Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths

Shunfan Zhou, Zhemin Yang, and Dan Qiao, *Fudan University;* Peng Liu, *The Pennsylvania State University;* Min Yang, *Fudan University;* Zhe Wang and Chenggang Wu, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity22/presentation/zhou-shunfan

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths

Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu†, Min Yang, Zhe Wang‡ and Chenggang Wu‡
Fudan University, †The Pennsylvania State University, ‡State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
{sfzhou17, yangzhemin, dqiao18, m_yang}@fudan.edu.cn, †pxl20@psu.edu, ‡{wangzhe12, wucg}@ict.ac.cn

Abstract

Symbolic execution and fuzz testing are effective approaches for program analysis, thanks to their evolving path exploration approaches. The state-of-the-art symbolic execution and fuzzing techniques are able to generate valid program inputs to satisfy the conditional statements. However, they have very limited ability to explore the finite-state-machine models implemented by real-world programs. This is because such state machines contain program-state-dependent branches (*state-dependent branches* in this paper) which depend on earlier program execution instead of the current program inputs.

This paper is the first attempt to thoroughly explore the state-dependent branches in real-world programs. We introduce program-state-aware symbolic execution, a novel technique that guides symbolic execution engines to efficiently explore the state-dependent branches. As we show in this paper, state-dependent branches are prevalent in many important programs because they implement state machines to fulfill their application logic. Symbolically executing arbitrary programs with state-dependent branches is difficult, since there is a lack of unified specifications for their state machine implementation. Faced with this challenging problem, this paper recognizes widely-existing data dependency between current program states and previous inputs in a class of important programs. Our insights into these programs help us take a successful first step on this task. We design and implement a tool *Ferry*, which efficiently guides symbolic execution engine by automatically recognizing program states and exploring state-dependent branches. By applying Ferry to 13 different real-world programs and the comprehensive dataset Google FuzzBench, Ferry achieves higher block and branch coverage than two state-of-the-art symbolic execution engines and manages to locate three 0-day vulnerabilities in *jhead*. Our further investigation shows that Ferry is able to cover more hard-to-reach code compared with existing symbolic executors and fuzzers. Further, we show that Ferry is able to reach more program-state-dependent vulnerabilities than existing symbolic executors and fuzzing approaches with 15 collected state-dependent vulnerabilities and a test suite of six prominent programs. Finally, we test Ferry on LAVA-M dataset to understand its strengths and limitations.

```
1   int box_size, box_type, saved_size = -1;
2   while(true) {
3       box_size = read_box_size();
4       box_type = read_box_type();
5
6       switch(box_type) {
7       case encode("tx3g"):
8           int size;
9           if(saved_size < 0) {
10              // default branch
11              size = 0;
12          } else {
13              size = saved_size;
14          }
15          // integer overflow when size > 0
16          uint8_t *buf = new uint8_t[size +/box_size];
17          // buffer overflow
18          read_payload_to(buf + size);
19          continue;
20      case encode("trak"):
21          saved_size = box_size;
22      // other 84 cases
23      ...
24      }
25  }
```

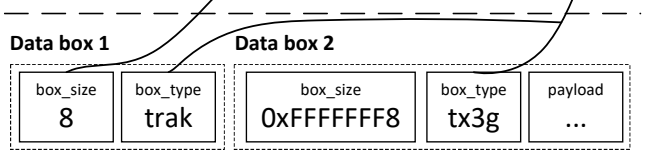


Figure 1: A motivating example.

1 Introduction

Symbolic execution [48] and fuzz testing [25] (or fuzzing) are well-known as effective approaches for dynamic analysis and software testing, thanks to their efficient program path exploration approaches. Unlike early work [77] which relies on randomly generated program inputs, state-of-the-art approaches [63, 68, 73–75, 78, 90, 91, 93, 96] focus on how to generate valid program inputs which satisfy the conditional statements and achieve high code coverage by reaching the branches behind them.

Despite the measurable success of previous works [68, 73, 90, 91, 96], existing approaches and tools have very limited ability to explore program-state-dependent branches in the finite-state-machine models implemented by programs. We now illustrate this issue through a real-world example shown in Figure 1. It implements a state machine to parse the incoming inputs of videos. State-of-the-art approaches can effectively explore the branches of line 6 since the value of `box_type` is unconditionally loaded from input in each iteration of the loop. However, to facilitate stateful program logic, real-world programs use certain variables to store the internal states of programs, and conditional statements can depend on the internal states instead of the current program inputs. For example, line 9 in Figure 1 depends on a program variable `saved_size`. When it is executed, `saved_size` is not determined by the program inputs processed in the current iteration (in which `box_type` is "tx3g" and line 7 branch is taken), instead, it has to be modified in an earlier iteration (which takes line 20 branch when `box_type` is "trak"). In this paper, we call `saved_size` a *state-describing variable*, and branches guarded by conditional statements which are data-dependent on such variables *state-dependent branches*.

The motivating example is bug-free in its default state where `saved_size` is -1 (so branch in line 11 will always be taken). Only a special set of input sequences can change the program state and trigger the integer overflow vulnerability in line 16: The sequence consists of two inputs, and each input is conceptually a data box. Each data box contains necessary meta-data `box_size` and `box_type` to describe its size and type, respectively, with optional `payload` data. The program implements a loop, and data boxes are handled sequentially during different iterations of the loop. The first "trak" data box changes program state by setting `saved_size` to its box size (a positive value). During the next iteration that handles the "tx3g" box, `size` is larger than 0 (since line 13 is taken), so attacker can trigger the vulnerability with a carefully-specified `box_size` value.

Such state-sensitive program logic breaks a de facto (implicit) assumption of state-of-the-art techniques [73, 74, 96]. That is, programs handle each part of inputs from a single file independently, and the effect of state changes caused by earlier inputs on how a later input is processed can be ignored. Instead, we observe that many real-world programs implement state-dependent branches in the context of state machines, to fulfill their application logic. Existing symbolic executors cannot efficiently explore the state-dependent branches and locate the vulnerability in our example (as discussed in Section 2), and we will show how our method finds it in Section 4.

We find that state-dependent branches are prevalent in many important real-world programs. Our manual vetting of the seven prominent programs of different categories which concern previous works [73, 74, 96] in Table 1 shows that they all implement non-trivial state machines containing tens or even hundreds of state-dependent branches. The existence of state-dependent branches degrades the path exploration efficiency of existing approaches in two ways. First, with scant knowledge of the internal program states, existing approaches cannot generate desired program inputs that explore a large set of program states, leading to a very limited exploration of state-dependent branches. Furthermore, a program statement can have various behaviors in different program states (e.g., line 16 in our motivating example is bug-free by default and only vulnerable when program state is changed), which means that it is not enough to just reach the statements once. Instead, they should be fully examined in different program states. This calls for a new metric to evaluate the state exploration capabilities of existing tools.

Table 1: Real-world programs of concern to QSYM, Angora and Matryoshka which contain state-dependent branches.

Program	**Category**
ffmpeg [23]	Audio/Video Converter
djpeg(libjpeg) [32]	JPEG Image Library
readpng(libpng) [35]	PNG Image Library
tiff2ps(libtiff) [14]	TIFF Image Utility
mutool(mupdf) [43]	PDF Viewer
readelf(GNU Binutils) [26]	ELF File Utility
file [9]	File Type Checker

In this paper, we propose the first program-state-aware symbolic execution framework *Ferry*, which requires no prior knowledge about program source code and input sequences. Ferry operates on compiled binaries. It automatically recognizes state-describing variables, uses them to guide symbolic execution and explores state-dependent branches. Besides, we propose several optimizations to further improve the efficiency of Ferry. We compare Ferry with two widely-used symbolic execution engines angr and KLEE, and three popular fuzzers AFL, Angora and QSYM on 13 real-world programs, including the seven programs listed in Table 1 and Google FuzzBench [56]. Our experiments show that with inferred state-describing variables, Ferry is able to cover an average of 38%/42% more basic blocks and 42%/47% more branches than these two symbolic executors, 18%/21%/8% more basic blocks and 22%/21%/8% more branches than three fuzzers, respectively. During experiments, Ferry successfully locates three 0-day vulnerabilities in *jhead* [8]. Furthermore, we demonstrate that Ferry can cover more program states. Specifically, we construct a test suite River which consists of six programs and 160 inserted vulnerabilities. The inserted vulnerabilities are distributed over different state depth, i.e., the least number of state-dependent branches involved in the path to the vulnerability. The test suite is then used to evaluate the symbolic executors and fuzzers above. Given the same time limit, Ferry locates all the 160 vulnerabilities while the other tools report at most 41% of the inserted vulnerabilities with shallow depth. We believe this test suite clearly showcases the effectiveness of Ferry, and provides a mean-

ingful benchmark for future works to evaluate their abilities to explore the deep logic (e.g., state-dependent branches) of real-world programs.

The main contributions of our work are as follows.

- We introduce program-state-aware symbolic execution, a new research direction that focuses on an under-investigated problem for symbolic execution.
- We offer new insights into addressing the non-trivial symbolic execution challenges faced by real-world programs with state-dependent branches. We propose novel algorithms and heuristics to implement a program-state-aware symbolic execution engine Ferry.
- We recognize practical challenges when applying Ferry to real-world programs and propose optimizations to further improve the efficiency and scalability of Ferry.
- We evaluate Ferry, showing that it is far more effective and efficient than state-of-the-art approaches and tools at locating "deep" security-related vulnerabilities in real-world programs.

2 Problem Statement and Analysis

Insight 1: A major hindrance to a symbolic execution engine is the ignorance of a program's internal states, which leads to an inefficient exploration of state-dependent branches.

State-of-the-art symbolic execution engines [68, 75, 90, 97] cannot efficiently explore the state-dependent branches (e.g., line 9 in motivating example), due to the difference between the logical program states and the symbolic states (discussed in other symbolic execution works [68, 90, 97], including execution paths, together with path constraints and mapping between all variables and symbolic expressions). The example in Figure 1 has three different logical program states[1], described by the different value combinations of `box_type` and `saved_size`: (1) "tx3g" with negative size value, (2) "tx3g" with non-negative size value and (3) "trak" with any size value. However, the given example has infinite number of symbolic states. For example, if an execution path keeps taking the "tx3g" branch in each iteration, it will shift to a new symbolic state every time it enters the loop, since each iteration introduces extra path constraints on inputs (i.e., appending a new "tx3g" data box). It can create an infinite number of different symbolic states, while it is actually exploring the same program state repeatedly. Such redundant exploration is meaningless and can easily lead to path explosion. Existing symbolic executors are not aware of such state-dependent semantic. They can either enumerate all the possible combinations of different box types, or hope that the engines "happen to" explore the program in the expected order.

[1] We ignore the omitted switch cases, while the number of program states is still far less than the number of execution paths even if we consider other cases.

It is worth noting that Figure 1 only shows a trivial example of state-dependent branch. Many modern programs have as many as thousands of state-dependent branches, and these branches can be nested. Carefully-specified state machines help developers avoid making mistakes when writing a program containing complicated application logic. Unfortunately, as illustrated above, their state-dependent branches prevent analysts and researchers from finding bugs and vulnerabilities within and behind them.

2.1 Challenges to Conduct Program-state-aware Symbolic Execution

Challenge 1: Program state inference. To bridge the gap between traditional symbolic execution engines and a program-state-aware symbolic execution, the most critical issue is how to infer the internal states of a given program. Unlike states of operating systems which are well-documented (as discussed in Section 7.3), in general, a program's states are bound to its implementation, and there is no explicit specification.

Since a program usually keeps track of its states with specific variables, i.e., state-describing variables, by identifying the state-describing variables, we can infer the program states. Unfortunately, recognizing such variables is challenging itself since there is little semantic or structural difference between state-describing variables and irrelevant ones. For example, a real-world program can remember a state with a variable of any type and with arbitrary name. The identification of state-describing variables greatly affects the effectiveness of program-state-aware symbolic execution. On one hand, if some state-describing variables are omitted, our symbolic execution may treat different states as the same one, leaving part of the states untouched. On the other hand, if we treat irrelevant variables as state-describing ones, the same program state may be executed many times, causing the exploration space of symbolic execution to explode.

Challenge 2: Runtime program state recognition. Even if we can identify the state-describing variables of a given program, it is non-trivial for a symbolic execution engine to recognize the current state. Specifically, real-world programs may not employ a specific enumeration variable to remember its current state. For example, the code snippet in Figure 1 can be thought to have two states: a normal one with `saved_size` ≤ 0 and a vulnerable one with `saved_size` > 0. As a result, it is difficult to recognize the current program state and determine whether a given program is executed in an explored state.

2.2 Characteristics of Real-world Programs with State-dependent Branches

To address the challenges above, we conduct an investigation on real-world programs with state-dependent branches to understand their characteristics. Specifically, we start from 106 independent programs of concern to and analyzed by AFL [18] (we exclude other 54 programs, e.g., Mozilla Fire-

fox and Internet Explorer which use multiple libraries or are not open-source)[2]. Our manual investigation shows that 91 out of 106 (86%) programs contain at least one state-dependent branch. Programs containing state-dependent branches are widely used as essential components in various aspects of software systems from network data transmission to multimedia data processing. More importantly, many of them are the building blocks of large projects and systems. For example, the Chromium project is powered by several libraries with state-dependent branches including *libpng* [35] and *libtiff* [14]. We further conclude their characteristics as follow.

- **C1: They receive a sequence of inputs from a single input source.** Apart from the configuration options (from global configuration files, environment variables or command-line arguments), each program receives input sequences from a single external input source such as a file or a socket. And since these programs are command-line applications or libraries, they receive no asynchronous inputs like signals or UI events.
- **C2: They handle inputs sequentially.** Instead of handling the entire input sequence at once, these programs handle one input (in the input sequence) at a time. Our motivating example is a typical example that contains an input-handling loop to continuously read data from a file. The input handled in current iteration of the loop is called the *current input*, and the inputs handled in previous iterations are called *earlier inputs*.
- **C3: They maintain the program states with one or more state-describing variables.** Program states are affected by some earlier inputs when there is a data dependency between the values of state-describing variables and these inputs. And these variables further change the program behaviors by affecting the branches taken for certain conditional statements.

In-scope Programs. Instead of trying to efficiently analyze any programs with internal states, we explicitly define the scope of target programs in this paper as the real-world programs with the above three characteristics.

Problem Statement. For programs with the above-mentioned characteristics, how to address the challenges identified in Section 2.1 for real-world programs so that state-dependent branches, such as the one shown in Figure 1, can be efficiently explored?

2.3 Model of Ferry

Definition of Program States. A state in a program can be characterized as follows: (1) a state is described by a set of state-describing variables; (2) when two states are different, at least one state-describing variable holds different values; (3) the program has different behaviors (i.e., taking different branch directions) in different states.

Automatically identifying state-describing variables is a challenging task. Our solution to tackle this challenge is based on characteristics of analyzed programs: As mentioned in Section 2.2, each program has a single external data source, which determines the execution of the program in such a way that the earlier inputs can affect the values of certain state-describing variables, which further determine the branches taken for some conditional statements when the current input is being processed.

During execution, program state is changing during the processing of input sequences. In this paper, we classify the state-changing events in concern into three categories: (1) a new state-describing variable is initiated; (2) the value of a certain state-describing variable is changed, which means that it is assigned a different value or the data constraints of its symbolic expression are changed; (3) a state-describing variable is released.

Specifically, the influence of input sequences on runtime program states can be specified with the following definitions: Firstly, we define each instruction that loads a part of the input sequence into a memory cell as an *input loading instruction* (e.g., `read_box_size()` and `read_box_type()` in Figure 1). Secondly, we call each instruction that initiates, releases or modifies the value or data constraints of a state-describing variable a *state-describing variable operation (**SDVO**)* instruction. With the definition above, the execution of an SDVO instruction will definitely lead to a state transition. Thirdly, we define the *data dependencies* between SDVO instructions and input loading instructions as follows: whenever a data dependency exists between a state-describing and an input loading instruction, we say that the corresponding SDVO instructions are *data dependent* on the input loading instruction. In practical, we catch such dependencies with dynamic taint tracking.

An intrinsic characteristic of analyzed programs is that data dependencies widely exist between SDVO instructions and input loading instructions. Ferry leverages this characteristic to identify state-describing variables.

Distinguishing Characteristics of State-describing Variables. State-describing variables bear the following two characteristics: (1) State-describing variables are explicitly/implicitly data-dependent on input data; (2) State-describing variables are checked in at least one conditional statement.

Remark. Although our approach directly exploits the three characteristics identified in Section 2.2, our technique is potentially applicable to all the programs that have the characteristic above, which means

- For a subset of the essential state-describing variables, their SDVO instructions are data-dependent on at least one input loading instruction;
- With different contents (i.e., values) of state-describing variables, the program may change its behavior by taking

[2] We present the detailed information of all the 160 investigated programs in https://drive.google.com/open?id=17XQpmufR0zdKv0c0hU70JTiMQkkOHCz-.

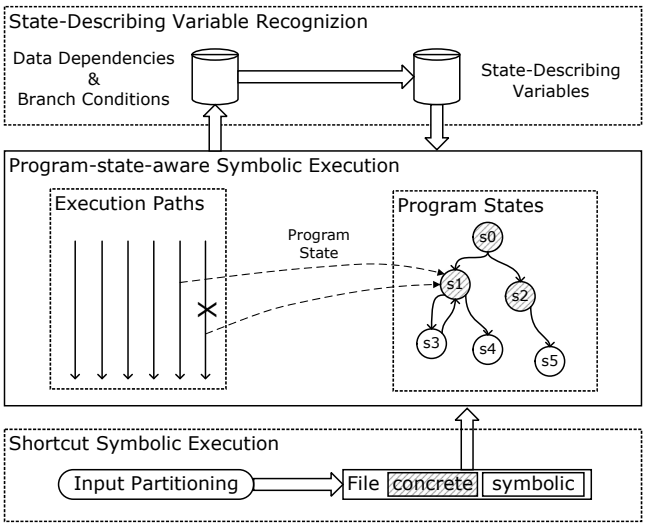


Figure 2: Overall architecture.

different branches.

3 Design

3.1 Overview of Ferry

Ferry operates in three steps, as illustrated in Figure 2. First, based on the insight that data dependencies exist between state-describing variables and inputs of programs, Ferry monitors the execution of a given program, and identifies the state-describing variables. Second, Ferry recognizes the runtime program states and drives a symbolic execution to explore a large set of program states. Furthermore, we observe that the complexity of real-world programs can affect the effectiveness of Ferry. Motivated by this observation, our third step further improves the efficiency by introducing two optimizations: state-reduction of inactive state-describing variables and shortcut symbolic execution.

3.2 State-describing Variable Recognition

In Section 2.3, we identify two distinguishing characteristics of state-describing variables, i.e., they are data-dependent on inputs (denoted as *InputDetermined* in the following paragraphs) and they are checked in at least one conditional statement (denoted as *BranchRelated*), and the variables with these characteristics are *StateDescribing*. It is worth noting that such characteristics of variables are recognized *globally*, which means if a variable is marked as *InputDetermined* in one execution path and marked as *BranchRelated* in another, we can mark it as *StateDescribing*. This section presents our automatic state-describing variable recognition mechanism based on the intrinsic characteristics of analyzed programs.

First, we apply a dynamic taint analysis from the external input source to recognize the *InputDetermined* variables. Our goal is to find the state-describing variables that depend on the input sequences. We tag all the inputs (in this case, from the video file, e.g., `box_size` and `box_type`) as *Tainted*, and trace their propagation in the program. In contrast to traditional taint analysis, our approach tracks the propagation of *symbolic* inputs instead of concrete ones. Specifically, we record the taint information as a specific type of data constraints on the symbolic inputs, so the taint tags are naturally propagated with the proceeding of symbolic execution, introducing no extra overhead. A variable is marked as *InputDetermined* if it receives a *Tainted* symbolic expression. In the given example, `box_size`, `box_type`, and `saved_size` are *InputDetermined*.

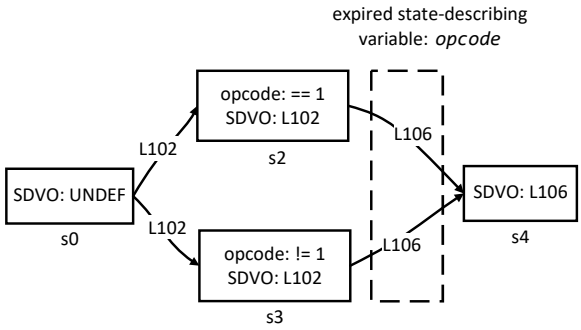
```
1    if (input > 0) {
2         state_var = CONST_VAL_1;
3    } else {
4         state_var = CONST_VAL_2;
5    }
6    if (state_var == CONST_VAL_1) {
7         tmp_var = …;
8    }
```

Figure 3: An control dependency example.

Apart from the explicit data dependencies, control dependencies, depicted in Figure 3, may also help identify state-describing variables. However, dynamic taint analysis generally does not cover control dependencies [81]. Actually, considering such dependencies is a double-edged sword. Although it augments the coverage, many irrelevant variables (false positives) could be falsely introduced. Thus, our analysis only considers the first-level control dependencies, that is, only the variables control-dependent on the others which are explicitly data-dependent on inputs are tainted. For example, in Figure 3, `state_var` is tainted since it is control-dependent on `input` which is explicitly dependent on inputs; while `tmp_var` is no more tainted since its guard condition checks `state_var`, which is control-dependent on inputs.

As revealed in Section 2.3, a program takes different branch directions in different states. Thus, for each conditional statement in the program, we record all the variables it accesses when checking the corresponding conditions, and tag them with *BranchRelated*. If an *InputDetermined* variable is used in a conditional statement, which means it is also *BranchRelated*, we mark it as *StateDescribing*. For example, `saved_size` in Figure 1 is *StateDescribing* because it is accessed by the conditional statement in line 9.

Identifying Invalid State-describing Variables. As mentioned in Section 2.3, variables have liveness durations and state transition happens when state-describing variables are released. For example, in Figure 4, `opcode` is state-describing. Supposing it is a local variable on stack, a function's local stack becomes invalid once the function returns. To avoid incorrectly recognizing program states with such expired variables, Ferry introduces two optimizations. First, the call to the `free()` function is regarded to be an SDVO instruction if the freed object contains state-describing variables. Second, once a function call is returned, we examine the state-describing variables, and discard the ones within the stack frame of the returned function. Besides, if any of the variables are discarded,




```
1     void caller(){
          ...
15        subfunc();
16        ...
17    }
      ...
100   void subfunc(){
101       int opcode = read_input();
102       if(opcode == 1)
103           do_something();
104       else
105           do_others();
106       return;
107   }
```

Figure 4: Removal of expired state-describing variable *opcode*.

we consider the return instruction as an SDVO instruction, and check whether a new state is introduced.

3.3 Program-state-aware Symbolic Execution

We now explain how to guide symbolic execution with the recognized state-describing variables. As introduced in **Challenge 2**, it is difficult to determine whether a given program is executed in an explored program state. To tackle this challenge, our solution is based on the following insight:

Insight 2: The state transitions in a program depend on the constraints on state-describing variables.

Symbolic execution engines record the constraints on symbolic inputs for every execution path, and ensure that any inputs that satisfy the same constraints will definitely take the same execution path. Besides, the program state information is captured by Ferry as state-describing variables. Thus, if two executions have the same constraints on every state-describing variable, they should follow the same state transitions.

As mentioned in Section 2.3, state transitions occur with the execution of certain SDVO instructions. In real-world programs, an instruction is an SDVO one under the following circumstances:

- **Initialization.** A new state-describing variable is initiated;
- **Data constraint alteration.** The constraints on a state-describing variable are changed by assignment or updated to take certain branch direction;
- **Variable release.** The release of a state-describing variable occurs with the end of its liveness.

Apart from state-describing variables, Ferry records where the latest state transition happens (i.e., the address of the SDVO instruction it last met). Two execution paths are considered to explore different program states if either their state transition locations or their constraints on the state-describing variables are different.

Then, once a state transition occurs, Ferry compares the current state with explored records. If a new unexplored state is observed, we record its SDVO instruction address and the constraints on the state-describing variables. Otherwise, we rollback the symbolic execution to explore different program states.

4 Algorithm of Program-state-aware Symbolic Execution

Algorithm 1: Program-state-aware Symbolic Execution

Input: Initial location l_0, initial program state s_0, instruction decoder `instrAt`
Data: Worklist W, program state store Ω, path predicate Π, symbolic store Δ

$W \leftarrow \{(l_0, s_0, true, \emptyset)\}$;
while $W \neq \emptyset$ **do**
  $((l, s, \Pi, \Delta), W) \leftarrow$ `pickNext`(W);
  switch `instrAt`(l) **do**
    `// handle assignment`
    case $v := e$ **do**
      if `isTainted`(e) **then**
        $s \leftarrow updateState(s, v, e)$;
      if $(\texttt{succ}(l), s) \in \Omega$ **then**
        break;
      $\Omega \leftarrow \Omega \cup (\texttt{succ}(l), s)$;
      $S \leftarrow \{(\texttt{succ}(l), s, \Pi, \Delta[v \rightarrow \texttt{eval}(\Delta, e)])\}$;
    `// handle conditional statement`
    case *if (e) goto* l' **do**
      $s\_true \leftarrow s$;
      $s\_false \leftarrow s$;
      foreach *state-describing variable m in* `eval`(Δ, e) **do**
        $s\_true \leftarrow updateState(s\_true, m, \Pi \wedge e)$;
        $s\_false \leftarrow updateState(s\_false, m, \Pi \wedge \neg e)$;
      end
      if `isSat`$(\Pi \wedge e)$ **then**
        $\Omega \leftarrow \Omega \cup (l', s\_true)$;
        $S \leftarrow S \cup \{(l', s\_true, \Pi \wedge e, \Delta)\}$;
      if `isSat`$(\Pi \wedge \neg e)$ **then**
        $\Omega \leftarrow \Omega \cup (\texttt{succ}(l), s\_false)$;
        $S \leftarrow S \cup \{(\texttt{succ}(l), s\_false, \Pi \wedge \neg e, \Delta)\}$;
    `// handle function return`
    case *return* **do**
      foreach *state-describing variable m in current stack frame* **do**
        $s \leftarrow updateState(s\_true, m, UNDEF)$;
      end
      $\Omega \leftarrow \Omega \cup (\texttt{succ}(l), s)$;
      $S \leftarrow \{(\texttt{succ}(l), s, \Pi, \Delta[v \rightarrow \texttt{eval}(\Delta, e)])\}$;
    case *halt* **do** continue;
  end
  $W \leftarrow W \cup S$;
end

The algorithm of program-state-aware symbolic execution is depicted in Algorithm 1. We maintain the state of symbolic

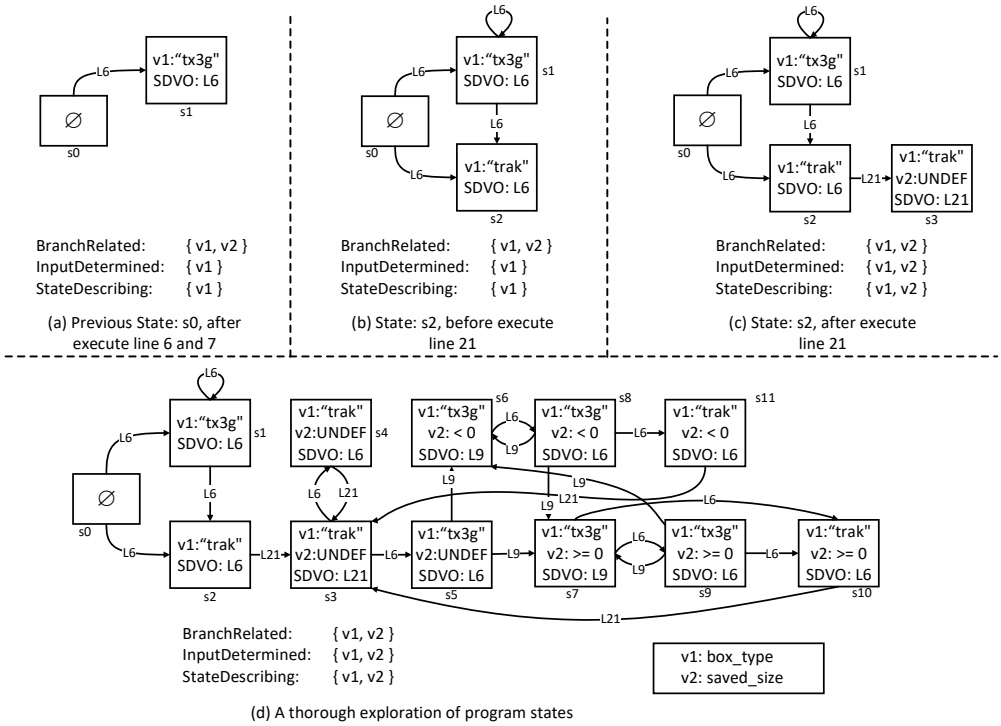


Figure 5: An example to illustrate our program-state-aware symbolic execution.

executor with (l, Π, Δ), in which a code location l records the address of the currently executing instruction, a path predicate set Π which stores the path constraints, and a symbolic store dictionary Δ, which maps runtime variables to concrete values or symbolic expressions. In addition, we introduce a state variable s, which stores the program state of current execution path, and a global state store Ω, which records the explored program states (including the address of latest SDVO instruction and state s). The program state s tracks three sets of variables as we illustrate in Section 3.2: *BranchRelated*, *InputDetermined* and *StateDescribing*.

By default, Ferry applies a breadth-first exploration strategy and the `pickNext()` function returns the next execution path to process. With a quad (l, s, Π, Δ) maintained for each execution path, we keep the program state s updated when an SDVO instruction is met. Specifically, a state-describing variable can be initialized or altered when an assignment or conditional statement occurs, and can be released when function returns. $updateState(s, v, e)$ function receives a original state s, the assigned variable v and the new value (or constraints) e, and returns an updated state. Besides, once we encounter a conditional statement, we leverage an SMT solver to check the satisfiability of its successors. Finally, only execution paths of unexplored states are added to the work list W for future process.

Figure 5 illustrates our symbolic execution with the motivating example. The first state-describing variable in our execution is the variable `box_type`, which is loaded from the input in line 4 (*InputDetermined*). In line 6, a switch instruction accesses it (*BranchRelated*), and our state-describing memory recognition marks it as *StateDescribing*. Thus, line 6 is an SDVO instruction, and by exploring its different branches, two new states are recorded (s_1 and s_2). If the state s_1 is first explored, the second execution of line 6 can determine s_1 as an explored state, and avoid traversing it again.

`saved_size` is another state-describing variable. A conditional statement in line 9 accesses it. However, not until the content from the input sequences flows into it (line 21) can we identify it as *StateDescribing*. Thus, the first execution of line 9 does not introduce new program states. Later when any execution path reaches line 21, `saved_size` is recognized as *StateDescribing*, therefore introducing a new state s_3. Then, our further execution to line 6 can enter branch "tx3g", since a new state-describing variable is introduced. The constraint on this memory location is further updated in the second execution of line 9. Thus, two new states (s_6 and s_7) are introduced.

5 Optimizations for Complex Real-world Programs

We notice that the complexity of real-world programs can affect the effectiveness of Ferry mainly in two ways: (1) real-world programs may have hundreds of even thousands of state-describing variables, which leads to a high overhead in state-describing variable tracking, and (2) real-world programs can have multiple state-dependent branches on an execution path reaching a "deep" vulnerability, accordingly, Ferry may get stuck in exploring the earlier state-dependent branches, and fail to explore the later ones. To tackle these practical problems, we further propose two optimizations, namely **State-reduction of Inactive State-describing Variables**

to reduce the unnecessary tracking of seldomly-accessed state-describing variables, and **Shortcut Symbolic Execution (SSE)** which enables Ferry to explore the "deep" states. It is worth noting that these two optimizations, although they do not necessarily apply to every program, do not affect the basic analysis proposed in Section 4.

5.1 State-reduction of Inactive State-describing Variables

To avoid exploring an already explored program state, Ferry needs to constantly compare the range (i.e., the set of all possible values) of each state-describing variable with the exploration history, causing nonnegligible performance overhead. To alleviate this problem, we propose an optimization, named *State-reduction of Inactive State-describing Variables*, which reduces the number of state-describing variables to track.

Instead of tracking all the state-describing variables, we set a number limit and only focus on those which are recently accessed (i.e., assigned new values or checked in conditional statements). Our investigation shows that state-describing variables are accessed with different frequencies. We monitor the execution of seven programs listed in Table 1 and focus on their access frequencies of different state-describing variables. Among all the state-describing variables used in these programs, 72% are checked less than five times, and the top 4% variables take up over 85% of all the variable accesses. This observation is also supported by our motivating example. The skewed distribution indicates that for many state-describing variables in a program, their values will remain unchanged for a long period of time. If an execution path does not access a state-describing variable, its value remains unchanged and it cannot cause a state transition.

The number of the tracked variables should be carefully selected. Tracking a low number may falsely ignore state-describing variables, causing different states to be recognized as the same one (thus paths are falsely pruned). And a high limit increases the number of variables to track, thus wasting time on constraint solving instead of path exploration. We conduct an experiment on the seven programs in Table 1 and try to figure out how different numbers of tracked variables affect the number of execution paths. We start from the limit of one state-describing variable to track, and find that we quickly run out of exploration paths on most of the programs within 30 minutes. On the other hand, a limit of over five state-describing variables has almost no effects on limiting number of comparisons between states. Finally, we find that tracking three variables helps us reduce the comparisons between states and reach deep program states, so we make it the default configuration in the following experiments.

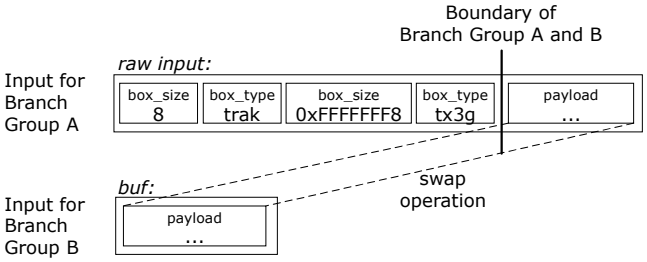


Figure 6: Our input partitioning on an example payload.

5.2 Shortcut Symbolic Execution with Input Partitioning

The execution path to the "deep" states in real-world programs is guarded by many state-dependent branches. Similar to path explosion, the number of program states grows rapidly with an increase in the number of state-dependent branches. Fortunately, we find that instead of trying to conquer all these branches, we can divide them into independent groups, which can be handled separately.

In real-world programs, developers do not handle the whole input sequence in one function, instead, the different parts of input sequence are handled by different groups of functions. For example, the input-handling loop in our motivating example only parses the `box_type` and `box_size` of each data box. The `payload` data, on the other hand, is read into the constructed buffer `buf` in line 18 and left to be processed by another function. Accordingly, the state-dependent branches in these functions also depend on different parts of the input sequence, and we can group these branches by the parts of input sequence which are handled by the functions to which they belong. Based on this observation, our idea is to bypass early-stage groups of state-dependent branches by selectively providing concrete program inputs.

When shortcut symbolic execution is enabled, manually-provided seed files are needed. With given input seeds, our shortcut symbolic execution proceeds as follows: First, SSE automatically recognizes the boundaries of different input parts which are handled by different groups of state-dependent branches with symbolic emulation. Then, with inferred boundaries, SSE selectively makes certain input partitions symbolic to symbolically execute the corresponding state-dependent branch groups in a state-aware way. It is worth noting that SSE proceeds on-the-fly during our state-aware symbolic execution. That is, Ferry with SSE first constrains the input data with the given seed file contents and applies symbolic execution (i.e., symbolic emulation). During this process, Ferry automatically recognizes the partition boundaries. Then, it tries to explore certain group of state-dependent branches by dynamically removing the constraints on the corresponding input parts.

Input Partitioning. An important problem here is how to automatically decide the boundaries in input sequence which separate the parts handled by different state-dependent branch groups. Let's assume earlier branch group A and later branch

group B handle two adjacent parts of the input sequence. When we locate the boundary between the two groups, we leverage a particular relationship between the function f which hosts group A and the function g which hosts group B. In particular, we observe that in 73 out of the 91 programs (80.22%) investigated by us, function f explicitly constructs a buffer for function g, and copies (or moves) the (unprocessed) data from its own buffer to the new one. Accordingly, we first monitor I/O functions (e.g., `fread()`) and mark the destination array as initial input buffer. Then, we recognize the data copy or move operations (e.g., `memcpy()`, `memmove()`) from one buffer to another in analyzed programs. The part of the input moved into new buffer is expected to be handled by another group of branches, and we set the boundaries between different parts. Figure 6 shows how our methodology works in the motivating example, in which the program code explicitly constructs a new buffer `buf`, and copies the `payload` to it (line 18). Then, a later state-dependent branch group uses the new buffer as the input source. We set a boundary before `payload`.

Shortcut Symbolic Execution. As illustrated above, an important property of input partitions is that they are handled by independent branch groups. If we use concrete values for the leading $N-1$ partitions, the corresponding early-stage branch groups will be executed with no path forked (i.e., shortcut). More importantly, the next move of the program is deterministic: it will read the Nth partition into buffer and use specific branch group to process it, so if we make Nth partition symbolic on-the-fly, we can target the corresponding branch group and explore it. When determining in which order to symbolize different partitions, we try to make the best use of provided file contents to drive Ferry to explore as many deep program logic as possible without path explosion, so we symbolize input partitions backward in order. That is, we first symbolize the last partition. When we are running out of execution paths, we further symbolize one previous partition and repeat the step.

6 Evaluation

In this section we evaluate the effectiveness of Ferry. Our evaluation proceeds as follows:

- We first evaluate whether Ferry can outperform other tools in code coverage, with 13 real-world programs and the comprehensive test suite Google FuzzBench [56]. Our evaluation shows that Ferry covers 38%/42% more basic blocks and 42%/47% more branches than two widely-used symbolic execution engines KLEE and angr. Given the same input seeds, Ferry with SSE outperforms three popular fuzzers (AFL, Angora and QSYM) in block and branch coverage. What's more, 15% of its basic blocks have never been reached by any fuzzer. Our further examination of the results shows that Ferry achieves a higher code coverage because it is able to explore deeper program logic than the counterparts with inferred state-describing variables.
- Apart from achieving high code coverage, this paper focuses on exploring "deep" program logic depending on program states. We have presented the prevalence of state-dependent branches in Section 2.2. In this section, we further reveal the importance of exploring deeper program logic by collecting the program-state-dependent vulnerabilities in the CVE database and evaluating whether Ferry and other tools can reproduce them. We further construct a test suite River, which facilitates our evaluation of the extent to which a tool is able to explore state-dependent branches. River consists of six programs with 160 manually-inserted vulnerabilities which can only be triggered in certain program states. Compared with two symbolic executors and three fuzzers, Ferry manages to locate over 40% more real-world vulnerabilities and 2.4 times as many manually-inserted vulnerabilities.
- To understand the performance baseline of Ferry, we use another comprehensive dataset LAVA-M [31], in which the vulnerabilities are state-irrelevant. In the worst case, Ferry performs almost the same as angr with BFS strategy except that it avoids exploring some duplicated paths. We further discuss the limitations of Ferry in Section 6.7.

Implementation. Ferry is built upon the angr framework [19] with 5070 lines of Python code. We use Z3 [50] to solve SMT queries and compare program states. Unicorn engine [4] is applied to accelerate the emulation of concrete file contents in shortcut symbolic execution.

Vulnerability Detection. In symbolic execution, how to drive the execution to reach a vulnerability-carrying branch and how to trigger/exploit a vulnerability (after the corresponding security vulnerability is reached through symbolic execution) are two separate issues. Since the second issue has not been well-addressed in the field of symbolic execution, this work has to leave it out of the scope. It should be noted that sanitizers [27], which work well together with software fuzzing, cannot be used to resolve the second issue. A native combination of symbolic execution engines and sanitizers cause severe path explosion because sanitizers introduce extra checks (i.e., conditional statements) on input data.

Experiment Setup. All our evaluations process on a Ubuntu 16.04 server, with a single 3.9GHz Intel i5-8600K CPU and 64GB of RAM. Ferry and most of the compared tools are single-threaded, except for QSYM which has a fuzzing process and a symbolic execution process. All tools are given the same timeout of six hours. We repeat the experiments for three times and report the average results. Basically, our program-state-aware symbolic execution technique is based on a BFS exploration strategy to recognize *InputDetermined* and *BranchRelated* variables. In our evaluation, we compare our approach against other symbolic executors under different exploration strategies including BFS and DFS. For KLEE, we further evaluate its unique random path strategy. Input seeds are needed for Ferry with SSE and fuzzers, our seed

files[3] are collected from AFL testcases [54] and FuzzBench. Following the performance recommendation from AFL [61], we exclude the large seeds from FuzzBench (they will also overwhelm the taint analysis of Angora), and limit the number of seeds. We apply the configurations above in all the following experiments.

6.1 Code Coverage

Benchmarks. To evaluate the code coverage of Ferry, we collect two classes of programs. Specifically, we use the comprehensive dataset FuzzBench from Google to evaluate the path exploration ability of Ferry and other tools. FuzzBench is an ever-changing benchmark. At the time of writing, it contains 49 programs [57]. We successfully analyse 20 programs in FuzzBench. The reason why we fail to include the others is two-fold. First, the FuzzBench is designed to be fuzzed by Google OSS-Fuzz [60], and most of our evaluated tools are not officially supported since Google only provides an adaptor for AFL-based tools. We list the reason why non-AFL tools fail to analyze the 25 programs as follow:

- Compiler and linker errors. The Google OSS-Fuzz (powered by libFuzzer [58]) requires customized target programs. Specifically, The programs need to override libFuzzer-provided interfaces like `LLVMFuzzerTestOneInput()` and `LLVMFuzzerInitialize()`, which are not supported by other fuzzers and symbolic executors, causing build failure. We have tried our best to rewrite the programs by providing wrappers for these interfaces and revise build scripts to adapt them to other tools, but sill fail to run 15 of them.
- Environmental modeling limitations. We observe that 10 out of the 25 programs implement complicated logic in their `LLVMFuzzerInitialize()` which cannot be handled by our environmental model. Among them, *curl* driver [21] introduces an extra HTTP 2.0 library [59] to construct HTTP requests, and the construction progress can cause path explosion in symbolic executors; the rest of them either dynamically create temporary files with unmodeled functions like `mkstemp()`, or depend on multithread functionalities which are not support by any of the symbolic executors.

Second, 4 programs fail to be built because FuzzBench does not follow the changes of the upstream codebases. For example, the files in *stb* [47] needed by FuzzBench have already been removed in its latest version and FuzzBench does not update related scripts. We further collect 13 real-world programs, with over 2 million LOCs in total. They are of concern to security researchers [68,73,74,90,96] and cover a variety of categories including processing of image, video/audio, PDF and binary. What's more, many of them, e.g., *libjpeg* [32] and *libtiff* [14], serve as the building blocks for important real-world projects including Chromium, Android and iOS.

0-day Vulnerabilities. Interestingly, during our evaluation, Ferry locates two unreported null-dereference bugs in *libstagefright*. We further analyzed them and found that they were internally fixed by Google [2, 3]. What's more, by feeding the inputs generated by Ferry to the programs instrumented by sanitizers, we find that Ferry found three 0-day vulnerabilities in *jhead*. Specifically, Ferry successfully locates four heap-buffer-overflow vulnerabilities, three of them are 0-day and one has been reported [11] yet not fixed.

Responsible Disclosure. We reported all the three 0-day vulnerabilities to the *jhead* author following responsible disclosure. Specifically, we collected the email address from author's home page [15], and then shared all the details about vulnerabilities including the vulnerable commits, code locations and PoC exploits. Finally, they have all been fixed in the latest version [55].

Table 2 reveals the block coverage and branch coverage of different tools. Noted that 7 programs in FuzzBench explicitly mark KLEE as "unsupported". We further find that KLEE fails to analyze *PoDoFo*, *libav*, *ffmpeg*, *FreeImage*, *mupdf* and *libtiff*, and the reason is the same: it requires compiling the analyzed programs to LLVM Intermediate Representation [40] and then symbolically executes on the LLVM IR, but it fails to support all the LLVM intrinsic functions generated during compilation, thus crashes at the beginning of symbolic execution. Besides, only Ferry and angr manage to analyze *libstagefright*. We analyze it and find that only they have correctly modeled the Android *bionic* [5] library, which is a build dependency of *libstagefright*.

We divide the evaluated tools into two groups, based on whether seed files are a necessary requirement. Specifically, only Ferry with SSE and fuzzers require seed files, while Ferry without SSE and other symbolic executors do not. For fair comparison, we compare their code coverage respectively. That is, in the first group, we compare Ferry with other two symbolic executors. We only keep the best results of angr and KLEE under different searching strategies. KLEE achieves extremely low coverage when analyzing *file*. Our manual check shows that this is due to the fact that KLEE's modeling of filesystem prevents it from loading a list of signatures to recognize file formats. And we exclude this abnormal data during our comparison with KLEE for a fair evaluation. Ferry (without SSE) covers 38%/42% more basic blocks and 42%/47% more branches than KLEE and angr, respectively.

In the second group, we compare Ferry with SSE to three famous fuzzers (we also present the basis code coverage introduced by seed files). We observe that Ferry with SSE performs bad on *jsoncpp*, *libarchive*, *libxml2*, *re2* and *zlib*, meanwhile the hybrid fuzzer QSYM, which is based on AFL, also shows no improvements on them compared to raw AFL. Our investigation reveals that this is because the string matching logic (e.g., regex processing in *re2*) in *jsoncpp*, *libxml2* and *re2*,

[3] Available at https://drive.google.com/file/d/1QPp6n4RNfEPH58tGvYkZy6_Od4QbZiyM/view?usp=sharing

Table 2: Code coverage of Ferry with/without SSE compared with symbolic execution engines (KLEE and angr) and fuzzers (AFL, Angora and QSYM). For symbolic executors with different exploration techniques, we show the best result for each program.

Program	**Ver.**	Block Coverage			Branch Coverage			Block Coverage					Branch Coverage				
		KLEE	angr	**Ferry**	KLEE	angr	**Ferry**	seed	AFL	Angora	QSYM	**Ferry w/. SSE**	seed	AFL	Angora	QSYM	**Ferry w/. SSE**
bloaty [20]	2020-05-25	N/A	2235	2530	N/A	519	756	9850	10466	11375	11402	11280	1387	2025	2234	2242	2435
freetype2 [24]	2017	1525	1725	2215	875	962	1327	4393	8611	7226	10253	9675	2593	6839	5909	8682	8763
harfbuzz [28]	1.3.2	N/A	1525	1732	N/A	765	856	2213	6291	8734	9716	8675	759	3372	4565	4968	4785
jsoncpp [30]	2020-02-13	N/A	318	320	N/A	96	96	1469	2036	1960	1991	1678	128	554	545	555	462
libarchive [12]	2019-05-03	1023	1017	1235	548	543	594	1453	3974	N/A	4037	3778	735	3179	N/A	3263	3125
libhevc [7]	2019-09-06	17645	19230	22322	6436	6846	7344	3659	29627	19591	30564	30872	2096	9324	7119	9382	9410
libhtp [13]	2019-09-14	1213	1210	1432	1125	1123	1334	1201	2084	2090	2062	2435	791	1835	1841	1834	2122
libjpeg-turbo [33]	2017-06-28	972	978	1132	642	645	875	1090	1996	1962	2302	3014	650	1559	1504	1826	2203
libpcap [34]	2020-02-12	N/A	760	772	N/A	465	525	780	1562	1730	1652	1725	235	1148	1325	1270	1518
libpng [36]	1.2.56	258	398	512	247	302	468	588	987	1252	1283	1209	400	760	943	1007	1056
libxml2 [38]	2.9.2	375	412	875	214	246	569	1028	3821	3084	3845	2065	668	3662	3294	5345	1765
Little-CMS [39]	2017-03-21	728	826	872	413	525	571	796	1853	2450	2417	2630	395	1107	1658	1778	1834
mbedtls [41]	2020-02-11	N/A	509	514	N/A	297	312	2318	2625	2520	2622	2854	1375	1586	1495	1582	1725
muparser [42]	2020-08-19	932	933	1231	681	684	912	1771	2702	2407	2687	3124	254	915	692	907	1025
openh264 [16]	2019-10-22	7112	7264	9312	5623	5812	7412	2559	11462	6106	10137	13245	1560	8974	7021	8965	10023
openthread [44]	2019-12-23	N/A	684	765	N/A	257	334	3885	4708	4812	4752	4620	1190	1748	1770	1757	1782
re2 [46]	2014-12-09	873	871	1346	327	327	548	2554	6336	5710	6396	4280	648	2930	2576	2932	2230
vorbis [51]	2017-12-11	N/A	625	642	N/A	458	472	888	1251	1190	1236	1642	635	1048	990	1059	1246
woff2 [52]	2016-05-06	1679	1875	2129	746	761	965	4032	4934	5183	4816	5431	824	1567	1575	1634	1725
zlib [53]	2020-05-06	249	258	265	142	149	184	132	527	433	522	318	70	416	317	413	267
libstagefright [37]	5.1.1	N/A	1043	2196	N/A	965	1765	1432	N/A	N/A	N/A	2577	1025	N/A	N/A	N/A	2232
libjpeg [32]	9c	458	386	695	213	175	363	1123	2016	2246	2289	2325	674	1677	1782	1790	1810
PoDoFo [45]	0.9.6	N/A	3862	4131	N/A	1864	2235	2570	5024	5436	5535	5726	1256	2234	2578	2630	2789
giflib [17]	5.2.1	285	343	446	179	212	235	852	1217	1375	1340	1324	672	853	969	978	1025
libav [6]	12.3	N/A	360	2297	N/A	297	1678	2575	5875	6233	6425	6472	2132	5325	5734	5864	6078
ffmpeg [23]	4.2.1	N/A	1057	1288	N/A	654	843	5685	7124	7756	8132	8079	3764	5513	5843	6012	6287
FreeImage [10]	3.18.0	N/A	629	2075	N/A	426	1297	4632	8023	8225	8234	8170	3347	5512	5615	5613	5724
ImageMagick [29]	7.0.1	787	817	848	514	532	617	4232	4758	5230	5133	5432	3756	4231	4482	4475	4875
jhead [8]	3.04	96	140	181	67	97	121	162	262	642	593	942	120	218	532	445	812
mupdf [43]	1.16.1	N/A	454	536	N/A	298	324	4672	5032	5335	5420	5624	1875	2021	2229	2394	2576
libtiff [14]	4.1.0	N/A	192	537	N/A	114	347	2124	3327	3576	3276	3234	1827	3016	3121	2748	2835
file [9]	5.37	*69	373	802	*40	375	725	1279	1525	1920	1837	1842	1165	1462	1899	1802	1825
readelf [26]	2.33.1	708	377	844	523	213	578	5438	7112	8125	7223	7429	3976	5602	7003	5642	5835

* We exclude the abnormal result of *file* when compared with KLEE.

and compression logic in *libarchive* and *zlib* create complicated path constraints which block the solver and cause state explosion in Ferry. We further discuss the limitations of Ferry in Section 6.7. Given the same seed files, Ferry shows better branch coverage on most of the benchmarks. Apart from the five program above, Ferry covers 18%/21%/8% more basic blocks and 22%/21%/8% more branches than AFL/Angora/QSYM, respectively. What's more, the distribution of Ferry's block coverage is significantly different from others. Specifically, 15% covered blocks by Ferry have never been reached by any fuzzer. We believe such a result indicates the capability of Ferry to cover hard-to-reach code.

Our manual inspection further reveals that Ferry is able to reach deeper logic of these programs while the others cannot. For example, when analyzing *ffmpeg*, only Ferry manages to explore the code in the underlying video decoder, and angr gets stuck in exploring repeatedly visited states (i.e., angr repeats the same instructions hundreds of times, and all the executions of these instructions are in the same valid state recognized by Ferry).

6.2 Reproducibility of Program-state-dependent Vulnerabilities

In this section, we evaluate whether Ferry can locate more program-state-dependent vulnerabilities than previous works.

Vulnerability Collection. A vulnerability is considered program-state-dependent if the path to it involves at least one state-dependent branch. We collect the program-state-dependent vulnerabilities in three steps. First, we determine the version of each program in concern to reduce the search scope. We set their versions to the ones with the most reported vulnerabilities in CVE database. Then we gather the information about each vulnerability, including its code location and the input sequences to trigger it. Specifically, we collect the vulnerabilities details via three ways: (1) we follow the reference links of each vulnerability in the CVE database, (2) we search them in Exploit Database [22], which provides the proof-of-concept exploits of many vulnerabilities, and (3) we try to find the patches that fix the vulnerabilities and learn from them. At last, we manually filter the ones which do not depend on program states. As depicted in Table 3, we finally collect 15 vulnerabilities from CVE database.

As mentioned above, symbolic executors cannot automatically trigger the vulnerabilities. Thus, we manually replace these vulnerabilities with `assert(0)` guarded by required triggering conditions so we can be notified as soon as the executors reach them. These vulnerabilities cover various vulnerability types, including heap and stack overflow, integer overflow, double-free, use-after-free, divided-by-zero and uninitialized pointer reference.

The results are shown in Table 3. Among the successful analyses of these tools, Ferry manages to reproduce 8 of the 15 vulnerabilities without enabling shortcut symbolic execution. After the shortcut symbolic execution is enabled, Ferry can locate all the vulnerabilities. We explain the reasons why other tools fail to locate the vulnerabilities as follow. For symbolic executors, the main reason is path explosion. Our manual check reveals that both KLEE and angr suffer from severe path explosions. For example, angr forks over 10 times more execution paths than Ferry within the six hours when

Table 3: Vulnerability reproducibility of various tools. We compare related approaches with Ferry with or without shortcut symbolic execution (SSE).

Program	CVE-ID	KLEE	angr	Ferry	AFL	Angora	QSYM	Ferry w/. SSE
libstagefright 5.1.1	2015-3827	N/A	No	Yes	N/A	N/A	N/A	Yes
	2015-3828	N/A	No	Yes	N/A	N/A	N/A	Yes
	2015-3829	N/A	No	Yes	N/A	N/A	N/A	Yes
	2015-3864	N/A	No	Yes	N/A	N/A	N/A	Yes
	2016-3830	N/A	No	No	N/A	N/A	N/A	Yes
ffmpeg 3.1.3	2016-2213	N/A	No	No	No	No	No	Yes
	2016-2329	N/A	No	No	No	No	No	Yes
	2016-10190	N/A	No	Yes	Yes	Yes	Yes	Yes
ImageMagick 7.0.0	2016-7532	No	No	No	Yes	Yes	Yes	Yes
	2016-7799	No	No	No	Yes	Yes	Yes	Yes
PoDoFo 0.9.5	2017-8378	N/A	No	No	No	No	No	Yes
	2018-5783	N/A	No	No	No	No	No	Yes
	2018-6352	N/A	Yes	Yes	Yes	Yes	Yes	Yes
	2018-8000	N/A	Yes	Yes	Yes	Yes	Yes	Yes
	2018-12982	N/A	Yes	Yes	Yes	Yes	Yes	Yes

analyzing *libstagefright*/*ffmpeg*/*ImageMagick* with BFS strategy. For fuzzers, whether them can reach the vulnerabilities is greatly affect by seed files. For example, no fuzzers are able to reach the CVE-2016-2213 in *ffmpeg* since it can only be triggered by video files containing special type of JPEG 2000 [1] payload. Unfortunately, such rare payload is not presented in any AFL testcase or FuzzBench seed, and hard to be efficiently generated by fuzzers. In contrast, Ferry with SSE only uses given seeds to avoid path explosion in early-stage video processing logic, and successful reaches the vulnerability after it makes corresponding input partitions symbolic.

6.3 "Deep" Program State Exploration

We propose a test suite River to facilitate us to evaluate the extent to which a dynamic analysis tool can explore state-dependent branches. River is different from other datasets, such as LAVA-M, in that it meant to evaluate a tool's ability to explore "deep" states, which is orthogonal to the block coverage criterion concerned by others. It bears the following characteristics: (1) all the inserted vulnerabilities are program-state-dependent, which means they can only be triggered by certain input sequences; (2) vulnerabilities are distributed over various *state depth*, i.e., the least number of state-dependent branching decisions it takes to reach the vulnerability. The ones with greater state depth have more constraints on input sequences, thus are harder to reach. We manually annotated all the inserted vulnerabilities and recorded their state depth information. Because River is meant to evaluate a tool's ability to explore "deep" states, the corresponding metric must take state depth into consideration. Accordingly, instead of measuring the total number of program states that have been explored, we argue that the most appropriate **metric** should focus on measuring how many "deep" states have been explored. In order to implement this idea, we let state depth be a key attribute of each inserted vulnerability. In this way, the corresponding (*vulnerability ID*, *state depth*) pair tells us not only that a new program state has been explored, but also whether the state is a "deep" state or not. It should be noticed that whether the metric of (*vulnerability ID*, *state depth*) pair can systematically measure a tool's ability to explore deep states largely depends on whether there is a vulnerability inserted at a representative set of reachable deep states.

Test Suite Construction. We construct the benchmark in two steps. First, we use program-state-aware symbolic execution to analyze the programs for six hours and record the necessary information for vulnerability injection including the code locations of each state transition, i.e., SDVO addresses, and path constraints to reach them. Second, we manually insert vulnerabilities after each SDVO. Note that we skip the SDVOs whose path constraints are too complicated for the human program analysts to understand. Each inserted vulnerability is a call to `assert(0 && state_depth && vul_id)` guarded by path constraints so it can be unconditionally exploited. The `vul_id` is unique and serves as the index to get detailed vulnerability information.

Program State Exploration Ability Evaluation. We use River to evaluate all the symbolic executors and fuzzers. We present the detailed vulnerability reproduction results in Appendix A and the distribution of inserted bugs reached by different tools over depth in Figure 7. Our results show that existing symbolic executors and fuzzers can only locate bugs with shallow depth. Specifically, when the depth is less than 20, the counterparts manage to locate 46% (51 out of 110) of the injected bugs. While only angr with DFS and KLEE with RP are able to locate 5, respectively, out of the 17 vulnerabilities with a depth greater than 40.

6.4 Accuracy of Inferred SDVO instructions.

To the best of our knowledge, there is no ground truth about the SDVO instructions of the given programs, preventing us from comprehensively evaluating the precision of SDVO instruction inference. Instead, we checked the SDVO instructions of *libjpeg* as an example. After 10 hours' code audit, we pointed out 49 SDVO instructions. Meanwhile, Ferry recognized 88 SDVO instructions, covering all the manually labeled ones. Our further manual verification confirmed that all the rest 39 recognized instructions are truly SDVO ones. Our code auditing omitted them because their code resides in the deep logic of the program, after multiple function calls and variable assignments, which hinders the tracking of inputs with human effort.

6.5 A Case Study

CVE-2016-3830 in libstagefright. CVE-2016-3830 is a reported vulnerability of `libstagefright`. Among all the evaluated symbolic execution engines, only Ferry with shortcut symbolic execution can automatically locate this vulnerability. Our manual inspection reveals the difficulties to uncover it:

- The vulnerability occurs in the audio decoder of AAC format, which is "protected" by at least two earlier state machines. That is, a first state machine which initializes the audio decoder and extracts the audio header and tracks, and the second state machine which leverages the audio header

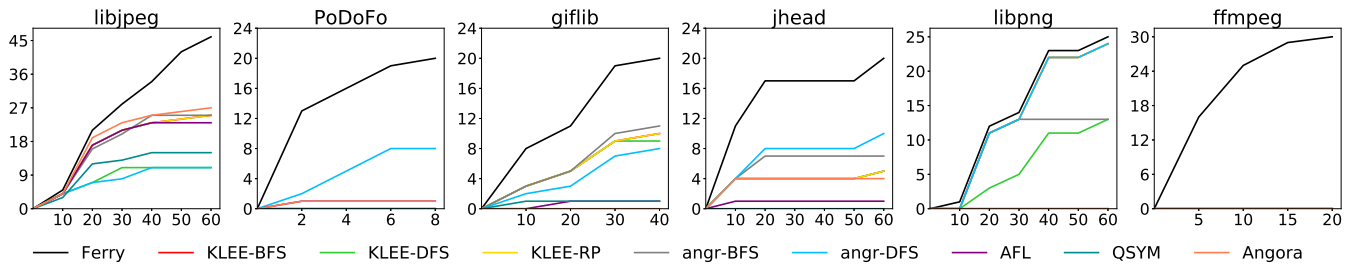


Figure 7: The cumulative distribution of inserted bugs reached by different tools over depth. The x-axis shows the state depth of injected bugs and the y-axis shows the number of bugs reached by each tools.

to configure the decoder. The vulnerability resides in the third state machine which decodes the audio tracks. Our experiment shows that angr suffers path explosion in the first state machine, and if shortcut symbolic execution is disabled, Ferry suffers severe path explosion in the second state machine.

- The vulnerability can only be exploited in a deep program state of the third state machine. Specifically, in a state after a mismatch between an expected payload size `adtsHeaderSize` and a calculated result stored in `aac_frame_length`. To measure the efficiency of state-of-the-art symbolic execution, we further applied this case to angr, by manually assigning the entry point of the third state machine to it. Our evaluation reveals that angr still fails to locate this vulnerability.

6.6 Performance Baseline

In the section, we try to find out the performance baseline of Ferry when exploring code logic with no state-dependent branches. We achieve this with dataset LAVA-M, since LAVA only inserts vulnerabilities at the locations that input bytes "do not determine control flow" (i.e., the control flow statements involved in the paths arriving at these locations are independent of the input bytes). That is, the execution paths towards these vulnerabilities involve no state-dependent branch. We present the detailed results in Appendix B. The results show that in the worst case, Ferry performs almost the same as angr with BFS strategy except that it avoids exploring some duplicated paths.

6.7 Discussion

Our experiments show that Ferry provides an effective symbolic execution engine for exploring state-dependent branches, and achieves a first step towards solving the challenging problem for symbolic execution: how to handle the programs with state-dependent branches. In this section, we discuss the limitations of Ferry and under what circumstances Ferry may fail.

State Explosion. Table 2 shows that the efficiency of Ferry is limited on certain program logic (e.g., string searching and compression algorithm). On one hand, they are common challenges for symbolic executors since such logic will generate complicated path constraints which can overwhelm the constraint solver. On the other hand, such program logic causes state explosion. For example, compression programs have huge internal state space, and any changes to an input byte will lead to different program states and change the way in which all subsequent inputs are processed. In such cases, Ferry suffers state explosion (which further causes path explosion). Our optimizations, which help alleviate the state explosion, cannot completely solve this problem.

As a first attempt to systematically tackle programs with state-dependent branches for symbolic execution, our technique works for the programs that bear the similar intrinsic characteristic explained in Section 2.3. However, it is currently not yet a general-purpose approach. A major challenge that still hinders the symbolic execution engines is how to identify state-describing variables in an arbitrary program. We leave this issue as our future work on Ferry.

7 Related Work

This section discusses two classes of dynamic analysis approaches: first, other works on symbolic execution; then, the fuzzing tests known as alternatives to symbolic execution engines in bug finding. We further discuss the related works which explore the state space in real-world programs, operating systems and network protocols and explain their differences with this paper.

7.1 Symbolic execution

Ferry is the first symbolic execution engine that systematically handles state-dependent branches of programs. Since the appearance of the first symbolic execution engine, several decades have passed with tens of symbolic execution engines proposed [70]. Based on their ways of achieving high code coverage, they can be characterized into three categories.

Mixing concrete and symbolic execution. Replacing part of symbolic values with concrete ones [64, 68, 69, 71, 75, 90] can significantly alleviate the path explosion, thus improving performance. For example, S2E [75] proposes an automatic bidirectional symbolic-concrete state conversion that enables execution to seamlessly and correctly weave back and forth between symbolic and concrete mode. In this paper, we propose a shortcut symbolic execution which also mixes a preceding concrete execution with a following symbolic one. However, unlike traditional concolic testing which randomly reverts

branch taking, our core problem here is how to automatically determine the boundaries of input partitions and explore the corresponding state-dependent branch groups. This problem is solved based on our unique observations.

Compositional execution. Compositional testing analyzes elementary units (i.e., a method or a procedure) in the program separately, and stores the analysis results in summaries encoding the input-output transformation of the units. It is first proposed by Dart [80], and several optimizations were proposed to improve compositional testing [79, 88]. Generally, a fundamental assumption of compositional execution is the loose coupling between functions, thus they can improve scalability by directly checking elementary units, rather than whole programs. However, in programs with state-dependent branches, the behavior of each elementary unit may vary, depending on the runtime program state.

Selective symbolic execution. Selective symbolic execution (also referred to as guided symbolic execution) [66–68, 83, 91, 94] applies heuristic policies to guide the selection of execution paths to follow, expecting to execute paths that are most likely to cover new code in the immediate future. For example, the heuristic of KLEE [68] is a combination of the minimum distance to an uncovered instruction, the call stack of the process, and whether the process has recently covered new code. Li *et al.* [84] use subpath program spectra to systematically approximate full path information, and guide symbolic execution to less traveled paths; Chopper [92], on the other hand, allows users to specify uninteresting parts of the code to exclude during the analysis. Ferry also applies a selective symbolic execution, by leveraging the insight that reached program states should not be explored again. While instead if prioritizing certain paths, Ferry automatically prunes redundant paths that explore reached program states.

7.2 Fuzzing

Fuzzing (or fuzz testing) is an automated testing technique that provides invalid, unexpected, or random data as inputs to a program, and monitors the program for its unexpected behaviors. Mutation-based fuzzers [65, 72] generate inputs by modifying (referred to as mutating) the provided reference inputs (seeds). They assume that the unexplored program behavior can be triggered by simple mutation of program inputs. However, such an assumption does not hold in programs with many state-dependent branches, since a mutated program input is difficult to trigger a serial of state-transitions. Generation-based fuzzers [93, 95] usually generate inputs based on a provided input model (or format). They usually require human efforts to specify the data template. Even worse, it is challenging to infer a program's states. Thus, modeling the inputs for a program with state-dependent branches could be time-consuming.

Besides, guided fuzzers [18, 73, 85, 89, 91] apply heuristics to select test cases that are likely to reach unexplored code. For example, AFL [18] uses program instrumentation to understand which inputs explore new program branches, and keeps these inputs as seeds for further mutation. Several optimizations [73, 85, 89] add new heuristics to increase branch coverage by solving various path constraints. Besides, Driller [91] relies on selective concolic execution to generate seeds for fuzzing. Although these tools are capable to handle single input checks, state machines in the real world usually contain a series of complicated checks, which can overwhelm them. Besides, they focus on exploring new paths, and are unaware of the program states discussed in our paper. As a result, their heuristics cannot drive them to explore program states efficiently.

7.3 Program and System State Exploration

Apart from Ferry, some related works also try to explore the state space in real-world programs, operating systems and network protocols. We explain their differences with this paper as follows. IJON [62] explores the deep state spaces of programs. It enables human experts to provide feedbacks to fuzzers to help explore state machines, while our work automatically infers the state-describing variables and requires no human efforts. The real-world programs analyzed in our work contain tens or even hundreds of state-describing variables, which can overwhelm human analysts.

In terms of exploring the state space in operating systems, state-aware kernel fuzzing techniques [63, 82, 86] monitor the states of operating systems during the exploration of kernel logic. Unlike program states concerned in this paper, operating system states are described by well-defined metadata, and state transitions happen when well-defined events (e.g., a syscall is invoked, an interrupt happens) happen. Accordingly, neither identifying state-describing variables nor recognizing runtime states is a major challenge in state-aware kernel fuzzing. In contrast, they are the major challenges in program-state-aware symbolic execution as shown in Section 2.1.

The stateful network protocols are another class of promising analysis targets since the outbreak of several severe vulnerabilities like Heartbleed [49]. The input sequences in network protocol are divided into individual messages between client and server. When a message is processed by server, a response with status code, which explicitly shows the server state, is replied. Current stateful protocol fuzzing works utilize these prior knowledge about network protocols (e.g., message structure and status code) when analyzing them. AFLNet [87], for example, uses protocol-specific information of message structure to extract individual messages, reads status code to determine server's state and reorders messages with heuristics to explore protocol states. And MACE [76] relies on user-provided state-machine model abstraction, and uses the information to guide concolic execution. In contrast, Ferry assumes no prior knowledge about analyzed programs and input structure. From Ferry's perspective, the status code bears no difference from other state-describing variables, and it tries to automatically recognize all the variables that describe

program state.

8 Conclusion

In this work, we make the first attempt to achieve program-state-aware symbolic execution on an important class of real-world programs where date dependency exists between input data and state-describing variables. We propose Ferry, a new symbolic execution engine which focuses on real-world programs with state-dependent branches. We demonstrate the effectiveness of our approach by applying Ferry to two comprehensive test suites, 13 real-world programs and 15 reported vulnerabilities. The experiment results show that Ferry clearly outperforms the existing approaches. Our evaluation also reveals that Ferry covers more code (basic blocks) and branches that other techniques and locates 2.4 times as many program-state-dependent vulnerabilities as state-of-the-art symbolic executors and fuzzers. Our research provides a successful experience for the following works on achieving general-purpose program-state-aware symbolic execution.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U1736208, U1836210, U1836213, 62172104, 61972099, 61902374), Natural Science Foundation of Shanghai (19ZR1404800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] JPEG 2000. https://en.wikipedia.org/wiki/JPEG_2000, 1997.

[2] MPEG4Extractor: more NULL derefernce fixes in parseChunk. https://android.googlesource.com/platform/frameworks/av/+/202fbed96db40ec5fb43d633fc97601a15a6dd7a, 2014.

[3] MPEG4Extractor: null check in MPEG4Source::parseChunk. https://android.googlesource.com/platform/frameworks/av/+/1391f933b49cfb56da9aa63f723de83b076cf888, 2014.

[4] Unicorn: The ultimate CPU emulator. https://www.unicorn-engine.org/, 2017.

[5] Bionic. https://android.googlesource.com/platform/bionic/, 2018.

[6] libav. https://libav.org/, 2018.

[7] android/platform/external/libhevc. https://android.googlesource.com/platform/external/libhevc, 2019.

[8] Exif Jpeg header manipulation tool. https://www.sentex.ca/~mwandel/jhead/, 2019.

[9] file. https://www.astron.com/pub/file/, 2019.

[10] FreeImage. http://freeimage.sourceforge.net/, 2019.

[11] heap-buffer-overflow detected in function process_dqt. https://bugs.launchpad.net/ubuntu/+source/jhead/+bug/1857521, 2019.

[12] libarchive. https://github.com/libarchive/libarchive, 2019.

[13] LibHTP. https://github.com/OISF/libhtp, 2019.

[14] LibTIFF - TIFF Library and Utilities. http://www.libtiff.org/, 2019.

[15] Matthias Wandel's (old) Home page. https://www.sentex.ca/~mwandel/index.html, 2019.

[16] OpenH264. https://github.com/cisco/openh264, 2019.

[17] The GIFLIB project. https://http://giflib.sourceforge.net/, 2019.

[18] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2020.

[19] angr. https://github.com/angr/angr, 2020.

[20] Bloaty McBloatface: a size profiler for binaries. https://github.com/google/bloaty, 2020.

[21] curl_curl_fuzzer_http. https://github.com/google/fuzzbench/tree/master/benchmarks/curl_curl_fuzzer_http, 2020.

[22] Exploit Database. https://www.exploit-db.com/, 2020.

[23] FFmpeg. https://www.ffmpeg.org/, 2020.

[24] Freetype2. git://git.sv.nongnu.org/freetype/freetype2.git, 2020.

[25] Fuzzing. https://en.wikipedia.org/wiki/Fuzzing, 2020.

[26] GNU Binutils. https://www.gnu.org/software/binutils/, 2020.

[27] Google Sanitizers. https://github.com/google/sanitizers, 2020.

[28] HarfBuzz. https://github.com/behdad/harfbuzz, 2020.

[29] ImageMagick. https://www.imagemagick.org/script/index.php, 2020.

[30] JsonCpp. https://github.com/open-source-parsers/jsoncpp, 2020.

[31] LAVA: Large Scale Automated Vulnerability Addition. https://github.com/panda-re/lava, 2020.

[32] libjpeg. https://www.ijg.org/, 2020.

[33] libjpeg-turbo. https://github.com/libjpeg-turbo/libjpeg-turbo, 2020.

[34] LIBPCAP 1.x.y by The Tcpdump Group. https://github.com/the-tcpdump-group/libpcap, 2020.

[35] libpng. http://www.libpng.org/pub/png/libpng.html, 2020.

[36] libpng. https://downloads.sourceforge.net/project/libpng/libpng12/older-releases/1.2.56/libpng-1.2.56.tar.gz, 2020.

[37] libstagefright. https://android.googlesource.com/platform/frameworks/av/+/master/media/libstagefright, 2020.

[38] libxml2. https://gitlab.gnome.org/GNOME/libxml2, 2020.

[39] Little CMS. https://github.com/mm2/Little-CMS, 2020.

[40] LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html, 2020.

[41] Mbed TLS. https://github.com/ARMmbed/mbedtls, 2020.

[42] muparser - Fast Math Parser 2.3.3. https://github.com/beltoforion/muparser, 2020.

[43] MuPDF. https://www.mupdf.com/, 2020.

[44] OpenThread. https://github.com/openthread/openthread, 2020.

[45] PoDoFo. http://podofo.sourceforge.net/, 2020.

[46] RE2. https://github.com/google/re2, 2020.

[47] stb_stbi_read_fuzzer. https://github.com/google/fuzzbench/tree/master/benchmarks/stb_stbi_read_fuzzer, 2020.

[48] Symbolic execution. https://en.wikipedia.org/wiki/Symbolic_execution, 2020.

[49] The Heartbleed Bug. https://heartbleed.com/, 2020.

[50] The Z3 Theorem Prover. https://github.com/Z3Prover/z3, 2020.

[51] Vorbis. https://github.com/xiph/vorbis, 2020.

[52] woff2. https://github.com/google/woff2, 2020.

[53] ZLIB DATA COMPRESSION LIBRARY. https://github.com/madler/zlib, 2020.

[54] AFL starting test cases. https://github.com/google/AFL/tree/master/testcases, 2021.

[55] Fix issue 24. jpgqguess could read a few bytes past what was allocated. https://github.com/Matthias-Wandel/jhead/commit/b711024c667382241fde92e201363689c60adbd2, 2021.

[56] FuzzBench: Fuzzer Benchmarking As a Service. https://github.com/google/fuzzbench, 2021.

[57] fuzzbench/benchmarks. https://github.com/google/fuzzbench/tree/master/benchmarks, 2021.

[58] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2021.

[59] nghttp2 - HTTP/2 C Library. https://github.com/nghttp2/nghttp2, 2021.

[60] OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2021.

[61] Tips for performance optimization. https://github.com/google/AFL/blob/master/docs/perf_tips.txt, 2021.

[62] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (Oakland 2020)*, 2020.

[63] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *The Network and Distributed System Security Symposium 2019 (NDSS 2019)*, 2019.

[64] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, 2014.

[65] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, 2016.

[66] Suhabe Bugrara and Dawson Engler. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC 2013)*, 2013.

[67] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 2008.

[68] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[69] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

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

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

[72] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland 2015)*, 2015.

[73] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland 2018)*, 2018.

[74] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019)*, 2019.

[75] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, 2011.

[76] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *USENIX Security Symposium*, volume 139, 2011.

[77] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, 1984.

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

[79] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, 2007.

[80] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005.

[81] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[82] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *The Network and Distributed System Security Symposium 2020 (NDSS 2020)*, 2020.

[83] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*, 2012.

[84] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. *ACM SigPlan Notices*, 48(10):19–32, 2013.

[85] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*, 2017.

[86] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 2018)*, 2018.

[87] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[88] David Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security 2015)*, 2015.

[89] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *The Network and Distributed System Security Symposium 2017 (NDSS 2017)*, 2017.

[90] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *37th IEEE Symposium on Security and Privacy (Oakland 2016)*, 2016.

[91] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *The Network and Distributed System Security Symposium 2016 (NDSS 2016)*, 2016.

[92] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[93] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland 2017)*, 2017.

[94] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*, 2013.

[95] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, 2017.

[96] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 2018)*, 2018.

[97] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

A Vulnerability Coverage on River

We use River to evalute two symbolic executors KLEE and angr, and three fuzzers AFL, Angora and QSYM. The results are shown in Table 4. Since Ferry shares the techniques to construct the benchmark, it is not surprising that it is able to locate all the inserted vulnerabilities. We observe that QSYM performs bad on River, and our manual investigation shows that this is due to one specific optimization of QSYM, that is, "QSYM attempts to detect repetitive basic blocks and then prunes them from symbolic execution", which makes it ignore many branches in input-handling loops. Among the evaluated tools, KLEE finds the most vulnerabilities (41%) and the symbolic executors perform better than fuzzers because their exploration techniques (especially BFS) and constraint solving abilities help them locate more shallow vulnerabilities.

B Reproducibility of LAVA-M Vulnerabilities

The LAVA-M dataset contains a total number of 2,265 automatically inserted vulnerabilities. Instead of inserting real vulnerabilities such as out-of-bound memory accesses, the "vulnerabilities" in LAVA-M are actually invocations of the `dprintf()` function, so it requires no extra efforts as we have previously discussed to trigger the vulnerabilities. We set the

Table 4: Vulnerability coverage of Ferry, symbolic execution engines and fuzzers. #vul and %vul stands for the number and percentage of reached vulnerabilities, respectively.

Program	**Ver.**	Ferry		KLEE-BFS		KLEE-DFS		KLEE-RP		angr-BFS		angr-DFS		AFL		QSYM		Angora	
		#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul
libjpeg	9c	46	100%	23	50%	11	24%	27	59%	25	54%	11	24%	23	50%	15	33%	27	59%
PoDoFo	0.9.6	20	100%	N/A	-	N/A	-	N/A	-	0	0%	8	40%	1	5%	0	0%	1	5%
giflib	5.2.1	20	100%	10	50%	8	40%	10	50%	11	55%	8	40%	1	5%	1	5%	10	0%
jhead	3.04	20	100%	5	25%	5	25%	5	25%	7	35%	10	50%	1	5%	0	0%	4	20%
libpng	1.6.37	25	100%	24	96%	12	48%	24	96%	13	52%	15	60%	0	0%	0	0%	0	0%
ffmpeg	4.2.1	29	100%	N/A	-	N/A	-	N/A	-	0	0%	0	0%	0	0%	0	0%	0	0%
Total	-	160	100%	62	39%	36	23%	66	41%	56	35%	52	33%	26	16%	16	10%	32	20%

Table 5: Number of vulnerabilities reported by Ferry, symbolic execution engines and fuzzers in LAVA-M.

Program	**Total**	Ferry	FUZZER*	SES*	KLEE-BFS	KLEE-DFS	KLEE-RP	angr-BFS	angr-DFS	AFL	QSYM	Angora
uniq	28	0	7	0	0	0	0	0	20	0	29	29
base64	44	44	7	9	44	42	44	44	39	0	48	48
md5sum	57	6	2	0	0	0	0	6	9	0	61	57
who	2136	311	0	18	321	324	324	311	311	0	1,225	1,541

* In the original paper of LAVA [31], the authors have evaluated unspecified coverage-guided FUZZER and unspecified symbolic execution engine SES. Since we use the same timeout and seed files as LAVA, we directly quote its results; the results are also directly quoted in two other works [63, 73].

same timeout (i.e., six hours) for all the tools, and we use the input seed files provided by LAVA-M. We present the detailed results in Table 5.

Since LAVA only inserts vulnerabilities at the locations that input bytes "do not determine control flow" (i.e., the control flow statements involved in the paths arriving at these locations are independent of the input bytes), the execution paths towards these vulnerabilities involve no state-dependent branch. This is significantly different from the River test suite in this paper, and it can explain why Ferry performs almost the same as angr with BFS strategy except that it avoids exploring some duplicated paths. Symbolic executors achieve a good coverage in *base64* and *who* because they manage to solve the constraints of the conditional statements that guard the inserted vulnerabilities, and fail in *md5sum* and *uniq* due to path explosion and complicated constraints introduced by string comparisons. We observe that angr with DFS strategy provides a different result on *uniq* compared with other symbolic executors. Our manual investigation shows that this is mainly because of the implementation of the DFS strategy in angr: angr combines DFS with random path selection by randomly choosing the next execution path after the previous one is finished instead of always taking the longest one, thus avoiding to get stuck in the string comparison loops of *uniq*.

In general, QSYM and Angora perform the best and are able to report vulnerabilities not listed by LAVA. AFL, on the other hand, only generates invalid inputs with random mutation and fails to find any vulnerability.