DSFuzz: Detecting Deep State Bugs with Dependent State Exploration

Yinxi Liu
The Chinese University of Hong Kong
Hong Kong SAR, China
yxliu@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

ABSTRACT

Traditional random mutation-based fuzzers are ineffective at reaching deep program states that require specific input values. Consequently, a large number of deep bugs remain undiscovered. To enhance the effectiveness of input mutation, previous research has utilized taint analysis to identify control-dependent critical bytes and only mutates those bytes. However, existing works do not consider indirect control dependencies, in which the critical bytes for taking one branch can only be set in a basic block that is control dependent on a series of other basic blocks. These critical bytes cannot be identified unless that series of basic blocks are visited in the execution path. Existing approaches would take an unacceptably long time and computation resources to attempt multiple paths before setting these critical bytes. In other words, the search space for identifying the critical bytes cannot be effectively explored by the current mutation strategies.

In this paper, we aim to explore a new input generation strategy for satisfying a series of indirect control dependencies that can lead to deep program states. We present DSFuzz, a directed fuzzing scheme that effectively constructs inputs for exploring particular deep states. DSFuzz focuses on the deep targets reachable by only satisfying a set of indirect control dependencies. By analyzing the conditions that a deep state indirectly depends on, it can generate dependent critical bytes for taking the corresponding branches. It also rules out the control flows that are unlikely to lead to the target state. As a result, it only needs to mutate under a limited search space. DSFuzz significantly outperformed state-of-the-art directed greybox fuzzers in detecting bugs in deep program states: it detected eight new bugs that other tools failed to find.

CCS CONCEPTS

• **Security and privacy** **Software security engineering**.

KEYWORDS

Fuzzing; Program analysis; Software testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11...$15.00
https://doi.org/10.1145/3576915.3616594

ACM Reference Format:
Yinxi Liu and Wei Meng. 2023. DSFuzz: Detecting Deep State Bugs with Dependent State Exploration. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616594

1 INTRODUCTION

Mutation-based fuzzers have been widely adopted in bug detection [16]. People have found that many program states are easy to reach, but some deep states are only accessible through a specific sequence of prior states in the right order. The bugs that only manifest themselves in the deep states are thus difficult to be detected. For instance, network protocols typically have complex internal states, some of which are only reached after receiving several specific types of messages; Rogue-like games move forward only when certain sequences of user inputs are provided. Similarly, data processing programs often contain hard-to-reach deep states. They would typically transit among multiple states (or processing units) depending on the already processed data bytes.

To visit the prior dependent states and the deep state in order, the program inputs must meet very complex and specific constraints. These constraints are hard to satisfy by fuzzers, as a deep state's conditional statement can be directly reached via a preferred path, *e.g.*, a short path, without having to reach its previous dependent states.

The `libpng` library is such an example. A simplified view is presented in Listing 1: it iteratively processes PNG image data chunks using multiple processing units in a `for` loop.

```
1   for (;;) {
2       PNG_CONST png_bytep chunk_name = png_ptr->chunk_name;
3       if (!png_memcmp(chunk_name, png_IHDR, 4))
4           png_ptr->mode |= HAVE_IHDR; // IHDR
5       if (!png_memcmp(chunk_name, png_PLTE, 4))
6           png_ptr->mode |= HAVE_PLTE; // PLTE
7       if (!png_memcmp(chunk_name, png_IDAT, 4)) {
8           if (!(png_ptr->mode & HAVE_IHDR))
9               png_error(png_ptr, "Missing IHDR before IDAT");
10          if (!(png_ptr->mode & HAVE_PLTE))
11              png_error(png_ptr, "Missing PLTE before IDAT");
12          png_ptr->mode |= HAVE_IDAT; // IDAT
13          ... // Bug location
14      }
15      ...
16  }
```

Listing 1: A simplified example showing indirect control dependency across different iterations.

The code to process a chunk in type `IDAT` (lines-12-13) is only executed if the chunk follows `IHDR` chunks and `PLTE` chunks. In other words, the program must have reached the `IHDR` and `PLTE` states corresponding to lines 4 and 6 before it can reach line 12 and

trigger the bug in line 13. One feasible bug-triggering path is `L3-4->L5-6->L7->L8->L10->L12-13`, which involves at least three loop iterations. The bug in state `IDAT` exhibits not only direct control dependency on L7, L8 and L10, but also indirect control dependency on state `IHDR` (L3) and state `PLTE` (L5). The fuzzers must address both types of control dependencies in the path constraint to reach the deep state and trigger the bugs there.

Existing fuzzers can discover and solve the direct control dependencies [31], but it still remains challenging to satisfy the indirect control dependencies for the deep-state bugs. Coverage-guided fuzzers can easily generate several inputs that separately satisfy the conditions in line 3, 5 and 7, but can hardly generate one input that satisfies all three in a specific order. Directed fuzzers are none the wiser. Even by setting line 13 as the target, they would usually prefer the shortest path that reaches state `IDAT`, rather than a longer path that first reaches `IHDR` and `PLTE`, and then enters `IDAT` via at least three loop iterations [10, 34].

One approach to solving the complex path constraints is performing dynamic taint analysis to identify the constraints' dependent input bytes and mutate them [8, 12, 18, 19, 26, 28, 38, 42]. However, using taint analysis alone cannot adequately address the issue of indirect control dependencies and order state transitions for three reasons. First, the dependent input bytes for indirect control dependencies might not be efficiently discovered. Dynamic taint analysis covers mostly the executed code. The indirect control dependent code for the deep targets might not be visited by the fuzzers, *e.g.*, they are on paths far away from the targets. In such cases, taint analysis would fail as it cannot deduce all the critical dependent input bytes for satisfying these constraints.

Second, it is challenging to precisely infer the control dependent input bytes for each conditional statement on the path. For instance, a local variable (*e.g.*, `chunk_name` in Listing 1) could take values from different data chucks (that are located at different positions in the input) and be used in multiple conditional statements (*e.g.*, L3, L5 and L7) on one path. For such cases, existing approaches either fail to distinguish these different data chunks [12, 38, 42], or need to mutate all the possible value combinations of these data chunks [8, 17, 19, 26–28, 46].

Third, the dependencies need to be satisfied by mutating different critical input bytes in the correct order. The existing fuzzers are unaware of the correct order of state transitions for reaching the deep location. As a result, they must examine numerous combinations of conditional branches to uncover the correct execution path, which could be time-consuming and sometimes infeasible under limited resources. This problem gets magnified for programs that process long inputs consisting of a large number of influencing data chunks. While only a few combinations of these data chunks can reach the deep state, existing approaches can only try all possible combinations one by one without inferring the expected order. Besides, there may be a large number of indirect control dependencies that need to be satisfied for deep states to be reached in the real world (Listing 4). Consequently, existing approaches are unable to efficiently reach the deep states.

In this paper, we aim to solve the challenges in detecting real-world bugs hidden in deep states reachable by only satisfying their indirect control dependencies. We observe that reaching such a deep state requires first reaching the prior dependent states in the correct order, which is hard for conventional fuzzers. We propose DSFuzz, a directed fuzzer that smartly explores the dependent program states to reach deep states and detect new bugs. It is equipped with two new techniques that we developed —a static state dependency analyzer and a progressive micro directed fuzzing scheme.

Specifically, we first identify dependent state transitions and build a state dependency graph (SDG), then identify deep states and set them as targets. We build SDG by first constructing the edges representing direct control dependencies from a def-use analysis, then storing the data flow information from a static inter-procedural data flow analysis, and finally identifying indirect control dependencies using the data flow information. For each state on the graph, we determine whether it is deep by computing the chance of satisfying all its indirect control dependencies.

We then start directed fuzzing toward these deep targets gradually. The fuzzer must follow the required state transitions on the SDG to reach the target deep state. The task can be divided into multiple phases, where in each phase DSFuzz solves a *micro directed fuzzing task*. In each phase DSFuzz progressively approaches the target state by making one transition from the currently reached state to the next dependent state on the state transition path. To make a state transition, it sets the next dependent state as the target and then performs micro directed fuzzing. In order to efficiently arrive at the previously reached states, it identifies their dependent input bytes and preserves their values. This is achieved by conducting a taint analysis on the conditions that enable each state transition, followed by analyzing the input ranges that influence the value of these conditions. By only conducting taint analysis on the conditional statements of dependent state transitions, DSFuzz avoids wasting energy on running taint analysis that cannot deduce dependent input bytes. It then mutates only the input bytes influencing the current state transition to solve the micro directed fuzzing task. Such a divide-and-conquer strategy allows DSFuzz to solve the challenge of satisfying complex indirect control dependencies.

We extensively evaluated DSFuzz with two widely used datasets—Fuzzbench [33] and Magma [21]. DSFuzz significantly outperformed two state-of-the-art directed greybox fuzzers—AFLGo [10] and ParmeSan [34]—in a variety of programs. It detected all bugs that can be detected by the two fuzzers combined, and was the most efficient tool for almost all bugs. It outperforms the two fuzzers in detecting not only the deep state bugs we identified, but also non deep state bugs in Magma. This is because our solution limits the fuzzer's search space by only targeting the dependent state transitions. It also detected eight previously unknown bugs that other tools failed to detect from 47 potentially buggy deep locations we identified, including an almost two-decade-old one. We responsibly disclosed the newly detected bugs to the relevant developers. With a component-wise evaluation, we demonstrate that our state dependency analysis and progressive micro directed fuzzing technique are both needed for detecting deep state bugs.

We made the following contributions.

- We uncover that deep program states having indirect control dependencies exist widely but are difficult to reach with state-of-the-art techniques.
- We propose to satisfy indirect control dependencies by identifying the implicitly required dependent states and reaching them progressively.

- We integrate the above analyses into DSFuzz, a directed fuzzing scheme for efficiently reaching deep states.
- We demonstrate the capabilities of DSFuzz—it outperformed two state-of-the-art tools in two widely used benchmarks, and detected eight new bugs that other tools failed to find.

2 BACKGROUND AND PROBLEM STATEMENT

In this section, we discuss existing approaches and their limitations (§2.1), and define our research problem, goal, and scope (§2.2).

2.1 Existing Approaches

Researchers have proposed various methods to reach deep states and detect deep bugs, but no existing approaches are specialized in satisfying indirect control dependencies.

Directed Fuzzing. A straightforward solution is to set the deep state locations as the targets and apply directed fuzzing [10, 34]. As discussed above, however, state-of-the-art directed fuzzers are discouraged from generating the necessary dependent data blocks/bytes that help reach prior dependent states (*e.g.*, `IHDR` and `PLTE`) because such inputs could be more distant from the targets compared to other states (*e.g.*, `IDAT`).

Structured Input Generation. People also try to generate structured inputs to pass the complex validity checks that block mutation-based fuzzers from further testing. For instance, the data block of `IHDR` might require a different length than the data block `PLTE`; one cannot simply mutate one to another by changing the chunk name bytes. Existing work mitigates this problem by modeling the input format [9, 37] or the range of data chunks [18]. However, these approaches can only ensure the generated data blocks are valid. The data blocks are not ordered in a way that could lead to a series of state transitions toward the deep states.

Critical Byte Mutation. Researchers have tried to solve the complex constraints to reach the deep states. A common solution is to identify the critical bytes that affect the control flow and further mutate these bytes [8, 12, 38]. However, this approach cannot efficiently discover the dependent input bytes for indirect control dependencies. The analysis can only cover executed code, while the indirect control dependent code for the deep state targets might not be visited. More complex solutions either only support minimal environment/instruction sets [14, 23, 35] or need to use symbolic execution [36, 38, 41, 44]. There is already evidence to suggest that setting up and maintaining a symbolic environment is not worth the effort [5].

Control Dependent State Identification. Researchers have found that the deep states must be reached via certain dependent states. Some works identify representative inputs that reach these dependent states to provide the necessary execution context for reaching deep states [8, 17, 26, 27]. However, such approaches do not consider the different occurrence sequences of these dependent states, and therefore cannot handle loops over them.

Some works propose to record all the occurrence sequences of the dependent state [19, 28, 46]. They would typically consider the variable `chunk_name` as the key variable and mutate different occurrence sequences of its value. Such a strategy would bring a large search space as most of the sequences do not trigger the bug. Mutating over all possible sequences is time-consuming and sometimes infeasible under the limited fuzzing resources. Currently, no existing solution is aware of the correct order of dependent state transitions for reaching the deep location.

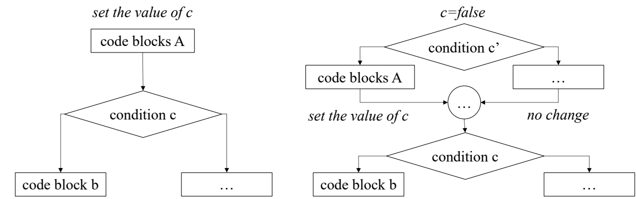

Figure 1: Direct Dependency

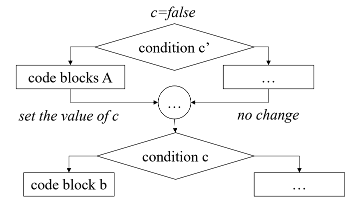

Figure 2: Indirect Dependency

2.2 Problem Statement

This paper aims to develop a new fuzzing technique to detect bugs in deep program states. Each branch (and the corresponding basic block) in a program is directly control dependent on its corresponding branching condition. The condition must be true for the program to execute that branch and reach the corresponding state. Variables used in that condition are referred to as key variables. The same branch (basic block) can be executed in different paths, each of its occurrences uniquely describes a program state that enables the transition to another state.

We abstract the process of satisfying condition c of block b into two steps. First, the program executes a set of code blocks A that set the value of the variables used in c. Second, the value of c is evaluated to determine whether the code block b will be executed. It is generally the case that the same set of conditions influence the control flow of code blocks in A and condition c (Figure 1). Code block b is directly control dependent on condition c, as all execution paths that reach condition c can reach the code blocks in A. However, A and c could exist in two different branches such that they may not be reached at the same time (Figure 2). Likewise, code blocks in A could be control dependent on another condition c', where a path that reaches condition c does not necessarily satisfy c'. Code block b is indirectly control dependent on condition c', as reaching code block b must first satisfy c' before c.

This paper examines indirect control dependency among program states, where one state's control flow is indirectly influenced by the values set in prior states. Reaching such a state requires first reaching the prior dependent states. For example, the iteration reaching the bug location in `IDAT` requires prior iterations that reach `IHDR` and `PLTE`. We point out that it is challenging for state-of-the-art fuzzers to satisfy such dependencies, as they do not intentionally differentiate the dependent prior states from the other possible states along exploration.

We consider a program state deep if it is only accessible through a specific series of indirect control dependencies, because the traditional directed fuzzers are good at satisfying the direct control dependencies. They are unlikely to reach a deep state when there are many hard-to-satisfy indirect control dependencies, as the majority of their fuzzing energy would be wasted on mutations that do not satisfy these dependencies. As a result, guiding the program to reach a deep state is challenging without identifying and satisfying all the prior states it needs to transit through first.

We define the deep state as a state where the chance P of passing all indirect control dependencies is lower than a boundary k, *i.e.*, $P < k$. Even though there are other definitions of deep states, the ones we focus on are both prevalent and previously ignored. Programs that contain lots of indirect control dependencies would benefit the most from our technique. For example, applications that process streaming data blocks typically follow different control flows based on the content of the already processed blocks, creating a chain of dependent data block processing states. Besides, applications with a large number of user-specified settings contain many dependent states. Effectively exploring these states is challenging, especially when the number of options for each setting is large.

We aim to explore previously hard-to-reach deep bugs by identifying not only direct but also indirect control dependencies between program states. We also aim to develop an efficient method to search for satisfying inputs for these deep bugs. Our solution is not intended as a replacement for existing general fuzzers, but as a complement. For instance, users can use our static analysis to check whether their target programs and code locations contain many indirect control dependencies. Another tool can then be used if the target programs and code locations do not contain any indirect control dependencies.

3 DESIGN

We first give an overview of our solution using the same motivating example (§3.1), then present the detailed component-wise design of DSFuzz following the workflow in Figure 3. DSFuzz builds the state dependency graph of a program and analyzes the indirect control dependencies in the graph to identify the target deep states (§3.2). Based on this information, we can perform directed fuzzing on these targets (§3.3). We explore the possible state transition paths to approach the targets (§3.4), and employ a micro directed fuzzing campaign for transiting to each individual dependent state on the paths (§3.5).

3.1 Overview

We design DSFuzz to explore deep program states efficiently. Transitions to these states are control dependent on some critical variables set in prior states. We have shown in the motivating example (Listing 1) that the exact state transition (*i.e.*, `IHDR, PLTE, IDAT`) is critical to uncover the bug, and such transitions involve some indirect control dependencies. In particular, `png_ptr->mode` must first be set at line 4 and 6 for passing the checks in line 8 and 10. This indicates that state `IHDR` and state `PLET` could be the prior states of state `IDAT`, as the transition to state `IDAT` is dependent on the critical variable `png_ptr->mode` set in state `IHDR` and state `PLET`.

DSFuzz can automatically identify such state transitions and build a state dependency graph (SDG). It first conducts def-use analysis to construct the initial SDG, and incrementally adds indirect call edges with dynamic taint analysis. Based on the SDG, DSFuzz can identify deep states and set them as targets (*e.g.*, the target in line 13). Such deep states are indirectly control dependent on some prior states: their control flows (*e.g.*, line 8 and 10) are influenced by values set in prior states (*e.g.*, line 4 and 6).

DSFuzz then performs directed fuzzing to efficiently explore a (short) transition path on the SDG to reach the deep states. It would first identify a sequence of dependent states for reaching a deep state. For example, to reach line 13, it identifies two dependent states `IHDR` and `PLET` that are in line 4 and 6, respectively. It then performs micro directed fuzzing to gradually make a sequence of transitions to reach these dependent states and the final target state, *e.g.*, transitions from the entry state to `IHDR` and `PLET`, and then to the target state `IDAT`. In each micro directed fuzzing task, DSFuzz tries to make a state transition from the current state S to the next state T. To identify and mutate the dependent input bytes for making the state transition, DSFuzz first runs dynamic taint analysis to obtain all the input bytes that have data flow to the dependent condition of state T. It then eliminates the input bytes that affect state S and precisely locates the ones that only affect state T.

3.2 State Dependency Analysis

DSFuzz builds a state dependency graph (SDG) to analyze the conditions of transitions among the states and find the deep states that have indirect control dependencies on other prior states. These deep states and their dependent prior states are set as the targets of our directed fuzzer, which we will describe in more detail in §3.3. We first present how we construct an SDG (§3.2.1), then show how we identify deep states based on its indirect control dependencies (§3.2.2).

3.2.1 State Dependency Graph Constrution. An SDG consists of nodes representing basic blocks and their corresponding states, and two types of directed edges between nodes representing either direct or indirect control dependencies between the states. We use solid directed edges to represent direct control dependencies and dashed directed edges to represent indirect control dependencies. Use Figure 4 as an example: node `%163 Bug Loc` is directly control dependent (as represented by a solid directed edge) on `%71 png_ptr-> mode`, which is indirectly control dependency (as represented by a dashed directed edge) on `Store %71`. We also use undirected edges to link nodes whose representing basic blocks are adjacent in the program's bitcode, *e.g.*, `Store %71` is located right after %105 IHDR in the bitcode.

As the first step of constructing an SDG, we leverage existing def-use analysis to construct the edges representing direct control dependencies and store the data flow information. This information will later be used for analyzing the indirect control dependencies. Algorithm 1 describes the steps we take for analyzing direct/indirect control dependencies and building the SDG.

Direct Control Dependency. We collect direct control dependencies from a def-use analysis and collect data flow information from a static inter-procedural data flow analysis. We dynamically update the SDG by adding the indirect call sites (§4.1). We use two data structures to store the data flow information of an SDG: one at the variable level and the other at the block level. The variable level one is *DFI*—a map that stores pairs of definition and write dependencies between LLVM *Values* (*i.e.*, the base class of all values computed in the bitcode). The block level one is SDG_b—another map that stores all the variable level data flow information corresponding to a basic block b. We differentiate different states of the same block by their execution context, *i.e.*, the blocks that have already been executed.

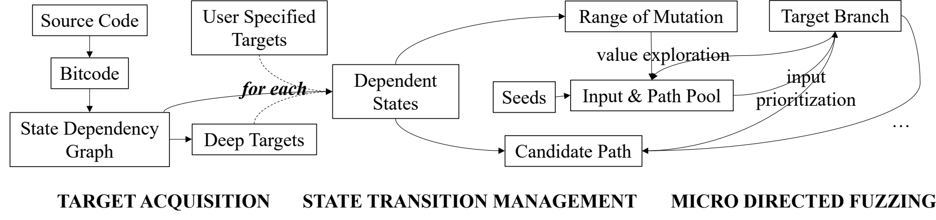

Figure 3: An architecture overview of DSFuzz.

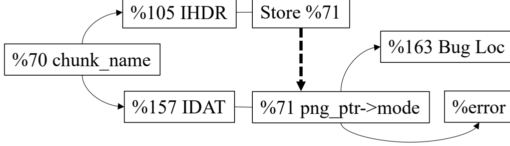

Figure 4: An Example of Identifying Indirect Control Dependency.

Algorithm 1 State Dependency Graph builder.

Input: $function$: The entry function of the analysis.
Output: SDG_b: Basic block level SDG, DFI: Data flow information.

1: Initialize $DFI \leftarrow \{\}$
2: **for all** $block \in collectBasicBlocks(function)$ **do**
3: Initialize $SDG_{block} \leftarrow \{\}$
4: **for all** $instruction \in collectInstructions(block)$ **do**
5: **if** $isDefInst(instruction)$ **then**
6: Initialize $DFI_{getVal(instruction)} \leftarrow \{\}$
7: **end if**
8: **if** $isGenInst(instruction)$ **then**
9: $insert\ getSource(instruction)\ into\ DFI_{getRes(instruction)}$
10: **end if**
11: **if** $isCall(instruction)$ **then**
12: $argDep \leftarrow SDGBuilder(getCallee(instruction), SDG_b, DFI)$
13: $updateDataDep(argDep, DFI)$
14: **end if**
15: **for all** $u \in getUses(instruction)$ **do**
16: $insert\ DFI_{getDef(DFI,u)}\ into\ SDG_{block}$
17: **end for**
18: **end for**
19: **end for**
20: **for all** $v \in getVariables(SDG)$ **do**
21: **for all** $w \in getDependentConditionals(SDG, v)$ **do**
22: **if** w is reachable without v **then**
23: Update w indirectly dependent on v
24: **else**
25: Update w directly dependent on v
26: **end if**
27: **end for**
28: **end for**

We perform static data flow analysis on all instructions in the order of program execution. The analysis operates differently depending on the type of each instruction. If the instruction is a definition, we initialize a corresponding set of uses in *DFI* (line 6). If the instruction is a generative one that creates a new LLVM value (§4.2), we record the direct control dependency between the instruction source and the result (line 9). In the case of a call instruction, we invoke a subprocess running Algorithm 1 for its target function, which would perform a similar static data flow analysis and provide information about the dependencies between function arguments (line 12). We can then store the argument dependencies into *DFI* (line 13). After that, we can connect data flow information with *SDG* nodes (line 16). To represent all data flows to a node, we store the *DFI* of the definitions of all uses in the instructions contained within the node. The stored data flow information will be later used to build the edge information in *SDG* (line 25).

Indirect Control Dependency. We analyze the SDG data flow information to identify indirect control dependencies. According to our definition in §2.2, a target state is indirectly control-dependent on another under two criteria. First, the indirectly control-dependent state sets the value of a variable, which is later used in a conditional for transiting into the target state. Second, a path reaching the target state does not necessarily reach the dependent state first; otherwise, they are direct control dependent.

To efficiently locate potential dependent states, we first search for the patterns where a data write operation flows to a control block. We iterate over the write statements in SDG (line 20), and extract the target control blocks in data flows (line 21). Nonetheless, the write statement does not necessarily enables a path to the target. The value set by the write statement could, for instance, prevent the control block from taking the branch to the target. We implement a more precise data flow analysis (§4.2) to rule out certain situations where the write statement cannot satisfy the expected branch condition.

After locating the data flow pattern, we check if the destination control block w is reachable via a path that does not visit the variable write statement v (line 22). Figure 4 gives an example of how we analyze the bitcode in Listing 2 to find the indirect control dependency between `%105 IHDR` and `%163 Bug Loc`.

```
1  // %70: chunk_name, %71: png_ptr->mode
2  %102 = tail call i32 @bcmp(i8* %70, i8* %IHDR)
3  br label %104
4  %104 = icmp eq i32 %102, 0
5  br i1 %104, label %105, label %109
6  // IHDR
7  %106 = load i64, i64* %71
8  %107 = or i64 %106, HAVE_IHDR
9  store i64 %107, i64* %71 // Store into %71
10 ...
11 %154 = tail call i32 @bcmp(i8* %70, i8* %IDAT)
12 br label %156
13 %156 = icmp eq i32 %154, 0
14 br i1 %156, label %157, label %exit
15 // IDAT
16 %158 = load i64, i64* %71 // Load from %71
17 %159 = and i64 %158, HAVE_IHDR
18 %160 = icmp eq i64 %159, 0
19 br i1 %160, label %png_error, label %163
20 ...
```

Listing 2: A simplified example of indirect control dependency.

By identifying the variable write statement with `Store`, we can get `%71 png_ptr->mode`, which flows to the control block `%160` in line 19. Since the `Store` instruction is under the control block `%105 IHDR`, and `%163 Bug Loc` depends on the control block `%160`, there exists a path from `%105 IHDR` to `%163 Bug Loc` that does not include the write statement. Besides, `%105 IHDR` and `%163 Bug Loc` are

control dependent on the same conditional that evaluates `%70 chunk_name`, which is executed once in each loop iteration. Getting to `%163 Bug Loc` requires a prior iteration that gets to `%105 IHDR`.

3.2.2 Deep States and Deep Targets. We determine whether a given target state b is deep based on the chance of satisfying all its indirect control dependencies (§2.2). We use $c_i \in C$ to denote all the indirect control dependencies between b and its dependent conditions. The probability that a dependent condition c_i is satisfied is p_i, which is approximated as the number of satisfying branches n_s^i divided by the number of all possible branches n^i, *i.e.*, $p_i = \frac{n_s^i}{n^i}$. We discuss the implementation details for calculating n^i in §4.1. The chance of satisfying all indirect control dependencies is further calculated by multiplying the chances of satisfying individual ones, *i.e.*, $P = \prod p_i$.

Our computation uses two approximations. Firstly, we assume each branch has the same chance of being satisfied on average; and secondly, we assume the satisfactions of individual dependencies are independent. While these assumptions may not always hold in practice, this simple metric is already helpful in estimating the difficulty of satisfying indirect control dependencies. A better computation could potentially lead to a more precise estimation, which we leave as future work.

We further refine the target locations by using the sanitizer-instrumented code locations inside the deep states we identify. Those locations are already marked as potentially buggy by the sanitizer, making it easier for the fuzzer to find potential bugs.

For example, `Bug Loc` is marked as potentially buggy by a sanitizer and has indirect control dependencies on `IHDR` and `PLTE`. Suppose the conditional for reaching `IHDR` and `PLTE` contains N branches, the chance of meeting both dependencies is $P = \frac{1}{N^2}$. As it turns out, $N = 49$, which suggests that 99.96% of the fuzzing trials would not reach the target. Therefore, state `Bug Loc` is considered a deep target.

3.3 Fuzzing Loop

After collecting the deep states, DSFuzz tries to reach them through directed fuzzing. The fuzzing loop of DSFuzz is different from a traditional directed fuzzer, in that it progressively reaches a target by reaching each of its prior dependent states.

In particular, DSFuzz divides the task into multiple phases, and solves a *micro directed fuzzing task* in each phase. It progressively approaches the target state by making one transition from the currently reached state to the next dependent state on the state transition path (§3.4). To make a state transition, it sets the next dependent state as the target and then performs micro directed fuzzing. The micro directed fuzzing operates similarly to a traditional direct fuzzer. It mutates specific input bytes to satisfy a series of path constraints to reach a given code location (§3.5).

Algorithm 2 illustrates the entire process. The algorithm accepts as input a set of input seeds I, a set of targets T, two versions of target programs (including one instrumented version $program_t$ for taint tracking and one without $program_{nt}$ for efficiency), and a metric d for computing the distance from a branch condition to a target. The metric d is similar to but different from the one in ParmeSan [34]. While ParmeSan calculates distances between conditionals, DSFuzz calculates distances between SDG states. Using

Algorithm 2 DSFuzz's fuzzing loop.

Input: I: seed inputs, T: the set of target conditions (optional), $program_t$: program with taint instrumentation, $program_{nt}$: program without taint instrumentation, d: distance metric.
Output: I_X: crashing inputs.

```
1:  conds ← initializeSeed(program_t, I) // Unexplored branch conditions and
    the corresponding input that uncovered that condition.
2:  for all target ∈ T do
3:    S ← satisfyIndirect(target) // Dependent states required by the target.
4:    while p ← getNextPath(S) is valid do
5:      range ← initializeRange(p) // No further mutations are made to these
        ranges of bytes.
6:      queue ← initializeQueue(p, d) // This priority queue stores data for
        conditions, with each item containing three fields: condition, distance, and
        serial number.
7:      Sort queue by distance and serial number.
8:      while queue ≠ Φ do
9:        c ← pop(queue)
          // Micro directed fuzzing:
10:       i ← inputSelect(conds, c)
11:       while c is still unexplored and !timeout do
12:         i' ← satisfyDirect(c, i, range)
13:         if program_nt(i') crashes then
14:           insert(i', I_X)
15:         end if
16:         if isInteresting(i') then
17:           path' ← program_t(i')
18:           for all unexplored branch c' on path' do
19:             conds[c'] ← i'
20:           end for
21:         end if
22:         if c was explored then
23:           conds ← conds − {c}
24:           range ← getRange(c, i')
25:         end if
26:       end while
27:       if timeout then
28:         break // Failed due to timeout. Please try another path.
29:       else
30:         pass // Successfully reached one state, proceed to the next.
31:       end if
32:     end while
33:     if queue == Φ then
34:       break // Successfully found a path to the target
35:     end if
36:   end while
37: end for
```

d would result in a lower distance for the deep states compared to the metric in ParmeSan. Such a lower distance would encourage DSFuzz to reach the deep states rather than the shadow states. The state transition management phase accepts target locations with their indirect control dependencies. Analyzing indirect control dependencies produces a series of dependent states for exploration (line 3). In order to reach the target location, it progressively explores a potentially feasible transition path through these dependent states (line 4). As an initial step in exploring each candidate transition path, DSFuzz identifies the dependent input bytes of the previously reached states in the path and avoids further mutation of those bytes (line 5). It then produces a priority queue that contains information about the conditions that must be satisfied (line 6). Each item in the priority queue is a tuple that contains three elements, namely the branch condition to flip for reaching a dependent state, the distance information calculated from the state dependency graph, and a serial number based on the number of times the corresponding dependent state appears in the candidate path. The serial number is set to one when the corresponding state occurs only once on the candidate transition path. It is increased by one for each new occurrence of the state. The items are ordered

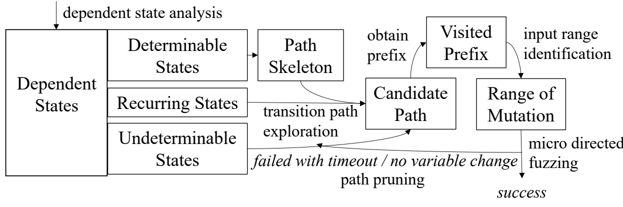

Figure 5: The Procedural Diagram of State Transition Management.

by 1) the state distance (line 7), 2) the position of the condition on the path, and 3) the serial number. DSFuzz pops the next condition from the priority queue to satisfy it and make a transition to the next state (line 9). To satisfy the condition, DSFuzz starts a micro-fuzzing campaign (line 10-26). It first runs input prioritization to determine the input to mutate from (line 10). When the condition is unexplored, DSFuzz mutates input bytes to satisfy it (line 12). The goal of mutation is to solve the constraints posed by the unexplored branches. DSFuzz flips existing bytes into the desired value that solves these constraints. After satisfying the constraints, DSFuzz tries to trigger bugs and collect crashes (lines 13-14). It stores all crushing inputs into $I_{\boldsymbol{X}}$, which is later produced as the output. When a new branch is explored, it would also add new inputs to the seed set (lines 17-20). As a last step, DSFuzz removes newly explored branches from the unexplored branch set, and updates the input ranges that do not require further mutations (lines 23-24).

When a micro-fuzzing campaign succeeds within a timeout limit (line 22), DSFuzz moves to the next state (line 30) or report success when there is no remaining state in the queue to explore (line 34). When it fails due to timeout (line 28), DSFuzz breaks the current path exploration campaign and selects a new path to explore.

3.4 State Transition Management

In this subsection, we explore the necessary state transition paths. These paths allow the fuzzer to drive the program to transit through the correct sequence of dependent states and progressively approach the target state. Figure 5 presents the detailed procedure.

As the first step, DSFuzz identifies the necessary dependent states of each target through dependent state analysis (§3.4.1). These dependent states can be connected via transitions to form potentially feasible transition paths to the target. However, due to the limitation of the static inter-procedural analysis, we cannot predetermine the number of occurrences of these dependent states or their order. Therefore, given the same set of dependent states, there could exist a huge number of possible transition paths, *e.g.*, some states need to be visited multiple times.

We overcome this huge complexity with three strategies (§3.4.2). First, we perform transition path exploration using a breadth-first search approach with a preference for shorter paths. Second, after exhausting all shorter paths, if we need to explore a longer path consisting of l states, we retrieve its visited prefix to reuse the known inputs and efficiently reach the initial $l-1$ visited states. Third, we conduct path pruning if DSFuzz fails to explore any of its containing states within a timeout. Paths with pruned prefixes will not be explored, resulting in a significant reduction of the search space in many cases.

DSFuzz proceeds with each candidate transition path by making a transition to the next state through micro directed fuzzing. Before each micro directed fuzzing campaign, DSFuzz utilizes precise dynamic taint tracking to examine the input ranges of the dependent states already visited, and employs these ranges to direct subsequent mutations (§3.4.3). DSFuzz finally mutates the corresponding input bytes used in the control code blocks of the next state to effectuate the transition.

3.4.1 *Dependent State Analysis.*

We aim to efficiently build a potentially feasible transition path by analyzing how candidate dependent states (obtained by §3.2) can be connected. We analyze whether a state should be included on the path, whether a state should be repetitively visited, and whether different state visiting orders result in different execution contexts.

Determinable States. DSFuzz first focuses on states where it is possible to determine the order in which they are visited. It computes the order based on the position of their parent conditionals in the program's control flow graph. States that do not affect the target or any of its predecessor states are considered irrelevant states. They should not occur in the transition path and waste DSFuzz's energy on exploring them. The input bytes of irrelevant states are removed to save the analysis time and reduce interference. DSFuzz would also randomly determine the order for dependent states whose order does not affect the value of key variables. This is the case when the states modify different key variables, or when each state increases or decreases the key variable by a fixed amount. It would not be worth the energy to explore other possible orders of these dependent states. In our motivating example, prior states apart from `IHDR` and `PLTE` would not affect the control flow to the target bug location. Furthermore, the resulting value of the key variable `%71 png_ptr-> mode` is not affected by the order in which these two states are visited. We let DSFuzz avoid visiting any other irrelevant states or altering the order for reaching `IHDR` and `PLTE` on the path. This allows us to minimize the number of transitions needed for reaching the target.

Recurring Dependent States. Some states need to be visited multiple times to set the correct condition values for making transitions to later states. In other words, the value of key variables K could be affected by the number of occurrences of a dependent state s_K, rather than being directly affected by some input bytes. We identify such states by analyzing how their individual occurrence affects K. Each indirect control dependency corresponds to a variable write in the source code associated with a `Store` instruction (*e.g.*, `png_ptr->mode |= HAVE_IHDR`). If the write statement directly sets the key variable to a particular value that depends not on other states, recurring such states produce the same result. We only allow these states to exist once on the transition path. However, if the key variable values are computed from their past values (*e.g.*, applying arithmetic operations to the original value), different rounds of state visits would produce different results. These dependent states can be visited multiple times for producing different variable values.

Undeterminable States. In many cases, the order of the dependent states cannot be pre-determined, *i.e.*, when there exists data flow between multiple dependent states S_K, the order in which these states are visited can impact the values of the variables in K. Under this circumstance, dependent states S_K must be visited in a specific order to satisfy the conditions using K. Therefore, we would

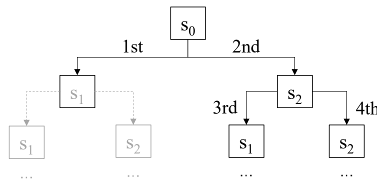

Figure 6: An Example of Transition Path Exploration.

experiment with various visit sequences for these interdependent states to locate the particular order needed.

In certain instances, the dependent state visiting sequence does not influence the values of variables within set K. Since the visiting order can be arbitrary, we pick just a random order to avoid unnecessary exploration.

3.4.2 Transition Path Exploration. DSFuzz progressively explores a feasible transition path to the target deep state by walking through the identified dependent states, *e.g.*, a transition path to the `Bug Loc` through `IHDR` and `PLTE`. DSFuzz can pre-determine the order of visiting determinable states, but it cannot do so for recurring dependent states and undeterminable states. This can result in a large number of feasible transition paths.

DSFuzz explores these feasible transition paths in a breadth-first search manner following the order of their length. This is because a shorter path typically requires solving simpler path constraints and therefore is potentially easier to explore. In addition to employing breadth-first search, DSFuzz would prune certain paths to further reduce the search space. DSFuzz prunes a path if the exploration of any of its dependent states fails within a timeout, or if DSFuzz has already explored a shorter path that yields the same K values. If a path has a pruned prefix, it will not be explored, either. This strategy significantly reduces the search space in many cases.

After selecting one candidate path, DSFuzz tries to progressively explore the path by invoking micro directed fuzzing. Each micro directed fuzzing campaign mutates inputs to satisfy certain conditions and lets the program transit from a prior state to the next dependent state. Whenever DSFuzz fails to satisfy a condition (§3.5.2) within a timeout, it breaks the current micro fuzzing campaign and alters to a different transition path. It continues by searching for a previously explored path that shares the longest common prefix of dependent states. It then restores the execution context from the last dependent state in the common prefix. Due to the breadth-first nature of the exploration process, a shorter path with the same prefix must have already been explored before exploring a longer path. Specifically, when all shorter paths have been exhausted, and a longer path consisting of l states needs to be explored, we can optimize the process by reusing the previously known inputs to efficiently reach the first $l-1$ visited states.

Figure 6 is an example to demonstrate how the breadth-first exploration and pruning operates. Assuming that the number of occurrences and the order of the two dependent states s_1 and s_2 are undetermined. DSFuzz initially attempts to explore the shortest path that includes both dependent states s_1 and s_2, namely $\{s_1, s_2\}$, starting from the initial state s_0. In this attempt, DSFuzz would first try to explore the prefix sequence of the path, namely $\{s_1\}$. After failing to reach s_1 during a single micro fuzzing campaign, DSFuzz prunes all paths that begin with s_1 and switches to a new path that starts with s_2, namely $\{s_2, s_1\}$. On the second attempt using two micro fuzzing campaigns, DSFuzz is able to progressively reach both s_2 and s_1, but is ultimately unable to reach the target state. Afterward, DSFuzz attempts to explore other short transition paths that start with $\{s_2, s_2\}$.

3.4.3 Input Range Identification. DSFuzz performs dynamic taint analysis to identify the dependent input bytes for every state transition. This is achieved by analyzing the key control variables employed in the conditional statements corresponding to each transition. One challenge that we must address is that the same key variables can take different values from different input bytes in different states (execution contexts). Performing taint analysis on these key variables without taking into account the execution context would result in a significant number of input bytes being affected, whose relationships with different states are not straightforward.

To solve this challenge, we propose a context sensitive dependent input bytes identification method. We identify and mark the positions (*i.e.*, offset ranges) of the dependent input bytes for each state transition in the dynamic taint tracking phase. Only the corresponding input bytes are mutated in one micro directed fuzzing task for solving a state transition. DSFuzz keeps track of the concrete values of dependent input bytes for successful state transitions, and replays them to quickly reproduce the past states. For example, when DSFuzz tries to reach `PLTE` after `IHDR`, it preserves the input bytes corresponding to `IHDR` and only mutates the bytes after them for reaching `PLTE`. Specifically, we consider the following two types of conditionals where we can produce input ranges to differentiate one state from another.

The first type consists of loops that iterate over continuous input bytes, *i.e.*, the first taint access in a loop iteration starts at an input location (right) after the last taint access of its previous iteration. Such a loop is widely used in programs to parse continuous data chunks, *e.g.*, input bytes of `IHDR` and `PLTE` form two continuous data chunks in the same size. For this case, we can use the index interval between different taint accesses over the same key variable to divide a variable-length input into a series of data blocks. Suppose the input reaches the prior state in a loop iteration whose first taint access index is idx_i and the first taint access index in the next iteration is idx_j. The extracted data block is $s[idx_i, idx_j]$. To reach a dependent state after the prior state, we should only mutate input bytes after idx_j and preserve the data in $s[idx_i, idx_j]$. In particular, we can mutate data blocks with variable length using this information, *e.g.*, when the data block has a length that is dynamically specified or when it ends with specific delimiters.

The other type consists of loops that iterate over a collection of control-flow-related local variables of the same type. Although the corresponding input bytes may be spread across the entire input, the data structures of the variables in different iterations are the same. Input bytes that influence the same data structure usually have the same layout. Therefore, by knowing the dependent input bytes of such one local variable, it is possible to identify and mutate the dependent input bytes of the other variables. Suppose we want to mutate input bytes to assign the value of variable b to variable a in the same type, in order to revisit a previously visited state. Let T denote the input bytes of a variable. We would recursively mutate T_a using T_b as follows, where $x, y, \ldots$ are the attributes in T. When

T_a is primitive, its value is set by replacing all of its influencing bytes with the ones of T_b.

$$update(T_a, T_b) : \begin{cases} T_a = T_b, & \text{if } T_a \text{ is primitive} \\ update(T_a.x, T_b.x), update(T_a.y, T_b.y), \ldots \end{cases}$$

3.5 Micro Directed Fuzzing

This section describes how DSFuzz runs micro directed fuzzing for making a state transition. Overall, this process is similar to traditional directed fuzzing. The key difference is that it produces feedback information for the state transition management phase. As a first step, DSFuzz performs input prioritization to select good inputs for reaching the target. It then runs the following input mutation process for each branch condition.

For each variable in the branch condition, DSFuzz analyzes its taint sources recursively to identify the corresponding critical bytes in the input. DSFuzz skips the variables whose values do not derive from external inputs. After analyzing all variables, it collects the identified critical bytes and mutates these bytes to satisfy the target condition.

3.5.1 Input Prioritization. We conduct input prioritization to select the good inputs for further mutation. For progressively satisfying the indirect control dependencies, we only choose inputs that lead to a transition path through all the reached dependent states (§3.4.2). When multiple such inputs exist, we prefer the one closest to the target condition. To evaluate the distance between a seed input and a target condition, we use the minimum distance between any branch condition that the input satisfies and the target condition. The branch condition with the shortest distance to the target should be the next condition that DSFuzz flips to take an unexplored branch towards the target.

We compute the distance between a condition c and a single target condition t by counting the minimum number of conditions DSFuzz needs to explicitly flip. In the equation below, $Suc(c)$ denotes the successors of c, $Suc(c, t)$ denotes the successors of c that can reach t. The distance between two conditions is zero when they are the same one. The distance between a condition and the target is considered infinite if none of its successors can reach the target. This is because the program will not reach the target through the branch condition taken by its predecessor. If all successors are capable of reaching the target, DSFuzz does not need to flip any input byte to proceed one condition closer to the target. Such conditions save mutation energy. To encourage visiting such conditions, we do not increase the distance by one. Instead, we set the minimum distance of its successors as its distance. When some, but not all, successors are capable of reaching the target, DSFuzz needs to flip certain bytes to evaluate an additional condition. This ensures that the program proceeds towards the target through one of the successors reachable to it, rather than through those that cannot reach it. The distance to the target is one plus the minimum distance of the successors that can reach the target.

$$d(c,t) = \begin{cases} 0 & \text{, if } c = t \\ \infty & \text{, if } Suc(c,t) = \Phi \\ \min_{s \in Suc(c,t)} d(s,t) & \text{, if } Suc(c,t) = Suc(c) \land Suc(c) \neq \Phi \\ 1 + \min_{s \in Suc(c,t)} d(s,t) & \text{, otherwise.} \end{cases} \tag{1}$$

Although we specify only one target at a time in our micro fuzzing loop, we can compute the distance between a condition and a set of target conditions using Equation 2, in which $R_T(c)$ denotes the set of targets that are reachable from c. The distance is zero when c is in the target set, and is infinity when none of the targets is reachable from c. In other situations, the distance is determined by computing the harmonic mean of the minimum distances from either c, its predecessors, or even predecessors of predecessors to each respective target. This method serves to prevent the overall distance from being needlessly inflated by the presence of infinite distances to certain targets.

$$d(c,T) = \begin{cases} 0 & \text{, if } c \in T \\ \infty & \text{, if } R_T(c) = \Phi \\ (\sum_{t \in T} \min_{c' \in Pre(c) \cup \{c\}} d(c',t)^{-1})^{-1} & \text{, otherwise.} \end{cases} \tag{2}$$

In the following example, two targets are nested in different branches of the same if-else conditional. Using the metric in prior work [34], the distances of both `condition_b` and `condition_c` are four, as they have only one successor to the targets. The distance of `condition_a` is two, which is even smaller than the distance of any of its successors. This would discourage the fuzzer from exploring deeper states after reaching `condition_a`.

```
1  while (condition_a) {
2    if (condition_b) {
3      if (...) { if (...) { if (...) {
4        ... // target 1
5      }}}
6    } else { // condition_c = !condition_b
7      if (...) { if (...) { if (...) {
8        ... // target 2
9      }}}
10   }
11 }
```

Listing 3: A simplified example showing distance evaluation.

Our metric in Equation 1 and Equation 2 instead can mitigate this problem. It ensures that the distance of `condition_a` (*i.e.*, $\frac{5}{2}$) is larger than those of `condition_b` and `condition_c` (*i.e.*, $\frac{20}{9}$), guiding the fuzzer to reach deeper states.

3.5.2 Taint Source Analysis. When interesting inputs are discovered in a micro-fuzzing campaign, we run byte-level taint tracking (line 17 in Algorithm 2) for later identifying the critical input bytes of a variable. To obtain the taint sources for a given variable v, we analyze its taint information collected from the byte-level taint tracking. In most cases, we can collect a determined number of input bytes and include them as critical bytes. Sometimes the mapping between the taint source and the variable is handled by some external function. For example in `if (!(f = fopen(fileName->c_str(), "wb")))`, we cannot directly obtain the specific input bytes that affect the variable `f` (either *true* or *false*). For these special cases,

we store pairs of argument values and return values of the external function. This helps us determine which argument values we can reuse to change the return value to a different one.

3.5.3 Value Exploration. We conduct value exploration to determine the value to set for the dependent input bytes. In particular, we do not mutate the input ranges that are already used for prior state transitions. We apply the following strategies to help determine what values we should mutate the dependent input bytes into.

Constant Variable. We model the possible constant choices of variables and directly copy them to the dependent input bytes of those variables. This includes the constant values or strings that are compared with the tainted key variables in the conditional. In addition to assigning directly a value that is used in the `cmp` or `switch` instructions, we observe several functions for comparing byte arrays. These include `bcmp`, `memcmp`, `memmem`, `strcmp`, `strncmp`, `strcasecmp`, `strncasecmp`, `strcasestr`, and `strstr`. If their arguments are known constant values, we would also directly use these values in the input to meet the condition. When the variable's value is not derived from some input bytes but from the number of dependent states, we break the micro fuzzing and try different dependent state combinations (§3.4.2).

Length Variable. If the value to be mutated is a length variable (*e.g.*, a variable obtained by evaluating the length of a data block), we also need to change the length of the corresponding data block to match the expected value. More specifically, when comparing the value of the `sizeof`, `size`, `strlen`, and `length` functions with a constant value in a branch condition using `>`, `<`, or `=`, we adjust the length of the dependent input bytes of the function's argument accordingly to satisfy the condition.

Random Exploration. If all the prior strategies do not fit, we fall back to random exploration. We transform the comparisons into constraints and mutate the continuous influencing bytes one by one, ordered by the number of branch conditions they affect.

4 IMPLEMENTATION

We implement a prototype of DSFuzz's fuzzing loop on top of ParmeSan [34], with additional feedback mechanisms for progressively satisfying indirect control dependencies. We implement a static analysis for constructing SDG based on LLVM passes, and build our dynamic taint analysis based on Polytracker [4]. In the following, we provide the most important technical details.

4.1 SDG Construction

We construct an SDG on top of the LLVM IR. As we illustrated in §3.2, SDG consists of the direct and indirect control flow dependencies between basic blocks and their corresponding states. In contrast to building traditional function-level control flow graphs (CFGs), which only contain intraprocedural direct control flow dependencies between blocks, building an SDG requires interprocedural information. Thus, we need to merge individual function-level control flow graphs with a global call graph (CG), thereby producing an interprocedural control flow graph (ICFG). Based on the ICFG, we can add indirect control dependencies and exclude basic blocks without any direct or indirect control dependencies to create an SDG.

To construct the ICFG, we start by building a static call graph from the LLVM IR. Even though we over-approximate the program's behavior by including all possible caller-callee pairs, the static call graph does not include indirect call targets that are only visible during dynamic execution. To incorporate these indirect targets, we dynamically update the call graph by running an instrumented program. We instrument all indirect call sites to enable dynamic identification of the indirect call targets. This allows for dynamically updating the indirect call edges in the call graph.

We construct the SDG by performing data flow analysis on top of the ICFG. To identify indirect control dependencies, we conduct backward data flow analysis from each conditional to collect variables that affect its value. We search for `Store` instructions to these variables and check their relative position to their data-flow-related conditionals to determine the presence of indirect control dependencies.

As state dependencies may change after an indirect call has been executed, we update SDG as follows. When dealing with an indirect call instruction, we start by running the SDG builder on the called function to update dependencies within the callee function. Next, we collect the data flows between the arguments to complete the data dependencies in the caller function. If the indirect call instruction is generative §4.2, we also update the dependency between the instruction source and the result. After that, we update the dependencies brought by the instruction to SDG.

We use SDG to calculate the likelihood of satisfying the condition for a particular branch. This calculation requires the number of branches of the conditional. If two conditionals depend on the values of the same set of variables, we would merge them into one. In this case, the number of branches in the merged conditional equals the sum of branches in each individual conditional.

4.2 Precise Data Flow Analysis

We run static data flow analysis at the bitcode level, where the actual data flow only takes place where the memory is read or written. This enables us to model the possible source of our data flows using `Store` and sink of our data flows using `Load`. Besides, we identify definition instructions using `Load` and `Alloca`. The remaining instructions are considered generative if they create new LLVM values.

We also run static taint tracking from the arguments of the main function and the I/O-related function calls to the states in the SDG, from which we identify control blocks that depend on external inputs. This information is used later to determine whether we can only flip a control block through indirect control dependencies.

We found that indirect control dependency does not hold when `Store` and `Load` are actually referring to different bytes of the same variable; such dependencies should be removed. For example, the variable `png_ptr->mode` consists of a collection of flags that label whether a particular chunk is parsed. Each flag only refers to a few bytes. The flag checks are implemented using pairs of `&=` and `|=` operations with the same constant value. We would remove the indirect control dependency (*e.g.*, `|= HAVE_IHDR` and `&= HAVE_PLTE`) if the constant value is not the same.

4.3 Dynamic Taint Analysis

Dynamic taint analysis collects all input bytes that affect a prior state from its taint access records. A taint access record tells which code block accesses which data block(s). The input stream processing loop is detected by analyzing the taint access pattern to check if the same code block accesses a continuous series of data blocks. We build our taint analysis on Polytracker [4], instead of Angora [12] or ParmeSan [34] for the following reason. Their taint analyses all output the same influencing bytes for a given variable because they are all built on top of the LLVM Data Flow Sanitizer [3], yet Polytracker offers two advantages over the taint analyses in Angora and ParmeSan. First, unlike the taint analyses in Angora and ParmeSan, which can only track the provenance of up to 16 taints at once, Polytracker can track up to $2^{31} - 1$ taints. Second, it can track every byte of input at once while imposing negligible performance overhead for almost all inputs.

5 EVALUATION

We extensively evaluate the effectiveness of DSFuzz by conducting the following experiments. First, we demonstrate that many applications have deep code locations that can only be reached via certain dependent states in a preliminary study (§5.2). We then show that DSFuzz helps efficiently reach bugs in these locations that are hard to reach by state-of-the-art directed fuzzers (§5.3). In particular, DSFuzz can detect eight new bugs that other tools cannot find, and we find that such hard-to-find bugs could persist for long periods of time (§5.3.3). We also conduct a component-wise analysis to explain the effectiveness of our design choices (§5.4).

5.1 Datasets and Setup

We chose two widely used datasets for our evaluation. The first one is the Fuzzbench benchmarks [33], which is Google's newest fuzzer benchmarks composed of 21 different programs. The other is Magma [21], which contains 138 verified bugs in nine programs with clear documentation and fault conditions. It contains the second-largest variety of bugs by CWE and the second-largest ratio of the number of bugs to the number of targets, after the CGC [2] and LAVA-M [15] datasets, respectively. We found that 16 out of its 138 targets were deep-state targets. The CGC dataset was not chosen because it required manual effort to analyze the bug reports, and some bugs would not cause a crash even with proof of vulnerability. We did not choose the LAVA-M dataset because most of its bugs were in shallow program states that had no indirect control dependencies. This does not align with our goal of exploring hard-to-reach bugs in deep states.

Since five out of seven programs in Magma (*e.g.*, libpng, libxml2, openssl, sqlite3, and PHP) are included in the Fuzzbench benchmarks as well, we merged these two datasets into one. These five programs are of different versions in Magma and Fuzzbench. For directed fuzzing, we selected the versions and seeds in Magma that contained bug condition information §5.4. We removed three programs in Fuzzbench from our dataset as they cannot be compiled under clang. We also removed PHP from Magma since it crashed under our instrumentation. In total, we got a dataset containing 19 different programs, including 13 from Fuzzbench and 6 from Magma. For the case `libjpeg-turbo`, we had to create a test harness that invokes the functionalities for processing the input stream, as the original test harness only processes the file header.

We perform our experiment on an x86_64 server with four 24-core 2.10GHz Intel Xeon Platinum 8160 CPUs running Debian 9. We assign each fuzzer to a dedicated core and leave 20% of the available cores unused to minimize interference. We enable the Address Sanitizer [39] for each fuzzing campaign and also enable the Undefined Behavior Sanitizer [32] for the applications that can be built under it. We compute the p-value in the Mann-Whitney U test between our tool and the tested baselines following the suggestion from Klees *et al.* [24].

5.2 Preliminary Study

Our research is motivated by the observation that some bugs are nested in deep locations and reaching them requires satisfying certain indirect control dependencies. We manually analyzed the dataset and found that 12 of the 19 programs contained indirect control dependencies, for which our tool can assist in satisfying them. Since indirect control dependencies are widely found in various programs, deep locations containing several indirect control dependencies may be practical. Using the state dependency analysis, we identified 139 potentially buggy code locations having indirect control dependencies. Using a value of $k = 0.005$ leads to 47 of these code locations being considered deep (§2.2). In practice, we found this boundary value to be strict enough to filter shadow bugs while producing a set of deep targets. Users can adjust the value of k to adjust the number of targets identified. Intuitively, these deep targets would be hard to reach for fuzzers without using information about the indirect control dependencies. These deep targets may not be properly tested due to the difficulty in reaching them. We tested our hypothesis by running directed fuzzing on the deep targets identified by DSFuzz in §5.3. We found that our tool can reach these locations efficiently and uncover previously unknown bugs, whereas existing approaches have trouble reaching them.

5.3 Bug Detection

We first demonstrate DSFuzz's good performance in detecting deep bugs compared with state-of-the-art directed fuzzers (§5.3.1). We then show that DSFuzz also performed generally better than other directed fuzzers in reproducing Magma bugs (§5.3.2), and it detected previously unknown real-world bugs that other tools failed to detect (§5.3.3).

5.3.1 Detecting Deep Bugs. DSFuzz was first compared with two state-of-the-art directed fuzzers AFLGo [10] and ParmeSan [34]. Other related works were either unavailable [11, 25, 47] or cannot support the dataset we used [22]. In particular, we found that Beacon [22] crashed when we tried to apply it to Fuzzbench. As a standard setting in evaluating directed fuzzers, we set a timeout limit of six hours for each target and give each target five independent trials [10, 11]. This time limit does not include the preprocessing time of AFLGo and ParmeSan. DSFuzz evaluates distance dynamically and does not need such preprocessing.

The targets for directed fuzzing were collected from two sources. First, we used the 47 deep and potential buggy locations identified by the state dependency analysis. Second, we used the known buggy

Table 1: Mean time to trigger deep bugs for each directed fuzzer.

Bug Idx	AFLGo		ParmeSan		DSFuzz
	Time(s)	p-Val	Time(s)	p-Val	Time(s)
1	745	0.007	1877	0.005	1026
2	–	0.003	15060	0.003	8803
3	–	0.006	15506	0.046	9193
4	–	0.005	–	0.005	13872
5	–	0.062	–	0.006	17179
6	–	0.015	–	0.003	10522
7	–	0.006	–	0.006	6626
8	–	0.03	–	0.11	10080
9	–	0.003	–	0.004	5608
10	–	0.006	–	0.006	4002
11	–	0.005	–	0.005	5786
12	–	0.006	–	0.006	15244

– denotes timeout (six hours).

Table 2: Mean time to trigger Magma bugs for each directed fuzzer.

Bug	AFLGo		ParmeSan		DSFuzz
	Time(s)	p-Val	Time(s)	p-Val	Time(s)
PNG003	16	0.005	102	0.006	38
PNG006	–	0.016	136	0.005	59
TIF005	–	0.002	14937	0.002	437
TIF006	–	0.006	15256	0.006	544
TIF007	6224	0.006	3821	0.006	306
TIF012	10159	0.006	14435	0.006	533
XML017	82	0.005	1460	0.004	197
PDF010	14583	0.006	–	0.006	567
PDF016	3914	0.005	452	0.003	146
SSL002	493	0.006	1923	0.008	243
SSL003	296	0.003	1155	0.006	206
SSL009	–	0.005	19810	0.006	664
PNG001	–	0.005	–	0.005	228
PNG002	–	0.033	–	0.002	1695
PNG007	–	0.006	–	0.006	235
TIF002	–	0.006	–	0.006	495
TIF003	–	0.003	–	0.005	12341
TIF004	–	0.002	–	0.002	18365
TIF006	–	0.005	–	0.005	17232
TIF009	–	0.012	–	0.006	675
XML001	–	0.002	–	0.003	12645
XML005	–	0.006	–	0.006	3372
XML009	–	0.004	–	0.004	258
PDF009	–	0.006	–	0.006	2390
PDF011	–	0.003	–	0.002	223
PDF017	–	0.006	–	0.006	17235
PDF020	–	0.005	–	0.006	4727

– denotes timeout (six hours).

locations from the Magma dataset to avoid a biased selection of targets in performance evaluation. Because of the page limit, we only present the bugs triggered within the time budget.

For our identified deep locations, DSFuzz could reach 39 of them within the time budget, AFLGo and ParmeSan could reach only 11 and 9, respectively. AFLGo and ParmeSan could reach some deep targets because their control dependencies could be easily satisfied with simple mutations from the initial seed inputs. Among the deep locations, DSFuzz identified 12 bugs, including three that remain unfixed in the recent version (§5.3.3). AFLGo and ParmeSan can detect 1 and 3 out of these 12 deep bugs, respectively. We present the detailed result of reaching these bugs in Table 1.

5.3.2 Reproducing General Bugs. We present in Table 2 the results for reproducing the Magma bugs, of which five are deep targets and could only be reached by DSFuzz. DSFuzz could reproduce other 10 non-deep bugs that other tools could not. Additionally, it outperformed existing directed fuzzers in reproducing all bugs except for PNG003 and XML018, which had shallow bugs. DSFuzz took more time as it had an overhead for setting up the dynamic taint analysis, which was not required in this simple example. The result indicates that DSFuzz is more appropriate for detecting bugs in deep states.

Table 3: Real-world bugs found by DSFuzz.

Project	Version	Bugs	Bug Type
libjpeg-turbo	2ee7264	2	buffer overflow
libpng	b7ea74c	3	memory leak, assertion error
libtiff	b51bb15	2	integer overflow, assertion error
libxml2	0137d98	1	memory leak
poppler	f7f4ae8	1	buffer overflow
Total		9	

5.3.3 New Bugs. This section explores whether DSFuzz can detect unknown bugs in real-world programs. We applied a timeout limit of 24 hours and reran DSFuzz for the 19 programs in the dataset. This experiment used the latest versions of these programs. Nine bugs were detected across five projects. The detailed results are shown in Table 3. To deduplicate the bugs, we first used AFL-CMin on the reported crashes, then compared the call stack, and finally performed manual verification. In addition, we ran other tools for 24 hours and found that only one assertion error in libpng can be reproduced by ParmeSan; the other eight bugs could not be found by any other tool, including general fuzzers in §5.4. We responsibly disclosed the newly detected bugs to the relevant developers; six have been confirmed and four have been fixed so far.

The detection of new bugs confirms our hypothesis that long-lasting bugs may reside in deep code locations that are hard to test. For example, the buggy memory allocation in Listing 4 was introduced in version 1.2.1 in 2001 and remained for over 20 years until it was fixed in September 2022.

Before reaching the buggy location, the program has to pass a series of conditionals containing a branch that leads to `return`.

```
1    if (!(png_ptr->mode & HAVE_IHDR))
2        png_error(png_ptr, "Missing IHDR before tRNS");
3    else if (png_ptr->mode & HAVE_IDAT) return;
4    else if (info_ptr != NULL && (info_ptr->valid &
       PNG_INFO_tRNS)) return;
5    if (png_ptr->color_type == GRAY)
6        if (length != 2) return;
7    else if (png_ptr->color_type == RGB)
8        if (length != 6) return;
9    else if (png_ptr->color_type == PALETTE){
10       if (length > (png_uint_32)png_ptr->num_palette ||
       length > PNG_MAX_PALETTE_LENGTH) return;
11       if (length == 0) return;
12   }
13   else return;
14   if (png_crc_finish(png_ptr, 0)) return;
15   ...
16   //bug location
17   (png_bytep)png_malloc(png_ptr, (png_uint_32)
       PNG_MAX_PALETTE_LENGTH);
```

Listing 4: A simplified real-world bug example.

Without proper direction, the fuzzer has a high chance of spending energy on exploring those `return` branches and would not reach the deep buggy location in a limited time budget. Even for directed greybox fuzzers, some constraints are hard to satisfy as they require the existence of variable setting operations that are not reached in the current control flow. In particular, the prior three conditionals that check the value of `png_ptr->mode` and `info_ptr->valid` cannot be altered easily. Simple input byte mutations cannot directly update the value of these variables. Changes can only be made by code that processes the prior data blocks. To successfully solve these constraints, the input should contain an `IHDR` data block before the `tRNS` data block and should not contain an `IDAT` data block before it. DSFuzz can generate such data blocks using its special design to satisfy indirect control dependencies.

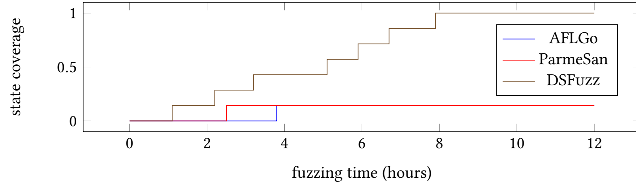

Figure 7: Dependent state coverage for each directed fuzzer.

This example also demonstrates the effectiveness of our micro directed fuzzing strategy for reaching deep states progressively. In particular, this bug is nested in a deep code location with seven dependent states. We present the progress for reaching these dependent states for each directed fuzzer in Figure 7. As the diagram illustrates, AFLGo and ParmeSan could reach some dependent states at an early stage, but they had difficulty reaching those that had indirect control dependencies. Thus, their dependent state coverages increased after 3.8 and 2.5 hours, respectively, but could not be improved further. DSFuzz, however, gradually increased its dependent state coverage until it reached the deep code location after 7.9 hours.

5.4 Component-wise Analysis

We aim to evaluate our contributions in identifying deep targets, exploring state transition paths toward these targets, and mutating inputs to trigger transitions. Since only DSFuzz addresses these challenges, we conducted component-wise analyses to demonstrate its effectiveness. As a first step, we tested whether DSFuzz can detect general bugs and compared it with state-of-the-art general fuzzers. To do so, we set a random unexplored branch as the target of each micro directed fuzzing campaign. Next, we used the 12 deep bugs that DSFuzz triggered in Table 1 as the test cases, and collected how many bugs it could still trigger without the transition path exploration and input range identification components. We present the detailed results in Table 5.

General Bug Detection. The unique feature of DSFuzz is its special design for satisfying indirect control dependencies. Deep states must contain certain hard-to-satisfy indirect control dependencies, but other hard-to-reach blocks may also contain some (probably fewer) and DSFuzz could help satisfy them to reach those blocks. Therefore, we developed DSFuzz$_{und}$, the undirected mode of DSFuzz, to test its ability of general bug detection. DSFuzz$_{und}$ targets randomly one unexplored branch condition for each micro directed fuzzing campaign. We compare DSFuzz$_{und}$ with a number of different general fuzzers: 1) AFL [45], a classic mutation-based greybox fuzzer; 2) AFLSmart [37], a greybox fuzzer with input-structure awareness; 3) Mopt [29], an AFL-based fuzzer with optimized mutation scheduling; 4) Angora [12], a fuzzer with taint propagation analysis; and 5) Redqueen, a fuzzer that tries to solve checksums and magic bytes. We chose them as they are state-of-the-art tools related to the optimizations in DSFuzz. We are aware of recent related works in fuzzing with taint inference [19, 28], but do not include them in our comparison as these tools are not available at the time of writing. We ran each tool for 24 hours with five repetitions.

The detailed results are presented in Table 4. We evaluate fuzzers using the mean edge coverage and the number of detected bugs.

Table 4: Edge coverage and bug detection for each general fuzzer.

Project	AFL	AFLSmart	Mopt	Angora	Redqueen	DSFuzz$_{und}$
freetype2-2017	20362	20222	20279	17018	27365	21399
harfbuzz-1.3.2	8404	8403	8401	7124	8632	8426
jsoncpp_jsoncpp_-fuzzer	638	638	638	599	638	638
lcms-2017-03-21	3021	3129	2913	4292	3257	4520
libpcap_fuzz_both	97	102	96	88	3330	3109
mbedtls_fuzz_-dtlsclient	8238	8701	8433	8129	8240	8603
openthread-2019-12-23	6207	6011	6212	5916	7149	6827
proj4-2017-08-14	5322	5348	5215	17278	5062	18237
re2-2014-12-09	3529	3532	3529	3037	3525	3238
vorbis-2017-12-11	2167	2167	2171	2016	2167	2167
woff2-2016-05-06	1862	1858	1857	101	1838	1858
zlib_zlib_uncom-press_fuzzer	961	965	965	923	961	965
libjpeg-turbo	3690	3665	3752	3406	3546	3802
libpng	1945	2122	1944	2183	2089	2328
libtiff	36563	36685	36397	29140	34657	35021
libxml2	12625	12817	12473	9823	12471	12784
openssl	13774	13772	13775	13840	13775	13912
poppler	38726	38760	39061	37282	37159	39872
sqlite3	34819	34925	34651	27749	32988	34193
Number of Bugs	4	3	5	0	3	5
p-Value	0.006	0.006	0.006	0.006	0.006	

Table 5: Mean time to trigger deep bugs for each tool.

Bug Idx	DSFuzz $_{byte}$	DSFuzz $_{rand}$	DSFuzz
1	2823	1394	1026
2	11232	9237	8803
3	18332	13382	9193
4	–	–	13872
5	–	–	17179
6	–	20229	10522
7	–	6873	6626
8	–	11298	10080
9	–	6023	5608
10	10293	4428	4002
11	–	5912	5786
12	–	–	15244

– denotes timeout (six hours).

We also present one mean p-value for each compared tool in the table. In general, we found that DSFuzz performed the best in 9 out of the 19 cases, especially in the four projects that contained many indirect control dependencies, including lcms, libjpeg, libpng, and poppler. In them, DSFuzz outperformed the second best tool by 5.3%, 2.3%, 6.6%, and 2.1%, respectively. In these cases, DSFuzz did not mutate combinations of data blocks that would not result in new coverage. For the proj4 and openssl cases, DSFuzz performed the best as they contained constraints that can only be resolved by our dynamic taint analysis. There were four cases, *i.e.*, json, vorbis, woff2, and zlib, for which several tools could achieve a similar average coverage. These cases did not have complex roadblocks: a simple random mutation can satisfy most of the constraints. AFLSmart outperformed all the other tools in five cases, for which it utilized the input format information that was too complex to be learned by other tools. Redqueen outperformed all the other tools in four cases that included a lot of checksums and magic bytes. Although DSFuzz could produce some of the magic bytes correctly through its value exploration strategies (§3.5.3), it was inefficient due to its reliance on dynamic taint analysis, which took more time than the mutation strategy used in Redqueen. DSFuzz also found the most number of bugs, including one in libxml2 that other tools cannot find.

Transition Path Exploration. We developed DSFuzz $_{rand}$, which is a version of DSFuzz that does not order the potential valid transition paths according to the strategies in §3.4.2. Instead, it randomly chooses one of the potential paths. We observed that three bugs

cannot be reproduced under this setting, and all the remaining bugs required more execution time. As a result of its improper transition path exploration strategy, DSFuzz $_{rand}$ would explore a longer transition path containing unnecessary dependent states, which increased analysis time and prevented it from detecting the three bugs. This result shows that our transition path exploration strategy is important to the effectiveness of DSFuzz.

Input Range Identification. DSFuzz would produce the same influencing bytes of a given variable compared to existing techniques (§4.3), but can conservatively choose to only mutate some bytes according to the state information it dynamically maintains (§3.4.3). Such a design contributes to a substantial degree of its efficiency. We developed DSFuzz $_{byte}$, which is DSFuzz without input range identification. It would mutate over all the input bytes in each iteration as long as they are identified as critical bytes. We found that DSFuzz $_{byte}$ could only trigger four bugs. This is because DSFuzz $_{byte}$ cannot store the already explored context without input range identification. In the remaining eight bugs, the complex constraints required a particular order of data blocks, which was difficult to mutate without the information about the explored data blocks. Even for the four bugs that it triggered, DSFuzz $_{byte}$ took 132% more time on average to reach them, suggesting that our input range identification technique also helped satisfy some simple constraints.

6 LIMITATIONS

Taint propagation in a complex program with long paths might be time-consuming, even though we have attempted to limit the number of calls through progressive fuzzing. The problem can be further alleviated by monitoring only a limited number of new branches for taint access.

Our implementation fails to satisfy some constraints (magic bytes with complex computations) as solving them is not our primary objective. If needed, DSFuzz can skip these constraints in order to save fuzzing resources, or combine other tools, such as symbolic execution, to support them.

Currently, our tool does not support the analysis of binaries. Nevertheless, our solution still applies when we can get their bitcode from reassembly.

7 RELATED WORK

Researchers have proposed various techniques for exploring hard-to-reach code locations.

Directed Greybox Fuzzing. A large number of works aim to reach deep code locations by only spending fuzzing energy on deep targets. AFLGo [10] is the first to use directed greybox fuzzing. Ankou [30] leverages distance-based fitness function, dynamic PCA, and adaptive seed pool update to assist directed fuzzing. Ijon [7] explores deep state spaces via human-in-the-loop annotation. ParmeSan [34] locates targets by sanitizers and solves path constraints by taint tracking. CAFL [25] conducts mutations toward a sequence of constraints rather than a set of target sites. DSFuzz statically locates a number of deep locations and applies a distance calculator for fuzzing toward them.

Meaningful Input Generation. People also try to reach deep code locations by improving the quality of generated inputs. One direction is to model the input format to successfully pass the parsing stage, following pre-defined grammar [6, 40], semantics [20], or data blocks [37]. Recent work GRIMOIRE [9] proposes to automate the identification of structured input specifications, WEIZZ [18] automatically identifies data blocks. Another direction is to focus on some inputs with higher security impact [43]. DSFuzz first infers the range of data blocks and then focuses on mutating combinations of data blocks that might reveal new coverage.

Dependent States. Researchers have found that some taints alter program states and implicitly change control flows. Such program-state changes are not covered by traditional code coverage. They propose inference-based taint analysis to identify these taints and prioritize their mutation. GREYONE [19] identifies the dependence between bytes and constraints and proposes a model to determine which branch to explore, which bytes to mutate, and how to mutate in each step. InvsCov [17] augments code coverage by learning the relationships among program variables from the input corpus and dependencies of those variables over all the observed executions. PATA [28] identifies the critical bytes in inputs that correspond with sequentially visited variables and mutates these bytes. DSFuzz identifies dependent states that change the execution context of the current state, and mutates different combinations of these states.

Critical Bytes Mutation. Mutating the correct bytes that pass complex conditions (*i.e.*, roadblocks) has been a key problem in improving code coverage. Vuzzer [38] collects immediate values and mutates critical bytes that are likely to satisfy the constraints. Angora [12] uses taint propagation to solve path constraints without symbolic execution. REDQUEEN [8] solves the magic values and checksums by observing the arguments to function calls and comparing instructions via virtual machine introspection. Laf-intel [1] splits multi-byte comparisons into several single-byte comparisons at the compiler level. SAVIOR [13] prioritizes the concolic execution of the seeds that are likely to uncover more bugs. DSFuzz uses dynamic taint analysis to identify the critical bytes and help constraint solving.

8 CONCLUSION

Input generation is challenging when the target location is reachable only by satisfying indirect control dependencies. We present DSFuzz, a directed fuzzing scheme for efficiently reaching deep states and detecting new bugs. To satisfy constraints that are implicitly affected by indirect control dependencies, it identifies dependent state transitions and constructs them progressively. It also carries out micro directed fuzzing to reach each dependent state. We evaluated DSFuzz over two widely used benchmarks, showing that it outperforms state-of-the-art tools and detected eight new bugs that other tools failed to find.

ACKNOWLEDGMENT

The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14210219).

REFERENCES

[1] 2020. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/.

[2] 2020. DARPA Cyber Grand Challenge. https://github.com/CyberGrandChallenge.

[3] 2021. Data flow sanitizer - clang 13 documentation. https://clang.llvm.org/docs/DataFlowSanitizer.html.

[4] 2022. PolyTracker: An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracing. https://github.com/trailofbits/polytracker.

[5] Cornelius Aschermann. 2020. *Algorithmic improvements for feedback-driven fuzzing*. Ph. D. Dissertation. Ruhr University Bochum, Germany.

[6] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[7] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[9] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. GRIMOIRE: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA, USA.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, USA.

[11] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[13] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[14] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[15] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA, USA.

[16] Joe W Duran and Simeon Ntafos. 1981. A report on random testing. In *ICSE*, Vol. 81. Citeseer, 179–183.

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event.

[18] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th International Symposium on Software Testing and Analysis (ISSTA)*. Los Angeles, CA, USA.

[19] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.

[20] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[21] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[22] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[23] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[25] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event.

[26] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.

[27] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 11th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Paderborn, Germany.

[28] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA, USA.

[30] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Seoul, Korea.

[31] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[32] Clang User's Manual. 2022. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[33] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece.

[34] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.

[35] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, 1–29.

[36] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[37] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[38] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA, USA.

[40] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th International Symposium on Software Testing and Analysis (ISSTA)*. Online.

[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[42] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA, USA.

[43] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[44] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, USA.

[45] Michal Zalewski. 2015. American fuzzy lop.

[46] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. 2022. Ferry:State-Aware Symbolic Execution for Exploring State-Dependent Program Paths. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA, USA.

[47] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.