SandPuppy: Deep-State Fuzzing Guided by Automatic Detection of State-Representative Variables

Vivin Paliath, Erik Trickel, Tiffany Bao,
Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili

Arizona State University
{vivin, etrickel, tbao, fishw, doupe, yans}@asu.edu

Abstract. Current state-of-the-art automated fuzzing approaches cannot explore deep program-states without human assistance. Recently, IJON allowed humans to provide code-annotations on the target to expose program state to the fuzzer. However, this requires a human to read, understand, and annotate the program source, which limits scalability and applicability.
In this paper we introduce SANDPUPPY, a technique that automatically identifies potential state-representative variables and applies IJON-style instrumentation to expose corresponding state to the fuzzer. To identify these variables and their semantics, SANDPUPPY collects runtime variable-value traces from an initial fuzzing run and analyzes them along with the program source to instrument the program and expose internal state to the fuzzer. This process repeats and identifies additional variables, allowing the exploration of deeper states.
We evaluated SANDPUPPY against synthetic and real-world targets representing various fuzzing-challenges. The results show that SANDPUPPY can automatically solve problems that purely coverage-based approaches cannot solve without assistance from human experts. For example, SANDPUPPY can automatically play and solve levels of *Super Mario Bros*, maze programs, and identify complex, deep states in real-world targets such as LIBTPMS, resulting from combinations of previously identified states. Evaluating SANDPUPPY on real-world targets such as JSONCPP, LIBTPMS, PCAPPLUSPLUS, and READELF demonstrates that SANDPUPPY generally improves coverage compared to AFL, AFL++, LAFINTEL, REDQUEEN, SGFUZZ, and FERRY. In all, SANDPUPPY identified ten unknown vulnerabilities in PCAPPLUSPLUS and one unknown vulnerability in DMG2IMG.

1 Introduction

In recent years, fuzz testing, a dynamic-analysis technique that attempts to trigger bugs with pseudo-random input, demonstrated its effectiveness by uncovering a large number of vulnerabilities in diverse software. Hence, fuzzing received significant interest in both academia [4, 10, 32, 35, 39, 45, 3] and industry [46, 7, 22, 1].

Modern fuzzing research has predominantly focused on maximizing *code coverage* — such as basic block coverage and branch/edge coverage — based on the notion that covering more code triggers more vulnerabilities. This is usually done by satisfying previously-failing program branch conditions, whether by solving them using symbolic techniques [39, 45], identifying specific parts of program input that should

be stochastically mutated to increase the chance of satisfying conditions [4, 10], or modifying the conditions themselves [32]. As a result of these interventions, code coverage is increased and additional bugs are found.

Interestingly, increased code coverage is not always correlated with more discovered vulnerabilities: Shoshitaishvili et al. showed that fuzzers could use human-assisted input mutation to find more vulnerabilities—*while triggering strictly less code*—than standard (automated) code-coverage guided fuzzers [38]. One explanation is that not all code coverage is equal [41], and coverage alone does not imply good exploration of a program's *state space*. IJON furthered this concept using manual source-code annotations to extend traditional code coverage feedback with *state-representative variables* [3]. The resulting *state-coverage guided fuzzer* showed promise in—with human help—triggering program states that standard code-coverage guided fuzzers cannot reach.

The next clear step in exploring the impact of state coverage on analysis outcomes is *automatic* reasoning about state-representative variables. Recent work in this area has made progress toward detecting state-representative variables [50, 5], but either use them purely to increase code coverage (rather than *state* coverage) [50] or is only applicable to specific types [5].

In this paper, we propose SANDPUPPY, the first technique to automate the process of identifying and using state-representative variables in fuzzing to increase state coverage, and thus detecting *state-dependent* bugs.

In a reductive sense, every variable used in a program represents some sort of state. However, this does not necessarily imply that variables are equally representative to explore program state-space. Annotating too many variables overwhelms the fuzzer, as it explores unproductive execution paths due to the generation of useless inputs [3]. Therefore, a core challenge for SANDPUPPY then, is to select a minimal subset of state-representing variables from the set of all variables used by a program.

Our solution is inspired by recent observations that a significant number of program states are encoded in a subset of *integer* variables in a program [50, 3, 5]. Building on this, SANDPUPPY identifies candidate integer-variables in a target program and then uses simple, yet novel heuristics, on dynamic runtime value-traces aggregated across multiple executions, to gain a coarse understanding of their semantics. For such variables, SANDPUPPY uses one of several annotation methods (depending on the identified variable semantics) to add these variables to a fuzzer's coverage feedback. SANDPUPPY enables state-coverage guided fuzzing without human effort and without overwhelming the fuzzer with spurious feedback.

Contributions. In summary, our contributions are as follows:

- We design an approach for detecting state-representative variables in a program and for exposing associated state to guide fuzzing towards deep, complex states.
- We implement this approach into a research prototype called SANDPUPPY.
- We evaluate SANDPUPPY against custom targets representing specific fuzzing-challenges as well as real-world programs. SANDPUPPY not only identifies interesting state-representative variables, but the exposed state allows it to explore a deeper and more diverse set of program states and solve certain challenges that purely coverage-based fuzzing techniques cannot. SANDPUPPY also identifies existing and novel vulnerabilities and improves coverage, compared to existing tools.

To support open science, we will open source our research artifacts, including data sets, seeds, and the SandPuppy prototype.

Ethics. We reported all identified vulnerabilities (11 in PcapPlusPlus and 1 in dmg2img). The PcapPlusPlus developers agreed with us and have fixed all vulnerabilities. The dmg2img developers did not respond to our disclosure, and we are continuing to contact them.

2 Background

Fuzzers aim to generate inputs exploring novel program states. Intuitively, the more states explored, the greater the chance to find vulnerabilities. However, lacking a way to directly measure *state*, most fuzzers tend use *code coverage* as a proxy. Coverage-based fuzzers, such as AFL, use a code-coverage metric which approximates the program's state, to identify inputs triggering novel program states. AFL places instrumentation at the edges of the program's control flow graph (CFG) in order to track edge-to-edge transitions. The instrumentation identifies transitions using tuples of identifiers (id_s,id_d), where id_s is the source-block identifier and id_d is the destination-block identifier.

AFL approximates the execution state of a program with a *bitmap*. Roughly speaking, when the program executes a transition, AFL hashes the tuple (id_s,id_d) to an index in the bitmap and increments the counter at that index. AFL also uses a *global bitmap* to store the history of all bitmaps for the current fuzzing session. Monitoring for changes in the global bitmaps allows AFL to determine whether the input used for the current test is "interesting" (i.e., if it triggered previously-unseen program transitions and resulted in new coverage).

Directly measuring program state-space coverage during fuzzing is difficult, and modern coverage-guided fuzzers use code coverage as an approximation of program state-space coverage. However purely coverage-guided fuzzers cannot effectively explore the state space of a program [3], as *code coverage does not always correlate to program-state coverage.* Ijon [3] addresses these issues by augmenting the coverage bitmap with values of human-annotated state-representative variables to *increase the correlation between bitmap coverage and state space coverage*: For example, consider a maze program where each valid input character permits a move direction, as shown in Figure 1. Although the user has many possible positions and paths, a coverage-guided fuzzer that only considers code or branch coverage will struggle to explore them. Ijon exposes x and y to the fuzzer, enabling the fuzzer to recognize inputs leading to new locations in the maze as interesting, essentially creating a *player-location-based coverage bitmap* (the bottom of Figure 1). This results in a better exploration of the state space and a solution to the maze.

Ijon requires human annotation of state-representative variables, which does not scale. In this paper, we explore techniques to *automatically* infer and use state-representative variables to improve fuzzing efficacy.

3 Overview

We propose SandPuppy, a technique that automatically identifies and instruments state-representative variables, thus exposing program state to the fuzzer.

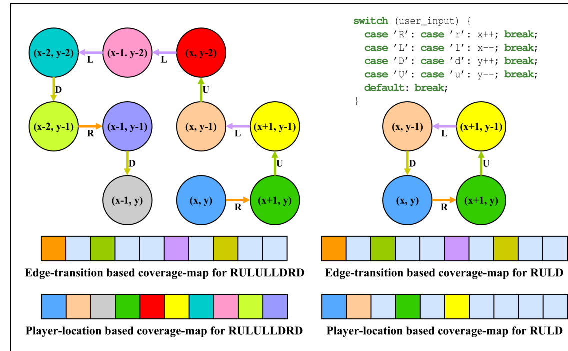

Fig. 1: Comparison of edge-transition based and player-location based coverage-maps for a maze program. Edge-transition cannot identify the distinct states resulting from different player locations whereas player-location can.

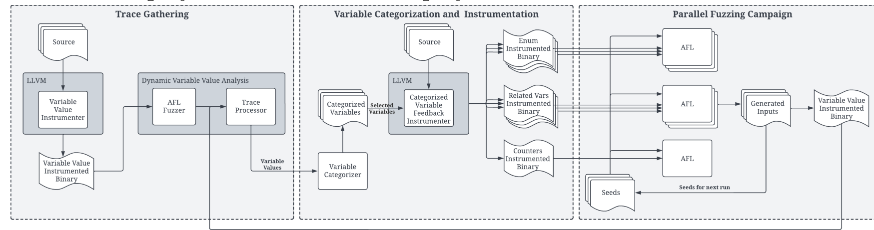

Fig. 2: SandPuppy's architecture is comprised of three phases: a trace-gathering phase that gathers variable value traces from an instrumented binary, a categorization and instrumentation phase that categorizes variables based on value traces and performs feedback instrumentation using selected variables, and a parallel-fuzzing phase where the instrumented binaries are fuzzed together. Inputs generated by the parallel-fuzzing campaign are used to generate additional traces and as seeds for subsequent runs.

Figure 2 provides an overview of SandPuppy's fuzzing pipeline. Given buildable program source, SandPuppy carries out the following steps iteratively, identifying and selecting new state-representative variables in each iteration:

Collect variable-value traces: SandPuppy instruments the target program to output integer-variable value traces. It then fuzzes this instrumented binary using a standard fuzzer for a fixed amount of time, using its accumulated set of test-cases (seeds). SandPuppy uses the trace results to select a tractable subset of state-representative variables, as described in Section 4.

Identify state-representative variables: SandPuppy analyzes the collected traces to identify potential state-representative variables and their semantic categorization (Section 5) using a set of heuristics (defined in Section 6 and applied in Section 7). Traces from inputs that result in crashes or errors are excluded to minimize noise by ignoring invalid states.

Fuzz: SandPuppy generates instrumented binaries that collect feedback from corresponding state-representative variables, as detailed in Section 8. It then runs a parallel-fuzzing session using this set of binaries. Note that generated inputs are

periodically synchronized between all fuzzing instances. While it is possible to generate a single binary for all identified variables, this significantly increases the likelihood of collisions with a single bitmap.

Repeat: The inputs generated by the parallel-fuzzing session are used as seeds for the next run's variable-value trace collection and fuzzing.

4 Trace Collection

SandPuppy classifies variables by observing their use by the program. It uses a trace collecting step to record the complete history of values that each integer variable receives throughout the instrumented program's execution.

SandPuppy needs a variety of traces. Consider a variable counting up to a number provided by input: if it is always zero, we have a single trace of the variable, which is indistinguishable from a trace of a constant variable set to zero and never modified.

For the first iteration of SandPuppy, we leverage an initial fuzzing phase to generate traces on multiple inputs, mitigating the influence of invalid inputs on later variable classification by discarding crashing and erroring traces. On subsequent iterations, we directly use inputs identified previously to generate traces. In this manner we can detect, categorize, and instrument any newly-detected variables. We also filter out the traces of any variable whose category has not changed since the last run, assuming that resources would be better spent on tracing and classifying still-changing variables.

For the variable v, we define the *value trace* T as a sequence of tuples (m,ν) where m is the source line-number where v was modified and ν is the current value of v. We also define $\mathbf{T}$ to be a sample of all traces collected for that variable from our fuzzing session; i.e., assuming the program was run against n inputs $\mathbf{T}=(T_1,T_2,...,T_n)$.

5 Semantic Categories

SandPuppy instruments *state-representative* variables to approximate program states at runtime. Different variables contribute differently to this approximation. Here, we describe categories of state-representative variables and our intuition of how they represent the application's state space. As we are not attempting to devise a general and exhaustive taxonomy of "state-representative variables", we focus on specific categories that see near-ubiquitous use across a wide variety of programs.

Enums. *Enumerated types* represent a fixed set of named values. While internal representation is arbitrary, they typically compile to integer values. Enums are used to enhance readability and leverage the type system to prevent variables from being set to invalid values. They can also be correlated with input, especially those involving structured formats.

Format processing code often feature variables encoding state associated with user input, which are attractive instrumentation targets. Listing 1.1 shows a message-processing function, where different message combinations can affect program state in unique ways. Coverage-based fuzzers identify values of `mtype`, but may not recognize combinations of messages as interesting states as they trigger already-seen control-flow edges. Conversely, fuzzers that can reason about the value of `mtype` directly may

```
void process_input(const char* input, state* state) {
  parsed_messages pmsgs = parse_input(input);
  for (int i = 0; i < pmsgs.num_messages; i++) {
    msg_type mtype = process_msg(pmsgs.messages[i], state);
    char* mtype_str = get_message_type(msg_str);
    if (strcmp(mtype_str, "type_1") == 0) {
        mtype = TYPE_1;
        process_type_1_msg(msg_str, state);
    } else if (strcmp(mtype_str, "type_2") == 0) {
        mtype = TYPE_2;
        process_type_2_msg(msg_str, state);
        ...
```

Listing 1.1: Enum variables exposing known state changes.

```
unsigned int processed_messages = 0;
void process_message(message* msg) {
  for (int i = 0; i < get_num_elements(msg); i++) {
    process_element(msg, i);
  }
  processed_messages++;
}
```

Listing 1.2: Counter variables exposing memory-related state.

be able to trigger additional program states. Similar code can be found in a wide variety of real-world software, such as FFMpeg, jsoncpp, libjpeg, and libpcap.

Counters. Counters are variables that count up or down to some value. We assume that counters are monotonically increasing or decreasing, and that they may do so by arbitrary (and varying) amounts. Counters can either be *static*: counting to a fixed value in every program execution, or *dynamic*: counting to different values in every program execution. Counters may also correlate with input as programs use them to count input elements or to sum input-element sizes. By guiding fuzzers towards maximum counter values, states revealing boundary and edge-case conditions can be explored.

Listing 1.2 shows an example of counter usage. By exposing the counter values `i` and `processed_messages` to the fuzzer and favoring executions that maximize their values, the fuzzer will explore program states involving both a larger number of messages and a larger number of elements per message.

Related Variables. Programs often feature *related variables* whose *combined* values represents a unique state. For example, variables x and y used to represent a player's location in a maze are related variables. By identifying related variables and exposing their combined state, fuzzers gain insight into a program's state space.

6 Semantic Category Heuristics

We use several heuristics on traces collected in Section 4 to classify variables as state-representative and sort them into categories that are described in Section 5.

Non-Semantic Categories. We start by defining heuristics to identify constants and booleans; two categories of variables that are not state-representative. Constants do not expose any useful state, as they do not change values. While booleans can expose useful state, coverage-based fuzzing is usually sufficient to explore these states as booleans often dictate control flow.

First, we define the observed set of values of the variable v as $\mathcal{V} = \bigcup_{T\in\mathbf{T}}\{\nu \mid (m,\nu)\in T\}$. We will also define the *number of unique values* $h_{nuniq} = |\mathcal{V}|$. If $h_{nuniq} = 1$, then it is highly probable that the variable is a constant. However, if $h_{nuniq} = 2$ and $\mathcal{V} = \{0,1\}$, then it is likely a boolean. There is one surprising case of constants: a variable initialized to zero and then set to a value later. To handle this case SandPuppy categorizes variables with $\mathcal{V} = \{0,n\}$ where $n \neq 1$ as constants.

6.1 Semantic Categories

Enums. We define a heuristic for identifying enums that take on values that are either directly present in the input or are mapped to certain input values. Additionally, we also assume that an arbitrary number of such values exist in the input. This means that, for enums, the number of times it is modified may correlate with the input size. Let us define the function $isz : 2^T \to \mathbb{Z}_{\geq 0}$ that returns the size of the input used to generate the trace T. We then have $\mathbf{S} = (isz(T_i) \mid T_i \in \mathbf{T})$ and $\mathbf{M} = (|T_i| \mid T_i \in \mathbf{T})$.

We can now calculate the Pearson correlation coefficient for our sample, called the *times-modified to input-size correlation*, as $h_{tmisc} = \frac{\sum_{i=1}^{|\mathbf{T}|}(s_i-\bar{s})(m_i-\bar{m})}{\sqrt{\sum_{i=1}^{|\mathbf{T}|}(s_i-\bar{s})^2}\sqrt{\sum_{i=1}^{|\mathbf{T}|}(m_i-\bar{m})^2}}$. The closer h_{tmisc} is to 1, the more likely that the variable is set based on values from the input. Note that this measure has some caveats, particularly if the discrete input elements are arbitrarily sized. Consider the example in Listing 1.1. With arbitrarily-sized messages, the program can process 3 messages of size n each, or a single message of size $3n$. In the first case the enum variable is modified three times, whereas in the second case it is modified only once. However, in both cases the input size is $3n$.

Consider another example demonstrated in Listing 1.1: The enum variable `mtype` is set based on the value of the string `mtype_str`. Also, `mtype` is modified on three lines; one for each message type. This means that if the number of lines where the variable is modified equals the number of unique values it can take, it is probably part of a construct such as the one in Listing 1.1. Therefore, we define the *number of modified lines* as $h_{nmod} = \left|\bigcup_{T\in\mathbf{T}}\{m \mid (m,\nu)\in T\}\right|$.

We also consider the case where values are assigned without being part of a structure similar to Listing 1.1. Here, the variable is only modified on one line.

Counters. We created several heuristics to identify if a variable is a counter. Our first intuition is to classify variables correlated with input size, as counters. This allows us to detect variables being used as such without detecting if they actually "count" (which other heuristics will do). If a strong correlation exists between a variable's maximum value in a trace and the input size for that trace, it is categorized as a counter. We first define $\mathbf{S} = (isz(T_i) \mid T_i \in \mathbf{T})$ and $\boldsymbol{\mathcal{V}}^{max} = (\nu_i^{max} \mid T_i \in \mathbf{T} \wedge \nu_i^{max} = \max(\{\nu_i \mid (m_i,\nu_i) \in T_i\}))$. Using this we can calculate the Pearson correlation coefficient for our sample, which we call the *maximum-value to input-size correlation*, as $h_{mvisc} = \frac{\sum_{i=1}^{|\mathbf{T}|}(s_i-\bar{s})(\nu_i^{max}-\bar{\nu}^{max})}{\sqrt{\sum_{i=1}^{|\mathbf{T}|}(s_i-\bar{s})^2}\sqrt{\sum_{i=1}^{|\mathbf{T}|}(\nu_i^{max}-\bar{\nu}^{max})^2}}$. Using h_{mvisc} as a measure of input-size correlation—the closer the value is to 1, the more likely it is that the variable holds values correlated with input size.

Next, we develop heuristics to detect counters and to distinguish between static and dynamic counters. We first define *maximum-values variance* as $h_{mvv} =$

$\frac{\sum_{i=1}^{|\mathbf{T}|}(\nu_i^{max}-\bar{\nu}^{max})^2}{|\mathbf{T}|}$. This heuristic is the variance of the observed maximum values of the variable. As static counters count over the same set of values every time, the corresponding value of h_{mvv} will be 0, while dynamic counters will have a non-zero value.

The second heuristic, the *average value-set-cardinality ratio*, is defined as $h_{avscr} = \frac{\sum_{T\in\mathbf{T}}\frac{|\{\nu|(m,\nu)\in T\}|}{|\mathcal{V}|}}{|\mathbf{T}|}$. This heuristic is an average of the ratio of the number of unique values in a trace, and the number of unique values across *all* traces for a particular variable. If h_{avscr} is 1, then the variable takes on the exact same set of values in every trace. If this variable is a counter, then it must be static, and we can ignore it as it does not expose useful state.

The next set of heuristics for identifying counters requires pre-processing of variable traces. The intent is to identify segments of traces that could be *counter segments*, meaning that the segment reflects a variable counting to some value. This is important because counters are frequently used multiple times in a single run, and so a trace can contain multiple such segments. It also ensures that each segment contains at least two values and that there are no repeating values in a segment. Runs can occur if a piece of code with a counter is called repeatedly, where the counter loops only a single time. But it can also happen in the case where a variable is being repeatedly set to the same value. Due to this ambiguity, we collapse such runs into a single value.

The first heuristic using counter segments is called *directional consistency*: $h_c = \frac{\left|\sum_{T\in\mathbf{T}}\sum_{S\in counter_segments(T)} dir(S)\right|}{\sum_{T\in\mathbf{T}}|counter_segments(T)|}$ where $dir(S)=1$ if $s_2 > s_1$ and -1 otherwise. The purpose of the directional consistency heuristic is to identify counter variables that are incremented or decremented consistently. For counters, h_c will be close to 1.

The final heuristic using counter segments is called *loop-sequence proportion*: $h_{lsp} = \frac{\sum_{T\in\mathbf{T}}\sum_{S\in counter_segments(T)\wedge|S|>2}|S|}{\sum_{T\in\mathbf{T}}|T|}$. This heuristic measures what proportion of a variable's traces are part of a loop sequence. For counter variables, this value should be close to 1. We deliberately exclude counter segments of length 2 or less, as segments of the form $[a,b]$ where $a > b$ or $a < b$ are common, meaning that even non-counter variables have a h_{lsp} close 1. However, it is far less likely for such variables to have segments of size 3 or higher where they are consistently incremented or decremented.

7 Identification of Semantic Categories

Once a set of variable-value traces are discovered, SandPuppy uses heuristics defined in Section 5 to categorize each variable into a semantic category. Note that the heuristics have thresholds associated with them, which we define here, and Section 9 describes the thresholds set during the evaluation. Variables deemed state-representative are instrumented to provide feedback during fuzzing (described in Section 8).

Enums and Counters. To determine an enum, SandPuppy checks if h_{tmisc} is greater than or equal to a threshold τ_{tmisc}. If it is modified on multiple lines, SandPuppy checks if the number of lines is the same as the number of values it receives.

SandPuppy identifies variables correlated with input size by checking if the variable's h_{mvisc} feature is greater than or equal to the threshold τ_{mvisc}.

For counters, we check to see if a variable's h_{lsp} and h_c are greater than or equal to the threshold τ_{lsp} and τ_c. To determine if a counter is static or not, SandPuppy

checks if h_{mvv} is 0 (i.e., it always counts to or from the same value) and h_{avscr} is 1 (i.e., it counts over the same set of values every time).

Related Variables. SandPuppy identifies related variables starting with all variables from the specified function and file that are not constants, booleans, or unclassified variables, to include those already been categorized as enums or counters.

SandPuppy then creates a candidate list of related variables for each variable. First, SandPuppy chooses one of the lines where the variable is modified. Second, it scans up to δ_{max} lines before and after to identify other modified variables. SandPuppy does not do this for every line that modifies the variable because the intuition is that related variables are modified in close proximity to each other. Lastly, SandPuppy prunes each variable's candidate list to include only those variables that it considers related, by ensuring that (1) both variables are modified on the same number of lines and (2) their trace-length distributions are identical.

SandPuppy's pruning strategy does exclude variable pairs that have identical trace-length distributions. For example, SandPuppy will not identify related variables when one variable is modified on one line, but the second is modified on mutually-exclusive conditions. Here, for any trace generated from the same input, both variables be modified the same number of times, even if overall, the second variable is modified on more lines than the first. This makes feedback instrumentation simpler by avoiding reasoning about the context in which variables are modified.

8 Feedback Instrumentation

Having mapped variables to semantic categories, SandPuppy instruments state-representative variables to generate several variants of the program that expose corresponding feedback information.

Our implementation is based on the feedback mechanisms proposed in Ijon [3].Instrumentation uses LLVM passes based on the FuzzFactory framework [31]. In all cases, we use the FuzzFactory max reducer function to aggregate feedback across multiple inputs. The max reducer function marks an input as interesting if in the corresponding bitmap, a new location is set, or if a location has a higher value.

Permutation Feedback. We perform this instrumentation on enum variables to identify inputs resulting in unique permutations of values, potentially allowing exploration of associated complex-states. We generate a variant with this feedback for each identified enum variable. This instrumentation is similar to Ijon's state-change log.

Maximization Feedback. This instrumentation is similar to the `IJON_MAX(x)` primitive [3]. For dynamic counters and variables correlated with input size, we employ instrumentation to guide fuzzing towards inputs that maximize their values. We generate a single variant with this feedback for all identified counters.

Related-variable Maximization Feedback (RVMF). We perform this instrumentation on every related-variable pair; it is similar to the `IJON_MAX(slot, x)` primitive [3]. It maximizes the value of one variable with respect to another. A real-world example of such code can be seen in FFMpeg's DTS-HD demuxer. We generate two instrumented variants for each identified pair of related variables, as we do not know which variable is the dependent one.

Related-variable Combination Feedback (RVCF). This instrumentation is performed on every related-variable pair, and is similar to `IJON_SET(hash_ints(x, y))` annotation in IJON [3].

9 Evaluation

We perform evaluations to understand SANDPUPPY's ability to automatically add state-representative variable feedback to fuzzing, and the impact of this to find bugs.
Reasoning about states. We first develop understanding of SANDPUPPY's core contribution by evaluating its variable semantic category identification in Section 9.1 and exploring the value of this identification in the actual exploration of complex states in software in Section 9.2.
Fuzzing for vulnerabilities. We then perform a comparative evaluation of SANDPUPPY against state-of-the-art fuzzers, both in terms of achieved code coverage in Section 9.3 and triggered vulnerabilities in Section 9.4.
Understanding variable impact. For a deeper understanding of individual categories identified by SANDPUPPY, we perform an ablative evaluation in Section 9.5.

We performed initial fuzzing for trace-gathering and classification on a Linux Mint 20.2 machine with an Intel Core i9-8950HK CPU and 32 GB RAM. Parallel fuzzing was performed on a 40-node Kubernetes cluster with each node having two Intel(R) Xeon(R) CPU E5-2670 v2 processors. Each fuzzing instance receives identical resources during the parallel-fuzzing session.

We set initial fuzzing time to one hour, and found that the instrumentation slows the fuzzer down significantly, making it equivalent to around 5-10 minutes of fuzzing with a regular AFL instrumented binary. The results will demonstrate however, that even with this performance hit, SANDPUPPY can identify and categorize state-representative variables.

Through manual experimentation and observation on LIBTPMS and the test-bed described in Section 9.1, we derived the following threshold values: $\tau_{mvisc} = 0.6$, $\tau_{lsp} = 0.7$, $\tau_c = 0.8$, $\tau_{tmisc} = 0.5$, and $\delta_{max} = 6$. We note that these values may be program dependent and could potentially vary between programs. However, in our experiments these threshold values appeared consistent across multiple programs.

9.1 Identification of Semantic Categories

There is a fundamental trade-off between extraneous instrumented state representative variables (too many variants slow down fuzzing) and insufficient state representative variables (missing exposing state to fuzzing). As our goal is to improve fuzzing efficacy, we evaluated SANDPUPPY against a custom test-bed and against real-world programs to better understand its state reasoning. We present some results here, and detailed results for our synthetic test bed in the to-be-released extended version.

We evaluated SANDPUPPY's automatic variable categorization on the following programs: LIBPNG (1.6.37), LIBTPMS (0.8.3), JSONCPP (1.9.5), DMG2IMG (commit `a3e4134`), and SUPER MARIO BROS. In this experiment, we manually created ground-truth of the semantic category of the variables. We discovered minor discrepancies

Table 1: Sample of classified variables with labels from various real-world programs. Variable names include the file and function name. The last column specifies whether the variable was identified as state-representative or not.

Target	Qualified Variable Name	Label	State Representative
libpng	pngrutil.c:png_check_chunk_name:i:3140	static_counter	N
	pngrutil.c:png_check_chunk_name:c:3147	enum_from_input	Y
	pngrutil.c:png_read_filter_row_up:i:3956	dynamic_counter	Y
dmg2img	dmg2img.c:percentage:i:77	static_counter	N
	base64.c:decode_base64_char:c:49	enum_from_input	Y
	dmg2img.c:read_kolyblk:k.XMLLength:148	dynamic_counter	Y
smbc	PPU.cpp:PPU::writeDMAEh:i:501	static_counter	N
	Main.cpp:mainLoop:world_pos:309	related_variable	Y
	Main.cpp:mainLoop:pos_y:311	related_variable	Y

limited to dynamic counters being classified as enums and some enum variables that were not identified. Two factors cause this: (1) lack of a sufficiently-representative trace-distribution for a variable and (2) ambiguity inherent in our approach for detecting enums, which depends on correlation between input size and how often the variable is modified. Our analysis, a sample which is in Table 1, shows largely accurate identification of enums, static and dynamic counters, and related variables.

We also compared the state-representative variables SandPuppy identified to those *manually identified* by Ijon [3] and found every variable manually identified by human-experts, including `command.index` in libtpms and Mario's screen position in Super Mario[1]. SandPuppy identified the same variables as Ijon, and with the appropriate semantic category.

9.2 Exploring Complex States

We evaluated SandPuppy's ability to explore complex state typically not accessible to purely coverage-based approaches; games and protocol parsers present a significant challenge as certain states can only be reached through a particular input sequence. We limited SandPuppy to a single run as initial trace-collection was sufficient to correctly identify and categorize variables that expose state information.

Super Mario Bros. We used the game version modified by Ijon authors [3]: Changes include reading input from `stdin` and speedrunning changes. SandPuppy identified a number of related variables, including those describing Mario's position in the world. We fuzzed Super Mario Bros with AFL and SandPuppy for up to 200 hours using identical seeds, recording Mario's progress. Results in Table 2 include how often the level was solved, the median time until the best solution was found (hh:mm), and the median percentage of distance covered.

SandPuppy solved all worlds except two and found the warp zones in worlds 1-2 and 4-2. World 2-1 could not be fully solved as Mario cannot use the spring due to the speedrunning code-changes. In world 4-4, Mario repeatedly gets stuck in a dead end. SandPuppy solved worlds 6-2 and 8-4 that Ijon could not.

While most solutions were found by targets using RVMF from player coordinates, some were using RVCF from the same variables. In a few cases, targets using RVMF

[1] Note that libpng is not included in the released data: `https://github.com/RUB-SysSec/ijon-data`

Table 2: SandPuppy, AFL, and Ijon playing Super Mario Bros (3 runs each). We show how often the level was solved and the median distance traveled. For SandPuppy and AFL, we include the median time (hh:mm) to find the solution.

	SandPuppy			**AFL**			**Ijon**	
World	Solved	Time	% Distance	Solved	Time	% Distance	Solved	% Distance
1-1	✓	27:58	100%	✗	168:04	69%	✓	100%
1-2	✓	30:03	100%	✗	96:15	68%	✓	100%
1-3	✓	41:10	100%	✗	58:04	53%	✓	100%
1-4	✓	04:47	100%	✗	32:47	87%	✓	100%
2-1	✗	46:55	94%	✗	136:42	65%	✗	94%
2-2	✓	04:13	100%	✗	61:37	74%	✓	100%
2-3	✓	05:15	100%	✗	40:25	81%	✓	100%
2-4	✓	05:38	100%	2/3	125:36	100%	✓	100%
3-1	✓	52:19	100%	✗	10:00	29%	✓	100%
3-2	✓	09:42	100%	✗	54:29	66%	✓	100%
3-3	✓	12:02	100%	✗	90:18	94%	✓	100%
3-4	✓	07:09	100%	✗	26:21	74%	✓	100%
4-1	✓	09:51	100%	✗	72:03	95%	✓	100%
4-2	✓	03:54	118%	✗	18:18	57%	✓	118%
4-3	✓	13:44	100%	✗	26:31	56%	✓	100%
4-4	✗	04:13	83%	✗	128:20	82%	✗	82%
5-1	✓	12:05	100%	✗	192:01	45%	✓	100%
5-2	✓	18:01	100%	✗	179:52	82%	✓	100%
5-3	✓	08:42	100%	✗	91:03	63%	✓	100%
5-4	✓	05:16	100%	1/3	78:34	100%	✓	100%
6-1	✓	10:34	100%	✗	02:28	47%	✓	100%
6-2	✓	136:20	100%	✗	11:17	37%	✗	82%
6-3	✓	09:08	100%	✗	32:03	85%	✓	100%
6-4	✓	04:03	100%	1/3	63:22	100%	✓	100%
7-1	✓	22:34	100%	✗	196:51	68%	✓	100%
7-2	✓	10:17	100%	✗	178:09	82%	✓	100%
7-3	✓	12:43	100%	✗	19:40	81%	✓	100%
7-4	✓	08:55	100%	1/3	28:21	100%	✓	100%
8-1	1/3	164:29	100%	✗	21:16	22%	2/3	100%
8-2	1/3	199:26	100%	✗	47:05	60%	2/3	100%
8-3	✓	10:47	100%	✗	19:12	72%	✓	100%
8-4	✓	21:10	100%	✗	09:25	47%	✗	-

from Mario's world and screen x coordinates also found solutions. As the latter is constant, this normally causes Mario to deadend. However syncing inputs with other fuzzing instances appears to have a synergistic effect, allowing one with a simple but incomplete strategy to improve on a partial solution from another instance. We believe this allowed SandPuppy to solve levels 6-2 and 8-4 when Ijon could not.

Ijon also examined the exploration of complex state in maze programs, and we discovered that SandPuppy was able to successfully solve all mazes, including one of our own creation. We do not present the details due to space constraints.

State Exploration Evaluation. We evaluated SandPuppy's ability to explore the state-space of stateful parsers on the libtpms library. This is a real-world software emulator for a Trusted Platform Module (TPM), and was also used in Ijon [3]. The protocol defines various command-types and associated payloads, which affect processor-state in different ways.

To evaluate the system on generating complex messages over a significant time-frame, we fuzzed libtpms for over 100 hours with SandPuppy, AFL, AFL++, RedQueen, and LAFIntel, using diverse seeds and equal compute.

We evaluate state-space exploration by tracking the following: total number of unique command-sequences, counts of unique command-sequences of different lengths,

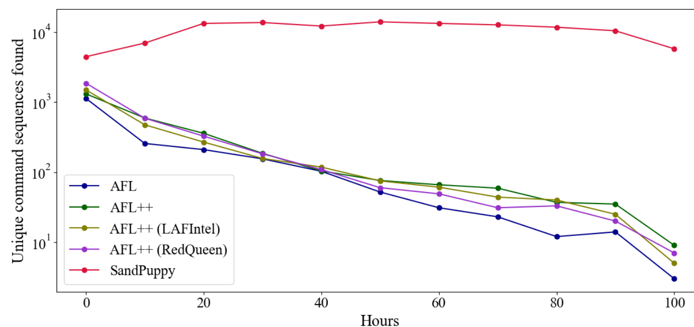

Fig. 3: Unique command-sequences with parameter size found over time in libtpms. SandPuppy consistently finds more unique-sequences throughout the session compared to other fuzzers, which "stall" as fuzzing progresses. **The y-axis is in log scale.**

Table 3: Number of distinct command subsequences and total unique sequences found by all fuzzers in libtpms, without accounting for command parameter-size and with. SandPuppy outperformed all other fuzzers.

Sub-seq. Length	AFL	AFL++	LAFIntel	RedQueen	SandPuppy
1	96 / 737	94 / 741	105 / 876	97 / 783	**111 / 115,751**
2	301 / 978	328 / 1,050	392 / 1,259	348 / 1,077	**671 / 116,613**
3	413 / 691	589 / 932	677 / 1,098	671 / 989	**879 / 2,659**
4	461 / 633	865 / 1,101	907 / 1,172	1,075 / 1,324	**1,186 / 1,792**
Unique Sequences	1,324 / 1,991	2,137 / 2,840	1,882 / 2,769	2,706 / 3,379	**2,998 / 119,278**

Table 4: Number of variables detected after each trace-generation step of each run.

Target \ Run	Initial	1	2	3	4	5
	Variables detected					
jsoncpp	90	107	107	107	108	108
libtpms	554	872	912	912	912	945
PcapPlusPlus	384	495	499	500	506	506
readelf	314	595	626	636	637	640

and counts of unique command sub-sequences of lengths 1, 2, 3, and 4. We also tracked command parameter-sizes, which helps distinguish between different varieties of the same command. SandPuppy was able to automatically identify and correctly categorize state-variables holding this information.

The results show that SandPuppy outperforms other fuzzers after a single run. SandPuppy found more unique sub-sequences overall, and more unique sequences of various lengths as well; especially when incorporating parameter size (Table 3). Based on sequences over time (Figure 3), SandPuppy finds them at a higher and relatively-constant rate compared to other fuzzers.

9.3 Coverage Evaluation

We used jsoncpp, libtpms, PcapPlusPlus (commit `a817631`), and readelf (binutils 2.32), and evaluated SandPuppy against AFL, AFL++, LAFIntel, RedQueen, SGFuzz and Ferry.

We ran SandPuppy for 6 iterations (runs), 4 hours each, for a total fuzzing-time of 24 hours. As the number of SandPuppy targets vary with each run, a one-to-one

Table 5: Improvement over baseline coverage. The value in parentheses is the percentage of unique coverage relative to the baseline. For FERRY comparison is done only using basic-block counts as it uses a symbolic approach and does not have initial seeds.

Fuzzer / Target	Coverage Improvement (%)							Blocks Covered	
	AFL	**AFL++**	**LAFIntel**	**RedQueen**	**SGFuzz**	**SandPuppy**	**Total**	**Ferry**	**SandPuppy**
jsoncpp	45.78 (0.00)	46.09 (0.00)	45.98 (0.00)	45.76 (0.00)	42.63 (0.33)	**46.65** (0.00)	47.10	623	**1314**
libtpms	96.78 (0.00)	111.77 (0.76)	112.26 (0.49)	111.06 (0.49)	0.33 (0.00)	**113.78** (5.67)	120.11	47	**3923**
pcapplusplus	28.37 (0.00)	34.23 (0.00)	35.97 (0.00)	**86.23** (10.67)	69.25 (0.00)	85.70 (10.40)	97.31	251	4910
readelf	235.58 (1.37)	188.52 (1.49)	174.74 (3.8)	178.11 (14.25)	n/a	**261.48** (9.6)	292.32	342	**8473**

comparison based on equal compute is difficult. We therefore calculate the rounded-up average of the number of targets fuzzed in each SANDPUPPY run, and allocate that many parallel-fuzzing sessions for the other fuzzers, which we then fuzz for 24 hours. The SGFUZZ and FERRY experimental setup was similar as described by the respective authors, and were run for 24 hours as well.

Instrumented variables are recognized only if traces execute their code path. Table 4 shows that SANDPUPPY detects an increasing number of variables in the trace-generation phase, after each fuzzing run. This allows it to potentially explore additional complex states and also suggests that coverage increases over each run. To quantify, we calculated the percentage increase over baseline coverage, which is derived from the initial seeds provided to all fuzzers. We also calculated the percentage of uniquely covered blocks with respect to baseline coverage for each fuzzer, to quantify uniqueness of coverage (see Table 5).

The results (Table 5) show that SANDPUPPY superseded or matched the performance of the other fuzzers in raw coverage improvement on JSONCPP, LIBTPMS, and READELF. For PCAPPLUSPLUS, SANDPUPPY's achieved 85.70% coverage improvement compared to REDQUEEN's 86.23%. SANDPUPPY provides significant unique coverage on LIBTPMS, PCAPPLUSPLUS, and READELF.

A few cases stood out when inspecting unique basic blocks that were covered by SANDPUPPY. In LIBTPMS, SANDPUPPY triggered a path that detects and mitigates dictionary attacks against the TPM processor by five successive authorization failures. In the case of PCAPPLUSPLUS, SANDPUPPY triggered parsing of DHCP requests, which rely on a specific combination of source and destination UDP ports. These findings show that in addition to improving coverage in general, SANDPUPPY can identify and cover multiple deep and complex states.

We did notice changes in variable classification from run to run and verified that re-classifications were accurate based on variable usage in the source code. Misclassifications have the potential to generate useless inputs, but otherwise do not appear to degrade performance. We also observed that some misclassified variables were accurately re-classified as non state-representative, which reduced the generation of useless inputs in subsequent runs.

Compared to SGFUZZ, SANDPUPPY exceeds coverage performance. The first reason is SGFUZZ's fragile regex-based static analysis; when used on BINUTILS source to instrument READELF, SGFUZZ crashed due to issues parsing enum definitions containing comments with braces. When used against LIBTPMS, SGFUZZ could iden-

Table 6: Bugs discovered by each fuzzer in PcapPlusPlus. SandPuppy discovered 10 out of 11 bugs (2 uniquely). LAFIntel, AFL++, RedQueen, and AFL discovered 9, 6, 4, and 3 bugs respectively. The last column specifies whether the vulnerable block was covered by other fuzzers.

Bug	AFL	AFL++	LAFIntel	RedQueen	SandPuppy	Covered by others
Heap overflow #1	✗	✗	✓	✗	✓	✓
Heap overflow #2	✓	✓	✓	✓	✓	✓
Heap overflow #3	✗	✓	✓	✓	✗	✓
Heap overflow #4	✓	✓	✓	✓	✓	✓
Heap overflow #5	✗	✓	✓	✓	✓	✓
Heap overflow #6	✗	✓	✓	✗	✓	✓
Heap overflow #7	✗	✗	✗	✗	✓	✗
Heap overflow #8	✓	✓	✓	✗	✓	✓
Heap overflow #9	✗	✗	✓	✗	✓	✓
Stack overflow #1	✗	✗	✓	✗	✓	✓
Stack overflow #2	✗	✗	✗	✗	✓	✓

tify certain enum definitions. It was not able to detect or instrument *any* variables, as their values are set using integer constants instead of enum constants.

SGFuzz also identifies false-positives that are not actually enums and the subsequent instrumentation causes a compilation error due to a type-mismatch. While an ignore-list solves this issue, SGFuzz does not distinguish between identically-named variables based on usage context.

To fairly evaluate Ferry, we used individual block counts as Ferry uses a symbolic approach that does not allow providing initial input seeds. SandPuppy significantly outperforms Ferry. When running against jsoncpp, PcapPlusPlus, and readelf, Ferry terminated early. In the case of libtpms, Ferry ran out of memory when given 64 GB.

9.4 Vulnerability Detection

We evaluated SandPuppy's ability to find vulnerabilities on two real-world targets: dmg2img and PcapPlusPlus.

We use **dmg2img** as it was fuzzed in academic research by WEIZZ [16] and Ijon [3]. SandPuppy identified multiple state-representative variables, including `kolyblk.XMLLength`, which was manually identified by Ijon authors. SandPuppy discovered 7 bugs; 4 were identified by WEIZZ and Ijon, and 2 are known issues (Bug 778819 and CVE-2021-32614). The last (unknown) bug is a one-byte read heap-buffer overflow in `convert_char4`, which we reported.

PcapPlusPlus is a library for capturing and manipulating network packets, featuring state-representative variables and stateful logic. Here we evaluated SandPuppy against AFL, AFL++, RedQueen, and LAFIntel, with equivalent fuzzing time and compute. The results are shown in Table 6: for a total of 11 previously unknown bugs (which we reported); including 2 stack-overflows and 9 heap-overeads. SandPuppy discovered 10 out of 11 bugs, out of which 2 were not found by other fuzzers. LAFIntel, AFL++, RedQueen, and AFL discovered 9, 6, 4, and 3 bugs respectively.

Of the two unique bugs, Heap Overflow #7 was triggered the "traditional" way: SandPuppy's state awareness triggered additional code. The buggy code containing

Table 7: Number of distinct command subsequences and total unique sequences with parameter size found for each feedback-evaluation run in LIBTPMS. Preceding minus signs describe runs where corresponding feedback was excluded. Perm by itself outperforms all others and excluding it leads to a dramatic drop in performance.

Sub-seq. Length	RVCF	Perm	Max	RVMF	-RVCF	-Perm	-Max	-RVMF	Random	Regular
1	222	**41,492**	638	260	15,824	623	16,864	18,062	379	12,998
2	277	**41,754**	756	335	16,381	930	17,230	18,552	497	13,547
3	218	700	166	262	**1,237**	454	820	1,064	298	978
4	69	669	38	104	**702**	113	601	654	271	377
Unique Sequences	425	**43,073**	768	559	17,177	1,013	17,852	19,189	1,170	13,964

Table 8: Coverage (% of basic blocks hit) for each feedback-evaluation run in LIBPNG. Preceding minus signs describe runs where corresponding feedback was excluded. Excluding Perm increases performance vs. all other runs. RVMF performs best overall.

	RVCF	Perm	Max	RVMF	-RVCF	-Perm	-Max	-RVMF	Random	Regular
Coverage	32.10%	31.80%	31.76%	33.60%	32.99%	**34.47%**	31.97%	32.15%	31.87%	32.13%

Stack Overflow #2 was executed by *all* fuzzers, but SANDPUPPY triggered it with the correct *state* by identifying and instrumenting a computed length-variable.

9.5 Feedback Instrumentation Evaluation

The results so far demonstrate that SANDPUPPY identifies state-representative variables providing feedback sufficient enough to improve coverage, vulnerability identification, and state-exploration. However not *all* instrumented variables and associated feedback necessarily expose *useful* state enhancing these metrics. There is also a risk of fuzzing slowdown due to generating a large number of useless inputs [3].

We designed an experiment to measure the effect of each feedback instrumentation on coverage and state-exploration. We fuzzed LIBTPMS and LIBPNG for 24 hours with a single feedback-type enabled/excluded, and with all feedback enabled on a random subset of variables. We evaluated these against a regular SANDPUPPY run.

Results in Table 7 show that permutation improves state exploration the most; excluding it reduces state exploration. In contrast, excluding other feedback types (RVCF, maximization, and RVMF) increases state diversity.

This trend holds when evaluating individual feedback: Permutation feedback by itself is almost four times better, while the remaining are not better than random. This suggests that, based on this particular diversity-metric, other feedback types do not expose useful state and generate slow inputs.

LIBPNG results are in Table 8. Here, *excluding* permutation feedback increases coverage. Variables instrumented include the chunk name and the chunk-header length. While the feedback it to recognize different combinations of chunk types and associated lengths, it results in a large number of inputs.

In contrast, RVMF improves coverage both by itself and when included with maximization feedback and RVCF. In either case coverage is better than when fuzzing with all feedback types, as some instrumented variable-pairs hold information related to image or chunk dimensions and sizes. These variables are also involved in a number of sanity or boundary checks and conditional parsing-logic. Maximizing these

variables with respect to each other is likely to exercise these paths, resulting in greater coverage. This is more likely when the fuzzing campaign is not overwhelmed by many uninteresting inputs, which is observed when we disable the permutation feedback.

Both cases demonstrate that fuzzing with all feedback types generally improves coverage and state exploration, the usefulness depends on the target and evaluation metrics. Better performance in future fuzzing runs could be achieved by analyzing the contributions of different feedback types, and including only those that enhance coverage.

10 Discussion, Limitations, and Future Work

We propose an approach that guides fuzzing towards deep states using feedback from automatically-identified state-representative variables. SandPuppy is used in an end-to-end, dynamic-feedback approach where test cases discovered after one full run are used for trace-generation and variable classification, and as initial seeds in a subsequent run. As coverage generally increases, new test cases have a chance to exercise new code paths, leading to the discovery of additional state-representative variables, thus allowing SandPuppy to explore deeper states within a program.

But all instrumented targets are not useful (Section 9.5). An evaluation step could attempt to optimize how each instrumented target improves coverage and their usage.

In large programs, SandPuppy can identify a large number of state-representative variables, resulting in a correspondingly large number of targets that need to be built and fuzzed. FuzzFactory can support up to four coverage maps at a time; this might allow exposing feedback from multiple variables through a single binary.

During coverage evaluation, we ran SandPuppy for six iterations of 4 hours each to compare against 24 hour runs of other fuzzers. However it is possible to run SandPuppy for any number of iterations of any duration. Furthermore, it is also possible to run SandPuppy for as long as needed, and terminate on other conditions.

As our intent was not to solve the variable semantic-categorization problem in general, we did not focus on a detailed analysis of false positives and negatives. The lack of a baseline and ground-truth also hampered this approach. While the results demonstrate that SandPuppy has practical benefits and improves code coverage, state coverage, and vulnerability discovery, a detailed analysis of the impacts of false positive and negatives on performance would be valuable.

11 Related Work

Core fuzzing. Seed selection and scheduling enhancements have increased AFL performance in different situations [8, 9, 44, 36, 12, 43]. Target information can be used to improve seed-mutation strategies [4, 2, 6, 19, 30, 34, 26, 24]. Taint tracking improved efficiency by focusing mutation only on interesting input [10, 35, 27, 28, 17].

Tweaking coverage. Angora, CollAFL, kAFL, and InsTrim enhance coverage-tracking and feedback by improving recognition of novel coverage [10, 18, 37, 20]. Angora and CollAFL reduce coverage-map collisions, thus helping AFL uniquely identify more program states. kAFL uses hardware-assisted techniques to improve

coverage information when fuzzing kernels, while InsTrim uses CFG-aware instrumentation to improve execution-path differentiation.

Input-state correspondence. LAFIntel [22] solves the magic-value comparison issue by splitting compare instructions into multiple smaller ones. Steelix [25] uses lightweight analysis for comparison progress and identifying locations of magic bytes.

Concolic [39, 49, 45, 21, 13, 11] and symbolic-execution [40, 29, 48, 47] approaches can handle magic bytes or deeply-nested branches, but struggle to explore complex/deep states relying on a specific sequence of state updates. Symbolic execution can help solve some of these problems, but scalability is a concern [8] due to state explosion.

RedQueen [4] is an alternative to symbolic/concolic execution and taint tracking, as it derives input-state correspondence. While SandPuppy also relies on internal program-state, we expose it to influence feedback instead of correlating with input.

Domain-specific state reasoning. Domain-specific fuzzing [31, 23, 33, 14, 42] has solved certain deep-state fuzzing problems. PerfFuzz [23] and SlowFuzz guide the fuzzer towards inputs to expose algorithmic-complexity vulnerabilities, while MemFuzz [14] augments coverage information by accounting for memory accesses.

General program state reasoning. Ijon uses manual annotations [3] for deep-state fuzzing. While Ijon overcomes a number of deep-state-fuzzing obstacles and surpasses prior fully-automated approaches, it requires a human to recognize state-representative variables that can improve coverage. SandPuppy builds on Ijon by automating detection and instrumentation of state-representative variables.

InvsCov [15] explores program state-space to improve coverage and detect vulnerabilities using runtime dynamic-analysis of program variables, similar to SandPuppy. But the focus is on identifying invariants and augmenting coverage feedback with this information, to partition program state-space and guide the fuzzer towards states violating these invariants, potentially triggering vulnerabilities.

SGFuzz [5] explores state-spaces using regex-based static analysis to identify enums in protocol parsers. SGFuzz also maintains a global variable-ignore-list of unqualified variable names. SandPuppy focuses on additional variable categories making it applicable to more program types. We also identify variables based on their semantics rather than syntax, allowing us to detect those that behave like enums.

Ferry [50] also explored identification of state-representative variables, categorizing them based on certain characteristics (different from SandPuppy). Rather than exploring the triggering of other program *states*, Ferry exclusively uses information from these variables to explore dependent *branches*. Based on our evaluation results, we saw that SandPuppy significantly outperforms Ferry.

12 Conclusion

SandPuppy is a fully-automated source-code based fuzzing approach that explores deep-states in programs by identifying and labeling state-representative integer variables using runtime value-traces, and instrumenting them to provide appropriate feedback based on their semantics. SandPuppy effectively automates Ijon, requires no knowledge or manipulation of program source, and results in performance that either matches, or in certain cases slightly outperforms Ijon. We believe that SandPuppy represents the first step toward fully-automated deep-state fuzzing.

Acknowledgements

This research project has received funding from the following sources: Defense Advanced Research Projects Agency (DARPA) Contracts No. FA875019C0003 and N6600122C4026; the Advanced Research Projects Agency for Health (ARPA-H) Grant No. SP4701-23-C-0074; the Department of the Interior Grant No. D22AP00145-00; and National Science Foundation (NSF) Awards No. 1663651, 2146568, 2232915, and 2247954. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Government, and no official endorsement should be inferred.

References

1. Libfuzzer – a library for coverage-guided fuzz testing., `https://llvm.org/docs/LibFuzzer.html`, accessed October 26, 2021
2. Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.R., Teuchert, D.: Nautilus: Fishing for deep bugs with grammars. In: NDSS (2019)
3. Aschermann, C., Schumilo, S., Abbasi, A., Holz, T.: Ijon: Exploring deep state spaces via fuzzing. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1597–1612. IEEE (2020)
4. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: $REDQUEEN$: Fuzzing with input-to-state correspondence. In: NDSS. vol. 19, pp. 1–15 (2019)
5. Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful greybox fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3255–3272 (2022)
6. Blazytko, T., Bishop, M., Aschermann, C., Cappos, J., Schlögel, M., Korshun, N., Abbasi, A., Schweighauser, M., Schinzel, S., Schumilo, S., et al.: $GRIMOIRE$: Synthesizing structure while fuzzing. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1985–2002 (2019)
7. Böhme, M.: Aflfast, `https://github.com/mboehme/aflfast`, accessed October 26, 2021
8. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344 (2017)
9. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. IEEE Transactions on Software Engineering **45**(5), 489–506 (2017)
10. Chen, P., Chen, H.: Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 711–725. IEEE (2018)
11. Chen, P., Liu, J., Chen, H.: Matryoshka: fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 499–513 (2019)
12. Chen, Y., Ahmadi, M., Wang, B., Lu, L., et al.: $MEUZZ$: Smart seed scheduling for hybrid fuzzing. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). pp. 77–92 (2020)
13. Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T., Lu, L.: Savior: Towards bug-driven hybrid testing. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1580–1596. IEEE (2020)
14. Coppik, N., Schwahn, O., Suri, N.: Memfuzz: Using memory accesses to guide fuzzing. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 48–58. IEEE (2019)
15. Fioraldi, A., D'Elia, D.C., Balzarotti, D.: The use of likely invariants as feedback for fuzzers. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2829–2846 (2021)
16. Fioraldi, A., D'Elia, D.C., Coppa, E.: Weizz: Automatic grey-box fuzzing for structured binary formats. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 1–13 (2020)
17. Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., Chen, Z.: $GREYONE$: Data flow sensitive fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2577–2594 (2020)
18. Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z.: Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 679–696. IEEE (2018)
19. Han, H., Oh, D., Cha, S.K.: Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In: NDSS (2019)
20. Hsu, C.C., Wu, C.Y., Hsiao, H.C., Huang, S.K.: Instrim: Lightweight instrumentation for coverage-guided fuzzing. In: Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research (2018)
21. Huang, H., Yao, P., Wu, R., Shi, Q., Zhang, C.: Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1613–1627. IEEE (2020)
22. Lafintel: Lafintel (Aug 2016), `https://lafintel.wordpress.com/`
23. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 254–265 (2018)

24. Lemieux, C., Sen, K.: Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 475–485 (2018)
25. Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A.: Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 627–637 (2017)
26. Lyu, C., Ji, S., Li, Y., Zhou, J., Chen, J., Chen, J.: Smartseed: Smart seed generation for efficient fuzzing. arXiv preprint arXiv:1807.02606 (2018)
27. Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., Zeller, A.: Parser-directed fuzzing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 548–560 (2019)
28. Mathis, B., Gopinath, R., Zeller, A.: Learning input tokens for effective fuzzing. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 27–37 (2020)
29. Ognawala, S., Hutzelmann, T., Psallida, E., Pretschner, A.: Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1475–1482 (2018)
30. Padhye, R., Lemieux, C., Sen, K., Papadakis, M., Le Traon, Y.: Validity fuzzing and parametric generators for effective random testing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 266–267. IEEE (2019)
31. Padhye, R., Lemieux, C., Sen, K., Simon, L., Vijayakumar, H.: Fuzzfactory: domain-specific fuzzing with waypoints. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–29 (2019)
32. Peng, H., Shoshitaishvili, Y., Payer, M.: T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 697–710. IEEE (2018)
33. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2155–2168 (2017)
34. Pham, V.T., Böhme, M., Santosa, A.E., Căciulescu, A.R., Roychoudhury, A.: Smart greybox fuzzing. IEEE Transactions on Software Engineering **47**(9), 1980–1997 (2019)
35. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: Application-aware evolutionary fuzzing. In: NDSS. vol. 17, pp. 1–14 (2017)
36. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 861–875 (2014)
37. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: $kAFL$: Hardware-assisted feedback fuzzing for OS kernels. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 167–182 (2017)
38. Shoshitaishvili, Y., Weissbacher, M., Dresel, L., Salls, C., Wang, R., Kruegel, C., Vigna, G.: Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 347–362 (2017)
39. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
40. Wang, M., Liang, J., Chen, Y., Jiang, Y., Jiao, X., Liu, H., Zhao, X., Sun, J.: Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 61–64 (2018)
41. Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., Su, P.: Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In: Network and Distributed System Security Symposium (2020). https://doi.org/10.14722/ndss.2020.24422
42. Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., Su, P.: Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In: NDSS (2020)
43. Wang, Y., Wu, Z., Wei, Q., Wang, Q.: Neufuzz: Efficient fuzzing with deep neural network. IEEE Access **7**, 36340–36352 (2019)
44. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 511–522 (2013)
45. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: $QSYM$: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th $USENIX$ Security Symposium ($USENIX$ Security 18). pp. 745–761 (2018)
46. Zalewski, M.: American fuzzy lop (2.52b), `https://lcamtuf.coredump.cx/afl/`, accessed October 26, 2021
47. Zhang, B., Ye, J., Feng, C., Tang, C.: S2f: discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing. In: 2017 13th International Conference on Computational Intelligence and Security (CIS). pp. 548–552. IEEE (2017)
48. Zhang, L., Thing, V.L.: A hybrid symbolic execution assisted fuzzing method. In: TENCON 2017-2017 IEEE Region 10 Conference. pp. 822–825. IEEE (2017)
49. Zhao, L., Duan, Y., Yin, H., Xuan, J.: Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: NDSS (2019)
50. Zhou, S., Yang, Z., Qiao, D., Liu, P., Yang, M., Wang, Z., Wu, C.: Ferry: State-aware symbolic execution for exploring state-dependent program paths pp. 4365–4382 (2022)