "Random walk approach to regret minimization," in *Advances in Neural Information Processing Systems*, 2010, pp. 1777–1785.

[13] Y. T. Lee and S. S. Vempala, "Geodesic walks in polytopes," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 927940. [Online]. Available: https://doi.org/10.1145/3055399.3055416

[14] Y. Chen, R. Dwivedi, M. J. Wainwright, and B. Yu, "Vaidya walk: A sampling algorithm based on the volumetric barrier," in *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2017, pp. 1220–1227.

[15] R. Kannan and H. Narayanan, "Random walks on polytopes and an affine interior point method for linear programming," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 561–570. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536491

[16] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2123–2138. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243804

[17] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 579–594.

[18] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2139–2154.

[19] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2095–2108. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243849

[20] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1032–1043. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[21] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2329–2344. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134020

[22] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[23] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2155–2168. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134073

[24] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018, pp. 679–696. [Online]. Available: https://doi.org/10.1109/SP.2018.00040

[25] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS*, Feb. 2017. [Online]. Available: https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

[26] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534772

[27] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[28] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program learning," *CoRR*, vol. abs/1807.05620, 2018. [Online]. Available: http://arxiv.org/abs/1807.05620

[29] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 116–122, 2018.

[30] "Afl: american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2014, accessed: 2014.

[31] "libfuzzer," https://llvm.org/docs/LibFuzzer.html, 2015, accessed: 2015.

[32] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[33] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[34] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1978, pp. 84–96.

[35] G. Singh, M. Püschel, and M. Vechev, "Fast polyhedra abstract domain," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 46–59. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009885

[36] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78. New York, NY, USA: ACM, 1978, pp. 84–96. [Online]. Available: http://doi.acm.org/10.1145/512760.512770

[37] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 504–515. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993558

[38] J. Jiang, L. Chen, X. Wu, and J. Wang, "Block-wise abstract interpretation by combining abstract domains with smt," in *Verification, Model Checking, and Abstract Interpretation*, A. Bouajjani and D. Monniaux, Eds. Cham: Springer International Publishing, 2017, pp. 310–329.

[39] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with smt solvers," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535857

[40] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νz - an optimizing smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 194–199.

[41] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of sat solutions for testing," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 549–559. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180248

[42] M. E. Dyer and A. M. Frieze, "On the complexity of computing the volume of a polyhedron," *SIAM Journal on Computing*, vol. 17, no. 5, pp. 967–974, 1988.

[43] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[44] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 110–121.

[45] "Oss-fuzz report," https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html, 2018, accessed: 2018-11-06.

[46] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, March 2014, pp. 323–332.

[47] "Shellphuzz: A python interface to afl, allowing for easy injection of testcases and other functionality." https://github.com/shellphish/fuzzer, 2015, accessed: 2015.

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[49] "Generated test cases for common image formats," http://lcamtuf.coredump.cx/afl/demo/, 2014, accessed: 2014.

[50] P. E. McKnight and J. Najab, *Mann-Whitney U Test*. American Cancer Society, 2010, pp. 1–1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470479216.corpsy0524

[51] R. Dutra, J. Bachrach, and K. Sen, "Smtsampler: Efficient stimulus generation from complex smt constraints," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2018, pp. 1–8.

[52] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 474–484. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070546

[53] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 627–637. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106295

[54] A. Yamada, T. Kitamura, C. Artho, E.-H. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental sat solving," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[55] A. Aquino, G. Denaro, and M. Pezz, "Reusing solutions modulo theories," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[56] C. Cadar, D. Dunbar, D. 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.

[57] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 725–741.