Matryoshka: Fuzzing Deeply Nested Branches

Peng Chen
ByteDance AI Lab
spinpx@gmail.com

Jianzhong Liu
ShanghaiTech University
liujzh@shanghaitech.edu.cn

Hao Chen
University of California, Davis
chen@ucdavis.edu

ABSTRACT

Greybox fuzzing has made impressive progress in recent years, evolving from heuristics-based random mutation to solving individual branch constraints. However, they have difficulty solving path constraints that involve deeply nested conditional statements, which are common in image and video decoders, network packet analyzers, and checksum tools. We propose an approach for addressing this problem. First, we identify all the control flow-dependent conditional statements of the target conditional statement. Next, we select the taint flow-dependent conditional statements. Finally, we use three strategies to find an input that satisfies all conditional statements simultaneously. We implemented this approach in a tool called Matryoshka[1] and compared its effectiveness on 13 open source programs with other state-of-the-art fuzzers. Matryoshka achieved significantly higher cumulative line and branch coverage than AFL, QSYM, and Angora. We manually classified the crashes found by Matryoshka into 41 unique new bugs and obtained 12 CVEs. Our evaluation demonstrates the key technique contributing to Matryoshka's impressive performance: among the nesting constraints of a target conditional statement, Matryoshka collects only those that may cause the target unreachable, which greatly simplifies the path constraint that it has to solve.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzzing, optimization, taint analysis, vulnerability detection

ACM Reference Format:
Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: Fuzzing Deeply Nested Branches. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3363225

[1]Matryoshka dolls are the set of wooden dolls of decreasing size placed one inside another, which emblematizes the deeply nested conditional statements that our tool can fuzz.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363225

1 INTRODUCTION

Fuzzing is an automated software testing technique that has successfully found many bugs in real-world software. Among various categories of fuzzing techniques, coverage-based greybox fuzzing is particularly popular, which prioritizes branch exploration to trigger bugs within hard-to-reach branches efficiently. Compared with symbolic execution, gray box fuzzing avoids expensive symbolic constraint solving and therefore can handle large, complex programs.

AFL [2] is a rudimentary greybox fuzzer. It instruments the program to report whether the current input has explored new states at runtime. If the current input triggers a new program state, then the fuzzer keeps the current input as a seed for further mutation [35]. However, since AFL mutates the input randomly using only crude heuristics, it is difficult to achieve high code coverage.

More recent fuzzers use program state to guide input mutation and showed impressive performance improvements over AFL, e.g., Vuzzer [30], Steelix [26], QSYM [41], and Angora [13]. Take Angora for example. It uses dynamic taint tracking to determine which input bytes flow into the conditional statement guarding the target branch and then mutates only those relevant bytes instead of the entire input to reduce the search space drastically. Finally, it searches for a solution to the *branch constraint* by gradient descent.

However, these fuzzers face difficulties when solving *path constraints* that involve nested conditional statements. A branch constraint is the predicate in the conditional statement that guards the branch. The branch is reachable only if (1) the conditional statement is reachable, *and* (2) the branch constraint is satisfied. A path constraint satisfies both these conditions. When a conditional statement s is nested, s is reachable only if some prior conditional statements P on the execution path are reachable. If the branch constraints in $\{s\} \cup P$ share common input bytes, then while the fuzzer is mutating the input to satisfy the constraint in s, it may invalidate the constraints in P, thus making s unreachable. This problem plagues the aforementioned fuzzers since they fail to track control flow and taint flow dependencies between conditional statements. Nested conditional statements are common in encoders and decoders for both images and videos, network packet parsers and checksum verifiers, which have a rich history of vulnerabilities. Though concolic execution may solve some nested constraints, Yun et al. showed that concolic execution engines can exhibit over-constraining issues, which makes it too expensive to solve the constraints [41], especially in real-world programs.

Figure 1 shows such an example in the program *readpng*. The predicate on Line 6 is nested inside the predicate on Line 4. [2] It is difficult for the fuzzer to find an input that reaches the false branch of Line 6 because the input has to satisfy the false branch of Line 4

[2]Although syntactically both Line 4 and Line 6 are at the same level, Line 6 is nested inside Line 4 in the control flow graph because the true branch of Line 4 is an immediate return.

```
1 // pngrutil.c, Line 2406
2 png_crc_read(png_ptr, buffer, length);
3 buffer[length] = 0;
4 if (png_crc_finish(png_ptr, 0) != 0)
5   return;
6 if (buffer[0] != 1 && buffer[0] != 2) {
7   png_chunk_benign_error(png_ptr, "invalid unit");
8   return;
9 }
```

Figure 1: An example showing a nested conditional statement on Line 6. It is difficult to find an input that reaches the false branch of Line 6 due to the check on Line 4.

```
1  void foo(unsigned x, unsigned y, unsigned z) {
2    if (x < 2) {
3      if (x + y < 3) {
4        if (z == 1111) {
5          if (y == 2222) { .... }
6          if (y > 1) { .... }
7        }
8      }
9    }
10 }
```

Figure 2: A program demonstrating nested conditional statements. Line 6 depends on Line 2, 3, and 4 by control flow, and on Line 2 and 3 by taint flow

.

as well. When a fuzzer tries to mutate the predicate on Line 6, it mutates only the input bytes flowing into `buffer[0]`, but this will almost surely cause the CRC check in `png_crc_finish()` to fail, which will cause Line 4 to take the true branch and return.

To evaluate whether current fuzzers have difficulty in solving path constraints involving nested conditional statements, we used Angora as a case study. We ran it on 13 open source programs, which read structured input and therefore likely have many nested conditional statements. Table 1 shows that on all the programs, the majority of unsolved path constraints involve nested conditional statements. On five of the programs, more than 90% of the unsolved constraints involve nested conditional statement.[3] This suggests that solving these constraints will improve the fuzzer's coverage significantly.

We design and implement an approach that allows the fuzzer to explore deeply nested conditional statements. The following uses the program in Figure 2 as an example. Suppose the current input runs the false branch of Line 6, and the fuzzer wishes to explore the true branch of Line 6.

(1) *Determine control flow dependency among conditional statements.* The first task is to identify all the conditional statements before Line 6 on the trace that may make Line 6 unreachable. They include Line 2, 3, and 4, because if any of them takes a different branch, then Line 6 will be unreachable. Section 3.3 will describe how we use intraprocedural and interprocedural post-dominator trees to find those conditional statements.

(2) *Determine taint flow dependency among conditional statements.* Among the conditional statements identified in the previous step, only those on Line 2 and 3 have taint flow dependencies with Line 6. This is because when we mutate y on Line 6, this may change the branch choice of Line 3 and hence making Line 6 unreachable. To avoid this problem, we must keep the branch choice of Line 3, which may require us to mutate both x and y, but this may change the branch choice of Line 2. Therefore, Line 2 and 3 have taint flow dependencies with Line 6. By contrast, the branch choice of Line 4 never changes as we mutate y to explore the true branch of Line 6, so it has no taint flow dependencies with Line 6. Section 3.4 will describe how we find those taint flow dependent conditional statements.

(3) *Solve constraints.* Finally, we need to mutate the input to satisfy several dependent conditional statements simultaneously. In other words, we need to find a new input that both reaches Line 6 and satisfies its true branch. We propose three strategies.
   - The first strategy conservatively assumes that if we mutate any byte flowing into any conditional statements that Line 6 depends on, then Line 6 will become unreachable. So this strategy avoids mutating those bytes when fuzzing Line 6.[4] (Section 3.5.1)
   - The second strategy artificially keeps the branch choices of all the conditional statements that Line 6 depends on when mutating the input bytes that flow into Line 6. When it finds a satisfying input, it verifies whether the program can reach Line 6 without altering branch choices. If so, then the fuzzer successfully solves this problem. Otherwise, the fuzzer will backtrack on the trace to try this strategy on Line 3 and Line 2. (Section 3.5.2)
   - The last strategy tries to find a solution that satisfies all the dependent conditional statements. It defines a joint constraint that includes the constraint of each dependent conditional statement. When the fuzzer finds an input that satisfies the joint constraint, then the input is guaranteed to satisfy the constraints in all the dependent conditional statements. (Section 3.5.3)

Our approach assumes no special structure or property about the program being fuzzed, such as magic bytes or checksum functions. Instead, our general approach to solving nested conditional statements can handle those special structures naturally.

We implemented our approach in a tool named Matryoshka and compared its effectiveness on 13 open source programs against other state-of-the-art fuzzers. Matryoshka found a total of 41 unique new bugs and obtained 12 CVEs in seven of those programs. Matryoshka's impressive performance is due not only to its ability to solve nested constraints but also to how it constructs these constraints. Traditional symbolic execution collects the predicates in

[3] Some conditional statements depend on other conditional statements by control flow, but they do not share input bytes.

[4] This strategy fails to work on this example because the fuzzer is left with no input byte to mutate.

all the conditional statements on the path. By contrast, Matryoshka collects the predicates in only those conditional statements that the target branch depends on by both control flow *and* taint flow. Our evaluation shows that the latter accounts for only a small fraction of all the conditional statements on the path, which greatly simplifies the constraints that Matryoshka has to solve.

2 BACKGROUND

Greybox fuzzing is a popular program testing method that incorporates program state monitoring with random input mutation to great effect. However, current state-of-the-art greybox fuzzers are unable to reliably and efficiently solve nested conditional statements. Fuzzers using either heuristics (e.g., AFL) or principled mutation methods (e.g., Angora) do not have enough information about control flow and taint flow dependencies between conditional statements to devise an input that can satisfy all the relevant branch constraints. Other fuzzers utilizing hybrid concolic execution such as Driller experience performance hits due to concretizing the entire symbolic constraints of a path [41, 42]. QSYM is a practical concolic execution fuzzer, but it is tailored to solve only the last constraint on a path, thus facing the same challenge of solving nested conditional statements as Angora.

Using Angora as an example, we evaluated the impact of nested conditional statements on Angora's performance and analyzed the constraints in eight programs that Angora failed to solve in Table 1, where each constraint corresponds to a unique branch in the program. The second column shows what percentage of the unsolved constraints are nested, which depend on other conditional statements by control flow and taint flow (Section 3.4). The third column shows what percentage of all the constraints, both solved and unsolved, are nested. Table 1 shows that the majority of the unsolved constraints are nested, ranging from 57.95% to nearly 100%. It also shows that nested constraints account for a substantial portion of all the constraints, ranging from 44.14% to 89.50%. These results suggest that solving nested constraints could improve the coverage of greybox fuzzers substantially.

Table 1: Percentage of nested constraints encountered by Angora

Program	Percentage of nested constraints in	
	all unsolved constraints	all constraints
djpeg	90.00 %	75.65 %
file	86.49 %	44.14 %
jhead	57.95 %	51.53 %
mutool	80.88 %	58.63 %
nm	84.32 %	68.16 %
objdump	90.54 %	73.95 %
readelf	84.12 %	70.50 %
readpng	94.02 %	89.50 %
size	87.86 %	71.46 %
tcpdump	96.15 %	78.98 %
tiff2ps	75.56 %	62.18 %
xmlint	78.18 %	72.37 %
xmlwf	96.18 %	68.16 %

3 DESIGN

3.1 Problem

State-of-the-art coverage-guided fuzzers, e.g., Angora [13], QSYM [41], VUzzer [30] and REDQUEEN [4], explore new branches by solving branch constraints, where a branch constraint is the predicate in the conditional statement that guards the branch. This typically involve the following steps. First, identify the input bytes that affect each conditional statement using dynamic taint analysis or similar techniques. Then, determine how the input bytes should be mutated, such as calculating the gradient of the predicate and using gradient descent, matching magic bytes or resorting to using a symbolic execution solver. Finally, execute the program with the mutated input and verify if this triggers the other branch in the conditional statement.

Although this approach is effective in solving many branch constraints, it fails when the target conditional statement becomes unreachable during input mutation.

Figure 2 shows an example. Let the variables `x`, `y`, and `z` contain different input bytes. Assume that the current input executes the false branch of Line 6, and the goal is to explore the true branch of Line 6. Then, the fuzzer determines, by dynamic byte-level taint analysis, that it needs to change the bytes in y. Consider two different initial values of x and y.

(1) $x = 0$ and $y = 1$. If the fuzzer mutates y to 3, then the program will no longer reach Line 6 because Line 3 will take a different (false) branch. This renders the fuzzer helpless when solving the branch predicate, even though a satisfying assignment $y = 2$ exists.
(2) $x = 1$ and $y = 1$. In this case, no value of y can satisfy the true branches of Line 2, Line 3, and Line 6 simultaneously, unless we also mutate x. However, since x does not flow into the conditional statement on Line 6, the fuzzer does not know that it should mutate x, so it can never find a satisfying assignment to explore the true branch of Line 6, regardless of the algorithm used to solve the constraint.

This example shows that to execute an unexplored branch, it is sometimes inadequate to mutate only the input bytes that flow into the conditional statement because doing so might render this statement unreachable. One could naively mutate all the input bytes, but that would increase the search space by many magnitudes to make this approach too expensive to be practical.

3.2 Solution overview

To overcome the problem in Section 3.1, our key insight is that when we fuzz a conditional statement, we must find an input that both satisfies the branch constraint and keeps the statement reachable. Most fuzzers that explore branches by solving branch constraints consider only the satisfiability criterion but fail to consider the reachability criterion. We propose the following steps to satisfy both criteria while mutating the input. Let s be a conditional statement on the trace of the program on this input. Our goal is to mutate the input to let s take a different branch. We call s the *target conditional statement* and say that the new input *satisfies* s.

(1) *Determine control flow dependencies among conditional statements.* Identify all the conditional statements before s on the trace that may make s unreachable. For example, if s is on Line 6 in Figure 2, then if any of the conditional statements on Line 2, 3, and 4 takes a different branch, then Line 6 will be unreachable. We call these the *prior conditional statements* of s, which s depends by control flow. By contrast, no matter which branch Line 5 takes, Line 6 will always be reachable. Section 3.3 will describe this step in detail.
(2) *Determine taint flow dependency among conditional statements.* Among the prior conditional statements of s, identify those whose corresponding input bytes may have to be mutated to satisfy s. For example, let s be Line 6 in Figure 2. Among its three prior conditional statements, only those on Line 2 and 3 contain bytes (x and y) that may have to be mutated to satisfy s. We call these *effective prior conditional statements*, which s depends on by taint flow. By contrast, Line 4 contains no input bytes that may have to be mutated to satisfy s. Section 3.4 will describe this step in detail.
(3) *Solve constraints.* Mutate the bytes in the effective prior conditional statements to satisfy s. Section 3.5 will describe this step in detail.

Figure 3 shows an overall design of Matryoshka regarding how the strategies are used in the fuzzing process.

3.3 Determine control flow dependency among conditional statements

For each conditional statement s, we wish to identify all its *prior conditional statements*, which are the conditional statements that, if taking a different branch, may cause s to be unreachable. Let the *immediate prior conditional statement* of s on a trace be the last prior conditional statement of s, i.e., there is no prior conditional statement of s between r and s. Note that if s is a prior conditional statement of t, and t is of u, then s is a prior conditional statement of u. This allows us to find all the prior conditional statements of s transitively: starting from s, we repeatedly find the immediate prior conditional statement, and then take the union of all such statements.

We propose two different methods for finding the immediate prior statement that is in the same function and that is in a different function, respectively. In our implementation for optimization, we cached all the found dependencies to avoid repeated computation.

3.3.1 Intraprocedural immediate prior conditional statement. Starting from a conditional statement s, we walk back on the trace. When we find the first conditional statement r

- that is in the same function, and
- that s does *not* post-dominate [1]

then r is the immediate prior statement of s. Our implementation used the post-dominator trees produced by LLVM [24].

If we cannot find such r, then s has no intraprocedural immediate prior conditional statement, and we will search for its interprocedural immediate prior conditional statement, to be described in Section 3.3.2.

3.3.2 Interprocedural immediate prior conditional statement. It would be straightforward to use interprocedural post-dominator trees for efficient handling, but unfortunately, LLVM does not provide such information, so we designed the following method for finding the interprocedural immediate prior conditional statement of s. Starting from s, we walk back on the trace to find the first conditional statement r that satisfies all the following:

(1) r is in a different function (let us call it f_r) than s, and
(2) f_r is still on the stack (i.e., it hasn't returned) when s is executing, and
(3) Let r_c be the last call instruction that f_r executed. r_c must exist because r is in a deeper stack frame than s. If r_c does not post-dominate r (note that r and r_c are in the same function), then r is the interprocedural immediate prior statement of s.

3.3.3 Irregular interprocedural control flow. Apart from function calls, the program could also exhibit irregular interprocedural control flows, for instance those involving EXIT and LONGJMP instructions. If a conditional statement r has at least one branch that leads to a basic block that contains irregular flows, then we consider it to be the prior conditional statement of all the statements after itself even when its frame is not on the stack. If s is a conditional statement after r, we add r and r's prior conditional statements to the set of s's prior conditional statements. In LLVM, the basic blocks containing irregular interprocedural control flows are terminated with UNREACHABLE instructions.

3.4 Determine taint flow dependency among conditional statements

For each conditional statement s, Section 3.3 finds all its prior conditional statements $p(s)$. Let $b(s)$ be the set of input bytes that flow into s where s is one or more conditional statements. When we mutate the input, as long as no conditional statement in $p(s)$ takes a different branch, s is guaranteed to be reachable. This seems to suggest that we should avoid mutating any byte in $b(p(s))$.

On the other hand, avoid mutating every byte in $b(p(s))$ may prevent the fuzzer from finding a satisfying assignment for s, as discussed in Section 3.1. Take Figure 2 as an example. Let s be Line 6. By Section 3.3.1, we determine that $p(s)$ consists of Lines 2, 3, and 4. Therefore, $b(p(s)) = \{x, y, z\}$. If we keep all the bytes in $b(p(s))$ immutable, then we are left with no input byte to mutate when trying to find an input to satisfy s.

The problem arises because Section 3.3 considers only control flow dependency among conditional statements, but it fails to consider whether taint flow dependencies exist between the conditional statements. We define the *effective prior conditional statements* of s, $e(s)$, to be a subset of the prior conditional statements of s, where to find an input to satisfy s, we may have to mutate some bytes flowing into a statement in $e(s)$. In other words, if a prior conditional statement r of s is not also an effective prior conditional statement of s, then no byte flowing into r needs to be mutated to satisfy s. This means that we may consider only the effective prior conditional statements and ignore the non-effective prior conditional statements.

Algorithm 1 shows the algorithm for computing effective prior conditional statements, which relies on the following property: if r is an effective prior conditional statement of s, and q is a prior

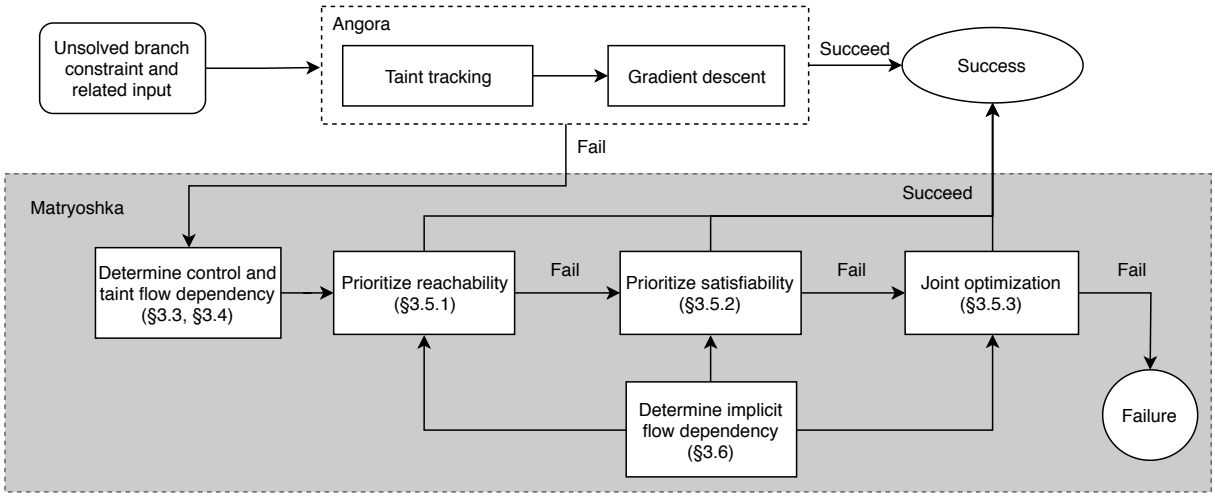


Figure 3: Overview of Matryoshka. In the figure, Angora represents any fuzzer capable of identifying constraints. When the fuzzer fails to solve a branch constraint guarding a new branch, Matryoshka determines whether the conditional statement is nested. If so, Matryoshka tries three optimization strategies: prioritizing reachability, prioritizing satisfiability, and joint optimization, during which it also identifies implicit flow dependencies when necessary.

conditional statement of s, and q and r share common input bytes, then q is also an effective prior conditional statement of s.

Algorithm 1 Find effective prior conditional statements

```
1:  function FindEffectivePriorCondStmt(s, stmts)  ▷
    s: conditional statement being fuzzed; stmts: prior conditional
    statements of s. Returns: effective prior conditional statements
    of s
2:      Initialize a union-find data structure.
3:      for all stmt ∈ stmts do
4:          T ← input bytes flowing into stmt.
5:          Union all t ∈ T.
6:      end for
7:      O ← ∅
8:      b_s ← any one byte flowing into s
9:      for all stmt ∈ stmts do
10:         b ← any one byte flowing into stmt
11:         if Find(b_s) == Find(b) then
12:             O ← O ∪ {stmt}
13:         end if
14:     end for
15:     return O
16: end function
```

3.5 Solve constraints

Section 3.4 determines the effective prior conditional statements for each conditional statement s. On the one hand, if we freely mutate the bytes flowing into any of them, s may become unreachable. But on the other hand, we may be required to mutate some of those bytes to satisfy the unexplored branch of s. Therefore we need to determine which of those statements whose relevant input bytes we may mutate. We propose the following three alternative strategies. Matryoshka tries them in this order and imposes a time budget for each strategy to ensure overall efficiency.

(1) Prioritize reachability
(2) Prioritize satisfiability
(3) Joint optimization for both reachability and satisfiability

Each strategy identifies a constraint over a set of input bytes. Then, it uses gradient-based optimization to solve the constraint. These strategies provide benefits only when s is nested, i.e., s has effective prior conditional statements. If s is not nested, Matryoshka simply uses existing strategies from Angora or other fuzzers to solve the branch constraint. Therefore, Matryoshka exhibits better performance in solving nested conditional statements while having the same ability as other fuzzers to solve non-nested conditional statements.

3.5.1 Prioritize reachability. This strategy pessimistically assumes that if we mutate any byte that flows into any effective prior conditional statements of a conditional statement s, s may become unreachable. Therefore, this strategy ensures that s is always reachable by avoiding mutating any byte that flows into any of s's effective prior conditional statements. Formally, let $b(s)$ be the bytes flowing into s, and $b(e(s))$ be the bytes flowing into s's effective prior conditional statements. Angora mutates all the bytes in $b(s)$, which may cause s to become unreachable. By contrast, this strategy of Matryoshka mutates only the bytes in $b(s) \setminus b(e(s))$, i.e. all the bytes that flow into s but that do not flow into any effective prior statement of s.

Take the program in Figure 2 for example. When we fuzz s on Line 3, its only effective conditional statement is t on Line 2. $b(s) = \{x, y\}$. $b(e(s)) = \{x\}$. Using this strategy, the fuzzer mutates only the bytes in $b(s) \setminus b(e(s)) = \{y\}$.

However, this strategy fails when we fuzz s on Line 6. In this case, its effective prior statements consist of the statements on

Line 2 and 3, so $b(s) = \{y\}, b(e(s)) = \{x, y\}$, but $b(s) \setminus b(dp(s)) = \emptyset$. Using this strategy, Matryoshka will fail to fuzz s because it finds no byte to mutate.

Algorithm 2 Find a satisfying input while prioritizing satisfiability

```
1: function FindInput(s, stmts)        ▷ s: the target conditional statement. stmts: effective prior conditional statements of s.
2:                                     ▷ Forward phase
3:     While keeping the branch choice of each r ∈ stmts, find an input i that satisfies the target branch of s.
4:     Run the program on i.
5:     if s's target branch is reachable then
6:         return Success
7:     end if
8:                                     ▷ Backtracking phase
9:     B_I ← ∅                         ▷ Input bytes not to be mutated.
10:    for stmt ∈ stmts in the reverse order on the trace do
11:        B ← input bytes flowing into stmt
12:        B_2 ← B \ B_I
13:        While keeping the branch choice of all r ∈ stmts where r appears before stmt on the trace, find an input i that satisfies the target branch of stmt, during which only the input bytes in B_2 may be mutated.
14:        Run the program on i.
15:        if stmt's target branch is reachable then
16:            return Success
17:        end if
18:        B_I ← B_I ∪ B
19:    end for
20:    return Failure
21: end function
```

3.5.2 Prioritize satisfiability. This strategy optimistically hopes that a mutated input that satisfies a conditional statement s can also reach s. It has a forward phase followed by a backtrack phase. During the forward phase, it mutates the bytes flowing into s while artificially keeping the branch choices of all the effective prior conditional statements of s, thereby guaranteeing that s is always reachable. If it finds an input that satisfies the target branch of s, it runs the program on that input normally (without artificially fixing branch choices). If this trace still reaches s and chooses the target branch, it succeeds. Otherwise, it enters the backtrack phase. During this phase, it starts from s and then goes backward to fuzz each of the effective prior statements of s in that order. When it fuzzes one such statement r, it avoids mutating any byte that may flow into s or any effective prior conditional statement of s that is after r. The process succeeds when the fuzzer successfully fuzzes all of these effective prior conditional statements. Algorithm 2 shows this algorithm.

Take the program in Figure 2 as an example. When we fuzz s on Line 6, its effective prior conditional statements are on Line 3 and 2. Let the current input be $x = 1, y = 1$. Under this input, both Line 2 and 3 take the true branch, and Line 6 takes the false branch. Our goal is to take the true branch on Line 6. Using this strategy, during the forward phase, the fuzzer mutates y while artificially forcing the program to take the true branch on both Line 2 and 3. If the fuzzer finds an assignment $y = 2$ to satisfy the true branch of Line 6, but since $x = 1, y = 2$ does not satisfy Line 3, it enters the backtracking phase. During this phase, it will first fuzz Line 3. Although this line is affected by two values $\{x, y\}$, since y flows into Line 6, the fuzzer will mutate x only. If it finds a satisfying assignment $x = 0$, it tries to run the program with $x = 0, y = 2$ without artificially forcing branch choices. Since this input reaches Line 3 and satisfies the target (true) branch, fuzzing succeeds.

By contrast, let us assume that the fuzzer finds a satisfying assignment $y = 3$ when fuzzing Line 6. During the backtrack phase, when fuzzing Line 3, since it can mutate only x, it cannot find a satisfying assignment. Therefore, fuzzing of s fails.

3.5.3 Joint optimization for both reachability and satisfiability. Both strategies in Section 3.5.1 and Section 3.5.2 search for a solution to one constraint at a time. Section 3.5.1 mutates only the input bytes that will not make the target conditional statement unreachable, while Section 3.5.2 tries to satisfy the target conditional statement and its effective prior conditional statements one at a time. However, they fail to find a solution where we must jointly optimize multiple constraints.

Let s be the target conditional statement. Let $f_i(\boldsymbol{x}) \leq 0, \forall i \in [1, n]$ represent the constraints of the effective prior conditional statements of s, and $f_o(\boldsymbol{x}) \leq 0$ represent the constraint of s. $\boldsymbol{x}$ is a vector representing the input bytes. Table 2 shows how to transform each type of comparison to $f \leq 0$. Our goal is to find an $\boldsymbol{x}$ that satisfies all $f_i(\boldsymbol{x}) \leq 0, i \in [0, n]$. Note that each $f_i(\boldsymbol{x})$ is a blackbox function representing the computation on the input $\boldsymbol{x}$ by the expression in the conditional statement i. Since the analytic form of $f_i(\boldsymbol{x})$ is unavailable, many common optimization techniques, such as Lagrange multiplier, do not apply.

We propose a solution to the optimization problem. Define

$$g(\boldsymbol{x}) = \sum_{i=0}^{n} R(f_i(\boldsymbol{x})) \tag{1}$$

where the rectifier $R(x) \equiv 0 \vee x$ (the binary $\vee$ operator outputs the larger value of its operands). Therefore, $g(\boldsymbol{x}) = 0$ only if $f_i(\boldsymbol{x}) = 0, \forall i \in [0, n]$. In other words, we combined the n optimizations into one optimization. Now we can use the gradient descent algorithm, similar to the one used by Angora, to find a solution to $g(\boldsymbol{x}) = 0$. Note that when we compute the gradient of $g(\boldsymbol{x})$ using differentiation, we need to artificially keep the branch choices of the effective prior conditional statements of s to ensure that s is reachable.

Let us revisit the program in Figure 2. Let $[x, y] = [1, 3]$ be the initial input. When we fuzz the target conditional statement s on Line 6 to explore the true branch, we cannot solve the branch constraint by mutating only y. Using joint optimization, we combine the branch constraints of s and its effective prior conditional statements on Line 3 and 2 to construct (by Equation 1 and Table 2):

$$g([x, y]) = R(x - 2 + \epsilon) + R(x + y - 3 + \epsilon) + R(1 - y + \epsilon)$$

where $\epsilon = 1$. On the initial input $[x, y] = [1, 3]$, $g([x, y]) = 2$. Using gradient descent, we will find a solution to $g([x, y]) = 0$ where $[x, y] = [0, 2]$.

Table 2: Transform a predicate into a function such that the predicate is satisfied when the function is non-positive. ϵ is the smallest positive value of the type for a and b. For integers, $\epsilon = 1$.

Predicate	$f()$
$a < b$	$a - b + \epsilon$
$a \leq b$	$a - b$
$a > b$	$b - a + \epsilon$
$a \geq b$	$b - a$
$a = b$	$\text{abs}(a - b)$
$a \neq b$	$-\text{abs}(a - b) + \epsilon$

```
1  void bar(int y, int z) {
2      int k = 0, n = 0;
3      if (z - y == 56789) {
4          k = 1; n = 1;
5      }
6      if (k == 1) {
7          if (z == 123456789) { .... }
8      }
9  }
10
11 void foo(int x, int y, int z) {
12     void (*fun_ptr)(int, int) = NULL;
13     if (z - x == 12345) {
14         fun_ptr = &bar;
15     } else {
16         fun_ptr = &other_fn;
17     }
18     (*fun_prt)(y, z);
19 }
```

Figure 4: A program showing implicit control and taint flow dependencies

3.6 Detect implicit effective prior conditional statements

The mutation strategies in Section 3.5 may fail if we cannot find all the control flow and taint flow dependencies among conditional statements. Section 3.3 and Section 3.4 described algorithms for finding all the *explicit* control flow and taint flow dependencies, respectively. However, they are unable to find *implicit* flows. Figure 4 shows such an example. The conditional statement on Line 13 causes an implicit taint flow into `fun_ptr` in function `foo`, which then implicitly determines the control flow whether the program will call the function `bar` or `other_fn`. Also, Line 3 causes an implicit taint flow into the variable `k`, whose value will determine the value of the predicate on Line 6. Therefore, both the conditional statements on Line 13 and Line 3 should be effective prior conditional statements for the target statement on Line 7. However, since the taint flow is implicit, the algorithms in Section 3.4 cannot find them.

Implicit taint flows may be identified using control flow graphs [23]. If a predicate is tainted, then the method taints all the variables that get new values in either branch of the conditional statement. For byte-level taint tracking, this method adds the taint label of the predicate to each of the above variables. For example, consider the predicate on Line 3 in Figure 4. Since the variables `k` and `n` are assigned new values in a branch of this conditional statement, this method adds the taint label of the predicate (i.e., the taint label of the variable `y`) to the variable `k` and `n`. However, this method often results in over taint or taint explosion, because it may add taint labels that will be useless to the analysis. For example, in the example above, while the taint label added to the variable `k` captures the implicit taint flow dependency from Line 3 to Line 6, the taint label added to the variable `n` is useless because it does not help identify new taint flow dependencies between conditional statements. Even worse, these useless taint labels will propagate further to other parts of the program, resulting in taint explosion.

We propose a novel approach to identify implicit control flow and taint flow dependencies between conditional statements without incurring either huge analysis overhead or taint explosion. The insight is that rather than identifying all the implicit flows, we need to identify only those that cause the target conditional statement to become unreachable during input mutation. Let s be the target conditional statement that was reachable on the original input but became unreachable on the mutated input. We run the program twice. First, we run the program on the original input and record the branch choices of all the conditional statements on the path before s. Then, we run the program on the mutated input with a special handling: when we encounter a conditional statement, we record its branch choice but force it to take the branch choice as in the previous run (on the original input). Therefore, the paths of the two runs have the same sequence of conditional statements. We examine all the conditional statements on the path from the start of the program to s in the reverse chronological order. For each such statement t, if it is not already an explicit effective prior statement identified by the algorithms in Section 3.4 and if its branch choices in the first run (on the original input) and the second run (on the mutated input) differ, it has a potential control flow or taint flow dependency with s. To test whether this dependency truly exists, we run the program on the mutated input with a special handling: we force all the following conditional statements to take the branch choice as in the first run:

(1) all the conditional statements before t on the path
(2) all the explicit effective prior conditional statements
(3) all the implicit effective prior conditional statements

If the program no longer reaches s, then t truly has implicit control flow or taint flow dependency with s, and we mark it as an implicit effective prior conditional statement of s.

The complexity of this algorithm is linear in the number of conditional statements before s that are affected by the mutated bytes but are not control-flow-wise dependent on s. However, since Matryoshka mutates inputs by gradient descent on a small proportion of the input, the number of statements we should test is likely to be few.

4 IMPLEMENTATION

We implemented Matryoshka in 8672 lines of Rust, and 1262 lines of C++ for LLVM pass. We borrowed from Angora the code for

byte-level taint tracking and for mutating the input by gradient descent [3]. When computing intraprocedural post dominator trees, Matryoshka uses LLVM's function pass `PostDominatorTreeWrapperPass` [24].

5 EVALUATION

We evaluated Matryoshka in three parts. In the first part, we compared Matryoshka with nine other fuzzers on the LAVA-M data set [14]. Next, we compared Matryoshka with three other most representative fuzzers on 13 open source programs. Finally, we evaluated how Matryoshka's ability to solve nested constraints contributes to its impressive performance.

We ran our experiments on a server with two Intel Xeon Gold 5118 processors and 256 GB memory running 64-bit Debian 10.[5] Even though Matryoshka can fuzz a program using multiple cores simultaneously, we configured it to fuzz the programs using only one core during evaluation. We ran each experiment five times and reported the average performance.

5.1 Comparison on LAVA-M

LAVA-M consists of four programs with a large number of injected but realistic looking bugs [14]. It has been widely used for evaluating fuzzers. However, it is approaching the end of its shelf life as the state of the art fuzzers (Angora and REDQUEEN) were able to find almost all the injected bugs in LAVA-M. While LAVA-M cannot show that Matryoshka advances the state of the art, it can show whether Matryoshka is at the state of the art.

Table 3 compares the number of bugs found by 10 fuzzers. Matryoshka and REDQUEEN are the best: they both found almost all the listed bugs in LAVA-M.[6]

5.2 Comparison on 13 open source programs

We compared Matryoshka with AFL, QSYM and Angora by line and branch coverage. We ran them on 13 open source programs shown in Table 4. We chose these programs because eight of them were used for evaluating Angora, and the rest were used frequently for evaluating other fuzzers.

5.2.1 Program coverage and efficiency. We compared line and branch coverage of AFL (1 Master + 1 Slave), Angora (+1 AFL Slave), QSYM (+ 1 AFL Slave) with and without optimistic solving, and Matryoshka (+1 AFL Slave). Table 5 shows the coverage after running AFL, Angora, QSYM, QSYM with optimistic solving disabled, and Matryoshka on two CPU cores for 24 hours (one core for AFL slave). Matryoshka outperformed AFL, QSYM, and Angora on all the programs, except on *xmlwf*, *mutool*, and *tiff2ps* where Matryoshka had similar performance with Angora. Matryoshka's advantage shines the most on *xmllint*, where Matryoshka increased line and branch coverage by 16.8% and 21.8%, respectively, over Angora, the fuzzer with the next highest coverage.

Figure 5 compares the cumulative line and branch coverage by AFL, Angora, and Matryoshka on the program *readpng* over time. Matryoshka covered more lines and branches than QSYM and Angora at all time, thanks to its ability to solve nested conditional statements.

The goal of coverage-based fuzzers is to increase coverage, as measured by cumulative line and branch coverage. By contrast, the number of tests generated and executed by the fuzzer per second has no correlation with either line or branch coverage across different fuzzers, because smart fuzzers may generate fewer tests but the tests are much more effective in triggering new branches.

5.2.2 Bug analysis, verification, and classification. Besides all the inputs that crashed the program during fuzzing, we also ran AddressSanitizer(ASAN) [31] on all the seeds found by Matryoshka and saved the inputs where ASAN reported errors. Then, we deduplicated the crashes by AFL's `afl-cmin -C` command.

We manually verified all the crashes and classified them into unique bugs shown in Table 6. Matryoshka found a total of 41 unique bugs in seven programs (it found no bugs in the rest six programs). We have reported all the bugs to the developers and 12 of them have been assigned CVE IDs.

5.3 Novel features of Matryoshka

We evaluated the key novel feature of Matryoshka: its ability to solve constraints involving nested conditional statements.

5.3.1 Solved constraints. On each program and given the same seeds, the constraints that Matryoshka can solve is a superset of the constraints that Angora can solve. This is because for each constraint, Matryoshka will first try to solve it using Angora's method. If it fails, then Matryoshka will start to use the methods in Section 3. We evaluated which constraints unsolved by Angora could be solved by Matryoshka. To eliminate the impact of randomness on the fuzzers, we collected the inputs generated by AFL and fed them as the only seeds to both Angora and Matryoshka. In other words, we discarded the new seeds generated by Angora and Matryoshka during fuzzing, respectively. We ran Matryoshka using three different mutation strategies described in Section 3.5 for five hours: *prioritize reachability* (Section 3.5.1), *prioritize satisfiability* (Section 3.5.2), and *joint optimization* (Section 3.5.3).

Table 7 shows the number of constraints that Matryoshka could solve but Angora could not. The table shows that Matryoshka could solve as few as 172 and as many as 1794 new constraints (that were unsolvable by Angora) per program. This demonstrates the effectiveness of the algorithms in Section 3. Table 8 compares Matryoshka's three strategies for solving constraints described in Section 3.5. The strategy prioritizing satisfiabily was the most effective, but there were constraints that this strategy could not solve but others could. The strategy prioritizing reachability was effective on *jhead* and *size*, and the joint optimization strategy was effective on *readpng*.

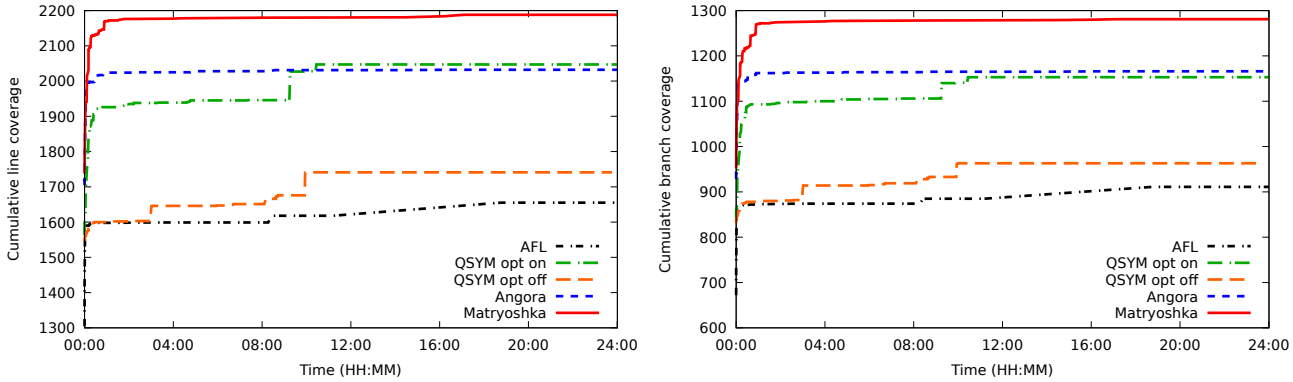
Figure 6 compares the cumulative constraints solved by each individual strategy over five hours on the program *size*. We can see that the strategies prioritize reachability (PR) and prioritize satisfiability (PS) contribute greatly to the the number of constraints solved early on in fuzzing, while joint optimization (JO) solves constraints slowly but continues to grow later on when the other two strategies have reached their respective plateaus.

[5] Matryoshka does not need that much amount of memory. We also successfully fuzzed all the programs on our laptop with only 8 GB memory.

[6] Matryoshka and REDQUEEN also found several unlisted bugs, which the LAVA-M authors injected but were unable to trigger. Table 10 shows the IDs of unlisted bugs.

Table 3: Bugs found on the LAVA-M data set by different fuzzers

Program	Listed bugs	Bugs found by each fuzzer									
		AFL	FUZZER	SES	VUzzer	Steelix	QSYM	NEUZZ	REDQUEEN	Angora	**Matryoshka**
uniq	28	9	7	0	27	7	28	29	29	29	29
base64	44	0	7	9	17	43	44	48	48	48	48
md5sum	57	0	2	0	Fail	28	57	60	57	57	57
who	2136	1	0	18	50	194	1238	1582	2462	1541	2432



(a) Line coverage **(b) Branch coverage**

Figure 5: Cumulative line and branch coverage on *readpng* by AFL, QSYM, Angora, and Matryoshka in 24 hours

Table 4: Programs used in evaluation in Section 5.2

Program	Version	Argument	Size (kB)
djpeg(ijg)	v9c		859
file	commit-6367a7c9b4		781
jhead	3.03		205
mutool(mupdf)	commit-08657851b6	`draw`	39 682
nm(binutils)	commit-388a192d73	`-C`	6659
objdump(binutils)	commit-388a192d73	`-x`	9357
readelf(binutils)	commit-388a192d73	`-a`	2119
readpng(libpng)	commit-0a882b5787		1033
size(binutils)	commit-388a192d73		6597
tcpdump(libpacp)	commit-e9439e9b71	`-nr`	6022
tiff2ps(libtiff)	commit-a0e273fdca		1517
xmllint(libxml2)	commit-d3de757825		6862
xmlwf(expat)	commit-9f5bfc8d0a		785

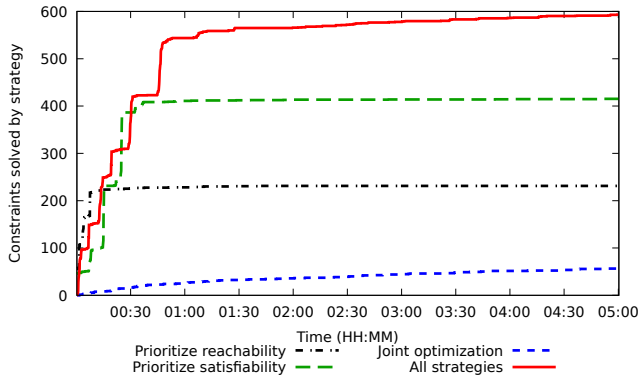


Figure 6: Cumulative constraints solved by Matryoshka's three strategies, respectively, in five hours on the program *size*. *All strategies* means trying prioritizing reachability, prioritizing satisfiability, and joint optimization, in that order until the constraint is solved.

Figure 1 shows an example where Angora could not reach the false branch of Line 6 but Matryoshka could. This is because when Angora mutated `buffer[0]` to satisfy the false branch of Line 6, it caused the CRC check on Line 4 to fail, so the function never reached Line 6. Using the strategy for prioritizing satisfiability, Matryoshka first found an assignment to `buffer[0]`, either 1 or 2, to reach the false branch of Line 6. Then, it backtracked to the prior conditional statement on Line 4. Through byte-level taint analysis, Matryoshka learned that all the input bytes flowed into Line 4, but since `buffer[0]` also flowed into Line 6, this strategy directed Matryoshka to keep `buffer[0]` fixed but to freely mutate all the bytes. Using gradient descent, Matryoshka found an input that satisfied the false branch of Line 4.[7]

[7]The reason why gradient descent helped Matryoshka to find a solution on Line 4 is that the CRC value itself was in the input. Therefore, gradient descent found that the objective function had a constant partial derivative with regard to the input bytes

Table 5: Comparison of coverage between AFL, QSYM, Angora and Matryoshka

Program	Line coverage					Branch coverage				
	AFL	QSYM		Angora	**Matryoshka**	AFL	QSYM		Angora	**Matryoshka**
		opt on	opt off				opt on	opt off		
djpeg	5951	5994	5967	5900	6144	1915	1899	1910	1855	2123
file	2637	3098	2799	3179	3277	1746	2073	1887	2102	2284
jhead	399	761	756	903	948	218	445	440	538	571
mutool	5247	5557	5493	5631	5694	2177	2429	2366	2495	2550
nm	4766	6002	5390	6261	6964	2765	3314	3122	3452	3866
objdump	3904	6380	5678	7906	8076	2291	3190	3090	4263	4297
readelf	7792	8357	7906	10203	11245	5810	5645	5863	7243	7993
readpng	1643	2047	1723	2027	2187	903	1142	956	1161	1278
size	3299	4960	3845	5332	5445	1937	2438	2222	2893	2965
tcpdump	13000	12485	13362	13691	13992	7455	7025	7630	8004	8210
tiff2ps	5193	4892	5054	5303	5291	3217	3048	3109	3325	3304
xmllint	5804	6221	6058	6516	7611	4877	5334	5129	5786	7045
xmlwf	4850	4732	4684	5011	5019	1965	1920	1886	2042	2041

Table 6: Classification of verified bugs found by Matryoshka. SBO: stack buffer overflow; HBO: heap buffer overflow; OOM: out of memory; OBR: out of bound read.

Program	Number of bugs				
	SBO	HBO	OOM	OBR	Total
file	4				4
jhead	2	15		6	23
nm	1	1			2
objdump			3	1	4
size		1	1		2
readelf		4			4
tiff2ps		1	1		2

Table 7: Constraints unsolved by Angora, and nested constraints unsolved by Angora but solved by Matryoshka

Program	Unsolved by Angora		Solved by Matryoshka	% of nested constraints
	All	Nested	Nested	solved
djpeg	1889	1700	345	20.3 %
file	610	527	172	32.6 %
jhead	4923	2853	316	11.1 %
mutool	1883	1523	249	16.3 %
nm	2564	2162	408	18.9 %
objdump	4418	4000	377	9.4 %
readelf	4012	3375	621	18.4 %
readpng	5353	5033	1170	23.2 %
size	4359	3830	593	15.5 %
tcpdump	4343	4079	1794	44.0 %
tiff2ps	8923	6564	330	5.0 %
xmllint	1838	1437	271	18.9 %
xmlwf	5233	5033	301	6.0 %

5.3.2 Effective prior conditional statements. A key insight that allows Matryoshka to solve nested constraints effectively is that it identifies effective prior conditional statements, whose branch choices may cause the target conditional statement to become unreachable, and solves a constraint that consists of only those statements, instead of all the prior conditional statements on the path as done in traditional symbolic execution. Table 9 compares the average number of effective prior conditional statements vs all prior conditional statements. It shows that the effective prior conditional statements account for a very small fraction of all the prior conditional statements (less than 5% on 11 programs, and less than 10% on all the 13 programs). This fact significantly reduces the complexity of the path constraints that Matryoshka solves and increases the likehood that the constraints can be solved.

containing the CRC value, so it directed Matryoshka to mutate those input bytes to reduce the objective function to zero.

6 DISCUSSION

6.1 Comparison with concolic execution

We compare Matryoshka with QSYM while its last branch solving is disabled. This directly compares the effectiveness of Matryoshka's optimization strategies to that of a concolic execution engine. Table 5 shows that Matryoshka performs better than QSYM in all the statistics. This demonstrates that prioritizing reachability, satisfiability, and joint optimization can be used on most path constraints effectively without having to resort to concolic execution.

Table 8: Constraints solved by *prioritizing reachability* (PR, Section 3.5.1), *prioritizing satisfiability* (PS, Section 3.5.2), and *joint optimization* (JO, Section 3.5.3).

Program	Constraints solved by		
	PR	PS	JO
djpeg	1	305	72
file	5	163	11
jhead	172	243	60
mutool	1	247	12
nm	30	321	78
objdump	47	343	53
readelf	2	573	86
readpng	0	1043	313
size	231	414	56
tcpdump	20	1742	59
tiff2ps	10	323	16
xmllint	1	252	31
xmlwf	1	253	97

Table 9: Number of average effective prior conditional statements vs. all prior conditional statements

Program	Average prior conditional statements		
	Effective	All	Effective/all
djpeg	21.69	1217.98	1.8 %
file	22.27	345.25	6.5 %
jhead	16.81	2425.00	0.7 %
mutool	20.08	2087.80	1.0 %
nm	27.93	842.54	3.3 %
objdump	23.93	493.24	4.9 %
readelf	7.23	2498.21	0.3 %
readpng	21.18	859.02	2.5 %
size	21.72	469.46	4.6 %
tcpdump	26.26	268.52	9.8 %
tiff2ps	30.44	1747.16	1.7 %
xmllint	11.80	502.39	2.3 %
xmlwf	5.88	655.31	0.9 %

6.2 Unsolved constraints

6.2.1 Unsatisfiable constraints. Some constraints are unsatisfiable. Figure 7 shows an example in *readpng*. The program calls `png_check_chunk_name` before calling `png_format_buffer`. `png_check_chunk_name` checks if the character is alphanumerical on Line 7. If not, it exits with an error. But later `png_format_buffer` checks the character again on on Line 20, so the false branch of this line is unsatisfiable.

6.2.2 Taint lost in propagation. Section 3.4 uses the results from byte-level taint tracking to determine the taint flow dependency between nested conditional statements. Similar to Angora, Matryoshka also extended DFSan [27] to implement byte-level taint tracking, but neither of the two is able to track taint flows through external libraries. We manually modeled the taint flow in common external libraries for Matryoshka, but this is in no way comprehensive.

```
1  // pngrutilc.c
2  void
3  png_check_chunk_name(png_const_structrp png_ptr,
4              const png_uint_32 chunk_name) {
5    for (i=1; i<=4; ++i) {
6      if (c < 65 || c > 122 ||
7          (c > 90 && c < 97))
8        png_chunk_error(png_ptr,
9          "invalid chunk type");
10     ...
11   }
12   ..
13 }
14 // pngerror.c 445
15 void
16 png_format_buffer(png_const_structrp png_ptr,
17             png_charp buffer,
18             png_const_charp error_message) {
19   ...
20   if (isnonalpha(c) != 0) { ... }
21 }
```

Figure 7: An example with an unsatisfiable constraint. The false branch on Line 20 is unsatisfiable because it is precluded by an earlier check on Line 7.

6.2.3 Program crashing when applying the strategy for prioritizing satisfiability and joint optimization. When mutating the input using the strategy for prioritizing satisfiability (Section 3.5.2) and joint optimization (Section 3.5.3), Matryoshka artificially keeps the branch choices of prior conditional statements. This may cause the program to crash. For example, a conditional statement may serve to prevent the program from accessing data out of bound. If we mutate the length of the data but artificially keep the branch choice of the conditional statement, the program may access data out of bound and crash.

6.2.4 Difficult joint constraints. The joint optimization strategy is the last resort for mutation. We examined the conditional statements that have at least one effective prior conditional statement in the program *tiff2ps* and found that they have on average 30 such prior statements in Table 9. It is difficult to solve such a complex joint constraint.

6.2.5 Constraint dependent on order of branches. On *xmlwf*, Matryoshka and Angora reached similar branches. The unreached branches are guarded by predicates that can only be solved through a specific combination of other branch choices, a situation that none of the fuzzers we tested are designed to handle. These situations are commonly seen in parser logic, where a conditional statement checks the internal state of the parser, while the current state depends on the order of the branches reached.

6.3 Other limitations of Matryoshka

6.3.1 Design limitations. Matryoshka's branch counting method is derived from AFL's, a coarse grained method that can only provide limited information about the program's internal state. This is to maintain compatibility with AFL and AFL-like fuzzers for synchronization, but leads to issues such as those mentioned in Section 6.2.5.

6.3.2 Implementation limitations. The current implementation of Matryoshka requires source code because we use compile-time instrumentation. We could overcome this limitation by instrumenting the executables. Matryoshka's taint tracking uses byte-level granularity as a balance between efficiency and accuracy, as bit-level taint tracking would require significantly more memory and computing power. Section 6.2 described other implementation limitations.

7 RELATED WORK

7.1 Solving complicated constraints

Symbolic execution has the potential to solve complex constraints [8, 10] and is used in fuzzing [18, 19, 11, 26, 4, 34, 28, 37, 41]. One example is Driller, which uses symbolic execution only when the co-running AFL cannot progress due to complicated constrains [34]. Steelix [26] and REDQUEEN [4] detect magic bytes checking and infer their input offsets to solve them without taint analysis. T-Fuzz ignores input checks in the original program and leverages symbolic execution to filter false positives and reproduce true bugs [28]. TaintScopre fixs checksum values in the generated inputs using symbolic execution [37]. In T-Fuzz and TaintScope, input checks and checksum checks are complex constraints. However, symbolic execution faces the challenges of path explosion and scalability [9, 33]. QSYM uses fast concolic execution to overcome the scalability problem, but similar to Angora, it solves only the constraint of the target conditional statement without considering any nesting relationships between other conditional statements [41]. By contrast, Matryoshka finds all those nesting conditional statements and searches for an input that satisfies all of them.

7.2 Using control flow to guide fuzzing

Run-time control flow can contain information useful for guiding fuzzing [30, 28, 37, 13, 25, 6, 12, 40, 16]. VUzzer uses control flow information to prioritize inputs that may explore deep code blocks but that do not lead to error handling codes [30]. Angora prioritizes fuzzing on unexplored branches [13]. AFLGo and Hawkeye measure the distance between the seed input and the target location in the control flow graph, and minimizes the distance in fuzzing [6, 12]. T-Fuzz [28] and TaintScope [37] use control flow features to find sanity checks and checksum checks, respectively. After that, they remove these checks to cover more code.

FairFuzz identifies the "rare branches" exercised by few inputs using control flow information and schedules the fuzzer to generate inputs targeting the "rare branches" [25]. If a path constraint does not exhibit taint flow dependencies on the "rare branches", FairFuzz can solve them efficiently similar to QSYM and Angora. Otherwise, the input bytes flowing into the path constraint will not be included in the mutation mask, e.g. nested conditional statements, and FairFuzz will experience difficulties while solving it.

Post dominator trees [1] were used to determine control flow dependencies [15]. Xin et al.[38] proposed a method to capture both intraprocedural and interprocedural control dependencies efficiently based on post-dominator trees. The method inserts code at the point before each conditional statement(BRANCHING) and the head of its immediate post dominator block(MERGING). Similarly, Matryoshka proposes an equivalent approach without the injections at the MERGING, which is more efficient in our case of finding all the prior conditional statements.

SYMFUZZ [11] uses control dependencies to infer input bit dependencies and use it to find an optimal mutation ratio for fuzzing. Under this method, nested conditional statements will introduce more complex input bit dependencies. SYMFUZZ utilizes this information to reduce mutation ratio for fuzzing, which is incapable of solving nested conditional statements efficiently.

7.3 Using taint tracking to guide fuzzing

Taint tracking can locate which input bytes flow into the security-sensitive code that may trigger bugs [17, 5, 21]. VUzzer [30] is an application-aware fuzzer that uses taint analysis to locate the position of "magic bytes" in input files and assigns these "magic bytes" to fixed positions in the input. VUzzer can find "magic bytes" only when they appear continuously in the input. TIFF [22] is an improvement over VUzzer, using in-memory data structure identification techniques and taint analysis to infer input types. Angora [13] tracks the flow of input bytes into conditional statements and mutates only those bytes. Matryoshka uses the same technique to identify relevant bytes in the input. Taintscope [37] uses taint tracking to infer checksum-handling code and bypasses these checks using control flow alteration since these checks are hard to satisfy when mutating the input. T-Fuzz [28] detects complex checks without taint tracking. Both approaches use symbolic execution to generate valid input that would solve target constraints. Checksum-handling code is an classic example of nesting conditional statements: the code that uses the value is nested under the conditional statement that verifies the checksum. Matryoshka is able to handle such code naturally.

DTA++ allows dynamic taint analysis to avoid under-tainting when implicit flows occur in data transformations[23]. It locates culprit implicit flows that cause the under-tainting through binary search and generates rules to add additional taint for those control dependencies only. However, the method may result in over-tainting. Matryoshka proposes an efficient approach that can avoid over-tainting when determining taint flow dependencies among conditional statements.

7.4 Using machine learning to guide fuzzing

Both Angora and Matryoshka view solving constraints as a search problem and take advantage of commonly used search algorithms in machine learning. Skyfire [36] learns a probabilistic context-sensitive grammar (PCSG) from existing samples and leverages the learned grammar to generate seed inputs. Learn & Fuzz [20] first attempts to use a neural network to automatically generate an input grammar from sample inputs. Instead of learning a grammar, Rajal

et al. use neural networks to learn a function to predict the promising input bytes in a seed file to perform mutations [29]. Konstantin et al. formalizes fuzzing as a reinforcement learning problem using the concept of Markov decision processes and constructs an algorithm based on deep Q-learning that chooses high reward actions given an input seed [7]. NEUZZ [32] uses a surrogate neural network to smoothly approximate a target program's branch behavior and then generates new input by gradient-guided techniques to uncover new branches.

7.5 Fuzzing without valid seed inputs

SLF [39] fuzzes programs without requiring valid inputs. It groups input bytes into fields where a field consists of consecutive bytes that affect the same set of checks. Then, it correlates checks whose predicates are affected by the same field. Finally, it uses a gradient-based method to mutate the fields to satisfy all the correlated checks. At a high level, SLF's approach is comparable to Matryoshka s strategy of prioritizing satisfiability (Section 3.5.2). The differences between Matryoshka and SLF are as follows. First, Matryoshka uses dynamic taint tracking to determine the bytes that flow into a predicate, while SLF uses probing. During probing, the SLF must flip each input byte individually, so if the input has n bytes, then the program must run n times. In contrast, dynamic taint tracking runs the program only once. Second, SLF determines the correlation between two checks based on their common input fields. However, this ignores their control flow flow dependency and may find unnecessary correlations. In contrast, Section 3.3 describes how Matryoshka determines the prior checks that the current check depends on by control flow. Third, SLF classifies some common checks into several categories and applies category-specific strategies effectively. For example, SLF can test offset/count of certain fields. By contrast, Matryoshka needs no prior knowledge of the types of checks and handles all checks uniformly. Finally, besides the strategy of prioritizing satisfiability, which is comparable to SLF's strategy, Matryoshka also provides the strategies of prioritizing reachability and of joint optimization. Table 8 shows that these three strategies are complementary: together they can solve many more constraints than any single one of them can.

8 CONCLUSION

Deeply nested branches present a great challenge to coverage-based fuzzers. We designed and implemented Matryoshka, a tool for fuzzing deeply nested conditional statements. We proposed algorithms for identifying nesting conditional statements that the target branch depends on by control flow and taint flow, and proposed three strategies for mutating the input to solve path constraints. Our evaluation shows that Matryoshka solved more constraints and increased line and branch coverage significantly. Matryoshka found 41 unique new bugs in 13 open source programs and obtained 12 CVEs.

9 ACKNOWLEDGMENT

We thank Dongyu Meng for helpful discussions.

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

This research was partially sponsored by the Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.

REFERENCES

[1] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5 of number 7, pages 1–19, 1970.

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

[3] Angora. URL: https://github.com/AngoraFuzzer/Angora.

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

[5] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security*, 2017.

[7] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. *arXiv:1801.04589*, 2018.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[9] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[10] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012.

[11] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *ACM Conference on Computer and Communications Security*, 2018.

[13] Peng Chen and Hao Chen. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, May 21–23, 2018.

[14] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[15] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization.

ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3):319–349, 1987.

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[17] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering*, 2009.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM SIGPLAN Notices*, volume 40 of number 6, 2005.

[19] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[20] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn & fuzz: machine learning for input fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[21] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *USENIX security*, 2013.

[22] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: using input type inference to improve fuzzing. In *34th Annual Computer Security Applications Conference*, 2018.

[23] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[24] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *CGO*, San Jose, CA, USA, March 2004.

[25] Caroline Lemieux and Koushik Sen. FairFuzz: targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv:1709.07101*, 2017.

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[27] LLVM dataflowsanitizer. URL: https://clang.llvm.org/docs/DataFlowSanitizer.html.

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

[29] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: neural byte sieve for fuzzing. *arXiv:1711.04596*, 2017.

[30] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: application-aware evolutionary fuzzing. In *NDSS*, February 2017.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*, 2012.

[32] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: efficient fuzzing with neural program learning, 2019.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. SOK: (state of) the art of war: offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

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

[35] Technical "whitepaper" for afl-fuzz. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[37] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE symposium on Security and privacy (SP)*, 2010.

[38] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *International symposium on Software testing and analysis*, 2007.

[39] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: fuzzing without valid seed inputs. In *International Conference on Software Engineering (ICSE)*, Montreal, Quebec, Canada, 2019.

[40] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. ProFuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[41] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : a practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, Baltimore, MD, 2018.

[42] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

APPENDIX

Table 10: IDs of bugs injected but unlisted by LAVA, because the LAVA authors were unable to trigger them when preparing the data set. Matryoshka found these bugs.

Program	IDs of bugs unlisted by LAVA-M but found by Matryoshka
uniq	227
base64	274, 521, 526, 527
md5sum	-
who	2, 4, 6, 8, 12, 16, 20, 24, 55, 57, 59, 61, 63, 73, 77, 81, 85, 89, 117, 125, 165, 169, 173, 177, 181, 185, 189, 193, 197, 210, 214, 218, 222, 226, 294, 298, 303, 307, 312, 316, 321, 325, 327, 334, 336, 338, 346, 350, 355, 359, 450, 454, 459, 463, 468, 472, 477, 481, 483, 488, 492, 497, 501, 504, 506, 512, 514, 522, 526, 531, 535, 974, 975, 994, 995, 996, 1007, 1026, 1034, 1038, 1049, 1054, 1071, 1072, 1329, 1334, 1339, 1345, 1350, 1355, 1361, 1377, 1382, 1388, 1393, 1397, 1403, 1408, 1415, 1420, 1429, 1436, 1445, 1450, 1456, 1461, 1718, 1727, 1728, 1735, 1736, 1737, 1738, 1747, 1748, 1755, 1756, 1891, 1892, 1893, 1894, 1903, 1904, 1911, 1912, 1921, 1925, 1935, 1936, 1943, 1944, 1949, 1953, 1993, 1995, 1996, 2000, 2004, 2008, 2012, 2014, 2019, 2023, 2027, 2031, 2034, 2035, 2039, 2043, 2047, 2051, 2055, 2061, 2065, 2069, 2073, 2077, 2079, 2081, 2083, 2085, 2147, 2181, 2189, 2194, 2198, 2219, 2221, 2222, 2223, 2225, 2229, 2231, 2235, 2236, 2240, 2244, 2246, 2247, 2249, 2253, 2255, 2258, 2262, 2266, 2268, 2269, 2271, 2275, 2282, 2286, 2291, 2295, 2302, 2304, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2499, 2500, 2507, 2508, 2521, 2522, 2529, 2681, 2682, 2703, 2704, 2723, 2724, 2742, 2790, 2796, 2804, 2806, 2810, 2814, 2818, 2823, 2827, 2834, 2838, 2843, 2847, 2854, 2856, 2915, 2916, 2917, 2918, 2919, 2920, 2921, 2922, 2974, 2975, 2982, 2983, 2994, 2995, 3002, 3003, 3013, 3021, 3082, 3083, 3099, 3185, 3186, 3187, 3188, 3189, 3190, 3191, 3192, 3198, 3202, 3209, 3213, 3218, 3222, 3232, 3233, 3235, 3237, 3238, 3239, 3242, 3245, 3247, 3249, 3252, 3256, 3257, 3260, 3264, 3265, 3267, 3269, 3389, 3439, 3443, 3464, 3465, 3466, 3467, 3468, 3469, 3470, 3471, 3487, 3488, 3495, 3496, 3509, 3510, 3517, 3518, 3523, 3527, 3545, 3551, 3561, 3939, 4224, 4287, 4295