Variability-Aware Fuzzing

Meah Tahmeed Ahmed
meah.ahmed@utdallas.edu
Computer Science Department
The University of Texas at Dallas
Richardson, USA

Arnab Dev
arnab.dev@utdallas.edu
Computer Science Department
The University of Texas at Dallas
Richardson, USA

Shiyi Wei
swei@utdallas.edu
Computer Science Department
The University of Texas at Dallas
Richardson, USA

Abstract

Modern software systems often provide a vast configuration space to enhance reusability and adaptability, but this configurability also significantly complicates bug finding. While existing static and dynamic variability-aware analysis approaches systematically explore the configuration space, they often suffer from scalability limitations. Conversely, grey-box fuzzing has demonstrated remarkable success in vulnerability detection through lightweight, iterative input space exploration, yet the state-of-the-art configuration fuzzers overlook the potential of integrating variability-aware analysis within the fuzzing process. In this paper, we present VAFuzz, a novel variability-aware fuzzer that integrates principled dynamic variability-aware analysis within the fuzzing process to enhance configuration space exploration. VAFuzz introduces new variability-aware seed selection and mutations to drive the fuzzing process. These are enabled by a new presence condition seed queue that tracks coverage and crash contributions across the configuration space, and a map that captures the relationship between data seeds and presence conditions. Our evaluation on a diverse set of programs show that VAFuzz outperforms the state-of-the-art configuration fuzzers on 21 out of 25 programs in terms of code coverage. It also detects more vulnerabilities than these baselines, including previous unknown bugs.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Variability-Aware Analysis, Variability-Aware Fuzzing, Automated Test Generation

ACM Reference Format:
Meah Tahmeed Ahmed, Arnab Dev, and Shiyi Wei. 2026. Variability-Aware Fuzzing. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773247

1 Introduction

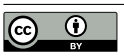
Real-world programs are often built with options to allow easy configuration to improve code reusability. Quality assurance in all configurations is critical to ensure the reliability and security of the whole software codebase. For example, CVE-2021-3156 [36], known as Baron Samedit, is a vulnerability in the sudo program that allows an attacker to gain root access but can only be triggered by running sudoedit with the -s option. Such a bug is called a *variability bug* [2].

While detecting variability bugs is a challenging task, past research has made significant theoretical and practical progress to analyze the configuration space of complex software systems [4, 6, 9–11, 30, 33, 41]. Approaches such as variability-aware analysis [8, 16, 17, 20, 23, 45] and execution [30–32] perform in-depth analysis of the configuration space by associating static and/or dynamic program properties with the presence conditions, aiming to reveal deep bugs in the configurable systems. For example, variability-aware execution [31] explores the behavior of a software system under different configurations by executing the program in a modified variability-aware interpreter and recording the association between the presence conditions and the program states. However, such approaches may incur a large performance overhead due to their nature of performing a thorough analysis of the large configuration space.

In the past decade, fuzz testing, especially grey-box fuzzing [5, 14, 39, 42, 53], has enjoyed great success in vulnerability detection [3]. In general, grey-box fuzzers (GBF) such as AFL [53], AFL++ [14], and LibFuzzer [39], use feedback-driven mechanisms (often code coverage) to guide the exploration of the program input space. Much of the success of GBF is believed to be attributed to its lightweight and iterative nature that allows a large amount of inputs to be explored [28]. However, most research on GBF has focused on how to efficiently explore the program's input data space to reveal vulnerabilities, without considering the large configuration space, which also directly determines the execution space of a program.

More recently, there have been a few efforts that aim to apply fuzzers to detect vulnerabilities considering the configuration space, making them configuration-aware or option-aware [24, 25, 48–50, 54]. For example, ZigZagFuzz [25] introduces mutators specifically for program options and reuse existing fuzzers' mutators for data files to find pairs of data input and configuration as the input corpus to fuzz. It also periodically optimizes this input corpus to reduce redundant pairs. While these fuzzers have produced some promising results, they have not performed systematic variability-aware analysis within the fuzzing process and the proposed mechanisms are loosely coupled with the GBF process, preventing them from reaching the full potential of using GBF to fuzz the configuration space (Section 2).

In this paper, we address the limitations in existing configuration-aware fuzzers by developing a novel *variability-aware fuzzer*, VAFuzz, that performs principled and lightweight variability-analysis of the configuration space and integrates this analysis into an efficient fuzzing process. First, we introduce the variability-aware



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773247

analysis of presence conditions during fuzzing to represent the potential benefit of fuzzing these conditions. In the context of VAFuzz, a presence condition is a set of option settings that are active in a program execution. This analysis is made possible through a lightweight variability-aware execution mechanism (which we call *data-configuration co-execution*). VAFuzz introduces two new data structures to guide the fuzzing process. (1) A *presence condition queue* is used and updated during fuzzing to record the history of how different presence conditions have contributed to the discovery of code coverage and crashes. This allows VAFuzz to understand which presence conditions are more beneficial for fuzzing, and VAFuzz is able to select such presence conditions for further configuration space exploration (2) A map is constructed to record the relationship between data seeds and presence conditions (i.e., under a condition which data seeds have been effective). Second, these data structures, as the result of our variability-aware analysis, are integrated in all essential stages of the GBF process. We introduce *presence condition selection and mutations* to facilitate the generation of related configurations for fuzzing. The *data association map* is used to facilitate *data seed favoring* to effectively select data seeds when a presence condition is selected for fuzzing. Overall, this in-process variability-aware analysis and highly integrated approach allows VAFuzz to explore the full potential of GBF when fuzzing configurable systems.

We implemented VAFuzz on top of AFL++ by adding 13,959 new lines of code. In our evaluation, we compared VAFuzz to three baseline fuzzers that target program configurations (ZigZagFuzz [25], ConfigFuzz [54], and AFL++-argv [14]) and five sampling baselines. Our results on 25 targets show that VAFuzz outperforms the baselines in terms of code coverage in 21 out of the 25 programs and is able to find 12 bugs, 7 of which could not be found by the baselines, including 2 previously unknown bugs. We also perform ablation study to confirm that components of VAFuzz collectively contribute to its effectiveness. This paper makes the following contributions:

- We identify and address key limitations of existing configuration fuzzing by introducing an in-process variability-aware analysis during fuzzing.
- We design and implement VAFuzz, which fully integrates the results of variability-aware analysis with two new data structures that drive and revamp the fuzzing process, and introduces new selection and mutation strategies for presence conditions.
- We evaluated VAFuzz on diverse programs and show that it outperforms the state-of-the-art baselines in both code coverage and bug finding capability.

We made the VAFuzz source code, evaluation setup, and results available here: [1].

2 Background and Motivation

A configurable software systems contains a set of *options*, used by developers to customize the behavior of the software. These options may be enabled, disabled, or set to different values. When an option is set to a specific setting, we refer it as an *option setting*; the combination of these option settings is called a *configuration*. The *configuration space* of a program is the set of all possible configurations. As the number of configurations of a program may be extremely large due to combinatorial explosion, it is challenging to test the configuration space to detect bugs that may only exist under some configurations (called *variability bugs*). There have been recent attempts to detect variability bugs with fuzzing. We categorize these efforts into two main directions.

First, several works aimed at *integrating the exploration of configuration space within the data fuzzing process* [24, 25, 48–50, 54]. We call a fuzzer *configuration-aware* when it is able to modify the configuration of the target program based on some feedback or information. We believe an evolution of this idea is to use variability-aware analysis to guide the fuzzing process. Such fuzzers can be termed as *variability-aware fuzzers*. The idea of using fuzzing to explore the configuration space of a program was first proposed by AFL [53] with the introduction of the *AFL-argv* feature. AFL-argv reads from the standard input (stdin) and injects the bits generated by the fuzzer into the program's argv array. While this allows AFL to access the configuration space, and mutate the program's configuration, it has shown to be inefficient in practice, as it does not have any context on what a valid configuration is, and relies on the target program to reject invalid configurations. ConfigFuzz [54] takes the idea of AFL-argv further by using a configuration grammar that represents program options and their possible settings and relationship. This allows ConfigFuzz to generate valid configurations that are then passed to the target program. POWER [24] introduces a new mutator that is specifically designed to mutate program options. Using this mutator, POWER first fuzzes configurations to find the most promising ones. Once these configurations are found, POWER then switches to fuzzing the data input of the program as a traditional fuzzer would but with the selected configurations. ZigZagFuzz [25] extends POWER, introducing an interleaving mechanism that periodically optimizes the input corpus to reduce redundant pairs of data input and configuration and explore new pairs. Note that all the fuzzers discussed above use existing GBF's feedback mechanism to guide the exploration of the configuration and data space. This is a limitation of these fuzzers, as the feedback is not designed to capture the complex relationships between program options and their impact on the fuzzing process. Another limitation is that these fuzzers do not perform variability-aware analysis to distinguish which groups of option settings have a significant and different impact on the fuzzing behavior.

Second, the other direction focuses on *analyzing and extracting the program options and their combinations, and use this information to guide fuzzing*. CarpetFuzz [48] aims to understand the relationship between program options and their combinations using Natural Language Processing (NLP). It extracts relationships between program options from the documentation of the program and utilizes this information to generate configurations for fuzzing. ProphetFuzz [49] extends CarpetFuzz and uses Large Language Models (LLMs) to extract high risk configurations and assemble commands from the documentation of the program. Instead of relying on the documentation, OSmart [50] uses a white-box approach to extract the relationships between program options and represents them in an option impact graph. While these approaches extract relationships between program options, they also do not perform variability-aware analysis to understand the impact of configurations on fuzzing. Additionally, they do not leverage runtime

feedback as they generate all configurations before the fuzzing campaign. This means that they do not adapt to the runtime behavior of the program, which leaves opportunities for improvement.

The Software Product Line (SPL) literature has developed black-box sampling strategies to mitigate configuration explosion [29]. Approaches like one-enabled, one-disabled, most-enabled-disabled, random sampling, and t-wise (e.g., pairwise and three-wise) testing [4] select subsets of configurations to maximize coverage while assuming access to feature models or documented constraints. Diversity-based sampling [52] further aims to maximize differences between configurations (e.g., using Hamming distance [35]) to expose variability faults. While effective for reducing the configuration space, these methods often lack runtime feedback and dynamic adaptation, limiting their use in fuzzing. Our work builds on these ideas by integrating configuration exploration within a feedback-driven fuzzing loop, leveraging the lightweight, iterative nature of grey-box fuzzing while systematically targeting impactful configurations. Additionally, the SPL literature has also developed white-box techniques using program analysis and symbolic execution to prune configurations while ensuring variability-related coverage [18, 19, 40, 43, 44]. These methods require feature models and access to source code, and are often computationally expensive, making them less practical for dynamic fault discovery using fuzzing. In contrast, our feedback-driven fuzzing approach explores impactful configurations without requiring prior models or static analysis. Note that these black-box and white-box approaches have explored both compile-time and run-time configurations; our work, similar to the configuration-aware fuzzers discussed above, focuses on run-time configurations. Integrating compile-time configurations in fuzzing poses new challenges (e.g., efficiency of compilation and feedback mechanism), which we plan to explore in the future.

3 VAFuzz Approach

To address the challenges discussed above, we design VAFuzz, which integrates variability-aware analysis directly into the fuzzing process. Figure 1 illustrates the overall workflow. VAFuzz augments traditional fuzzing by iteratively exploring both the configuration space and the input space. This process is guided by two key data structures. First, the *Presence Condition Queue* (top-right) is a ranked list that tracks each presence condition's contribution to unique code coverage. Variability-aware analysis computes a *priority value* for each condition, which is used to update the queue. Second, the *Data Association Mapping* (top-center) records, for each presence condition, the set of data files that have previously yielded new coverage under that condition. Together, these structures capture run-time configuration–coverage interactions and guide the selection and mutation of both presence conditions and data files throughout the fuzzing campaign.

As shown in Figure 1, similar to traditional grey-box fuzzers, VAFuzz takes the target program and the initial data seed corpus as inputs. Note that because VAFuzz supports fuzzing both the data space and the configuration space of the input program, we explicitly refer to the program inputs as data seeds/files. In addition, VAFuzz takes the *configuration grammar* of the target program as an input. Specifically, the JSON-based configuration grammar specifies possible option values along with cross-tree constraints. This grammar is manually constructed by the user based on the program's documentation and by tracing the argument parsing in the program's source code. This process took us 5-10 minutes for each target program. The configuration grammar and a data input are sufficient to run VAFuzz. Using the configuration grammar, VAFuzz first *generates a set of initial configurations* (Section 3.1). These configurations (or in later iterations of VAFuzz, the configurations generated by the presence condition mutation) are executed by the *Data-Configuration Co-Execution* component (Section 3.2). This component resembles the existing GBF's fuzzing process where data seeds are selected, mutated, and executed. In VAFuzz, however, each mutated data file is executed under all the configurations currently being fuzzed. The idea to perform this co-execution is that the results from running these related configurations on the same set of mutated data files can reveal the common properties these executions hold under certain presence conditions (e.g., which presence conditions are responsible for covering which branches).

Once the data-configuration co-execution component exhausts the assigned number of executions of the presence condition being fuzzed, the *coverage traces* produced from the executions are analyzed in the *Variability-Aware Feedback Analysis* component (Section 3.3). The high-level idea of the *Variability-Aware Presence Condition* analysis is to find unique code segments that are likely to be contributed by fuzzing each presence condition. The results of this analysis are used to compute the priority values of the analyzed presence conditions to add into or update the presence condition queue. The *Variability-Aware Data Seed Analysis* reveals which data files allowed covering new code segments under a presence condition, and are used to update the *Data Association Mapping*. The *Data Seed Favoring* part ensures that data seeds that have shown to be effective under a selected presence condition are favored, based on the data association mapping.

The *Presence Condition Selection and Mutation* component decides which presence condition to fuzz and its number of executions, and we design three mutation operators to generate related configurations (Section 3.4). These strategies are designed to efficiently fuzz the presence conditions that are likely to lead new code coverage. In addition, VAFuzz applies a clustering algorithm to reduce the presence condition queue if it exceeds a threshold, to optimize memory utilization (Section 3.5).

This overall process allows VAFuzz to infer relationships between different options, and between options and data files, key to efficiently exploring the configuration space. We now describe the design of VAFuzz's components.

3.1 Initial Configuration Generation

The input configuration grammar of VAFuzz specifies the set of options and their possible settings in the target program. VAFuzz supports four types of options. The *Boolean* option has two possible settings: True and False; the possible settings of the *Choice* option are explicitly defined as categorical values in the grammar. The ranges of the *Integer* and *Real* options are also explicitly defined. This grammar can be constructed by inspecting the target program's documentation and/or by tracing the implementation.

The *Initial Configuration Generation* step generates configurations with a single option being set, while every other option is

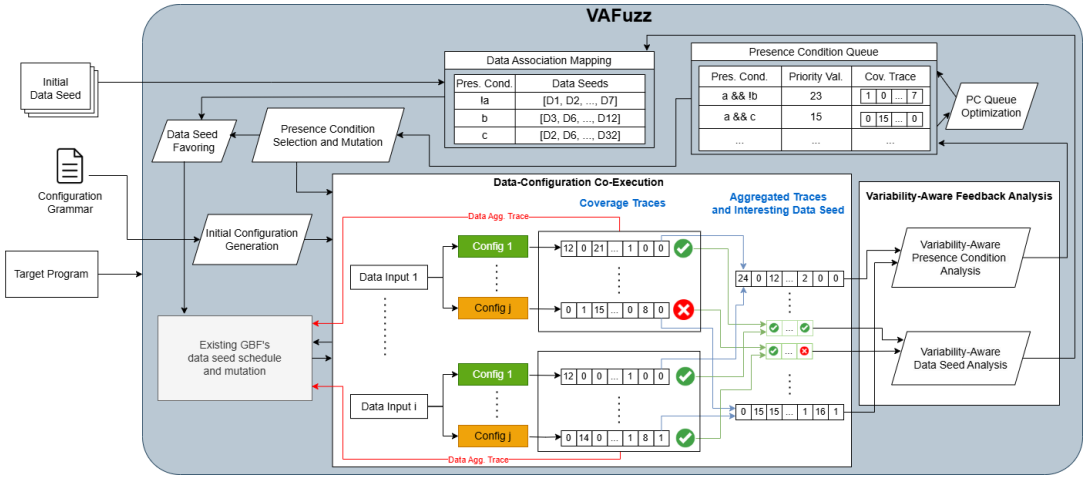


Figure 1: VAFuzz workflow.

absent or turned off. We call this strategy one-enabled. For Boolean and Choice options, VAFuzz generates an initial configuration for each possible setting in these options. For Integer and Real options, VAFuzz randomly samples three values within the specified range. We chose to generate this initial set of configurations because they broadly cover the configuration space (with little interaction between the options) which serves as a primer for the fuzzer to start exploring the configuration space. In the case this initial set of configurations is large (i.e., greater than 50), VAFuzz randomly samples 50 configurations from the generated set. We have empirically tested the sample sizes of 15, 30, 50, 100, and 150, and found that there is no significant difference in code coverage but the execution speed slows down when the sample size exceeds 50.

3.2 Data-Configuration Co-Execution

Inspired by the idea of variability-aware analysis [8, 16, 16, 17, 20, 23, 45] and execution [30–32], where the program properties are associated with propositional formula (e.g., presence conditions), we design the data-configuration co-execution component of VAFuzz. This component explores multiple concrete executions of the program configurations in parallel. To overcome the scalability limitations of variability-aware analysis and execution which stem from the need to account for all possible concrete executions in the program's configuration space, VAFuzz selectively executes a set of related configurations in each iteration to allow the variability-aware feedback analysis (Section 3.3) to efficiently analyze the effect of presence conditions. The data-configuration co-execution component is integrated into the grey-box fuzzing process seamlessly to retain the benefit of effectively fuzzing data files.

Algorithm 1 outlines the data-configuration co-execution process in VAFuzz. This algorithm takes the following inputs. (1) A set of configurations, C. These configurations are generated either by the initial configuration generator (Section 3.1) at the start or by the presence condition mutation (Section 3.4.1). In Figure 1, these configurations are illustrated as Config 1 to Config j (green and orange boxes). (2) A set of data seeds, D. Initially, VAFuzz uses the input data seed corpus. During fuzzing, these seeds are updated

Algorithm 1: Data-Configuration Co-Execution

Input: Configurations $C = \{c_1, \ldots, c_n\}$; data seeds $D = \{d_1, \ldots, d_m\}$; global coverage bitmap b_g; presence-condition priority $priority_{pc}$
Output: Aggregated traces T; favored data I

$b_{cur} \leftarrow b_g$
GBF.updateDataQueue(D)
$totExecs \leftarrow$ calculateTotalExecs($priority_{pc}$)
while $totExecs > 0$ **do**
  $d \leftarrow GBF$.getNextData()
  $t_{tmp} \leftarrow$ InitializeTrace()
  for $c_i \in C$ **do**
    $t_{(c_i,d)} \leftarrow$ Execute(c_i, d)
    T.put(c_i, T.get(c_i).aggregate($t_{(c_i,d)}$))
    if $GBF.IsInteresting(t_{(c_i,d)}, b_{cur})$ **then**
      I.put(c_i, I.get(c_i).add(d))
    $t_{tmp} \leftarrow t_{tmp}$.aggregate($t_{(c_i,d)}$)
  $b_{cur} \leftarrow GBF$.updateTraceAndDataQueue(t_{tmp})
  $totExecs \leftarrow totExecs - 1$
return (T, I)

based on which presence conditions are being fuzzed, determined by the data seed favoring component (Section 3.4.2). (3) A global coverage bitmap, b_g, which is a bitmap representing the count of the number of times a program branch that has been executed by VAFuzz. This is the same data structure used in popular fuzzers such as AFL++ [14]. (4) A priority value for the presence condition, $priority_{pc}$. This value is determined by the Presence Condition Mutation (Section 3.4.1) and its default value is set to 1 for the initial set of configurations.

Algorithm 1 returns the *aggregated traces* and *favored data* as outputs. Each entry of the aggregated traces T maps a configuration c_i to the coverage trace t_i that aggregates the counts of branch executions on all the data executed under c_i during the data-configuration co-execution. Each entry of the favored data maps c_i to a set of data inputs deemed interesting when executing them under c_i. These

traces as well as the favored data sets are shown in Figure 1 under *Aggregated Traces and Interesting Data Seed.*

Line 1 initializes b_{cur}, which we keep to record the current snapshot of the global coverage bitmap, with the input global bitmap b_g. Line 2 updates the favored seeds in the data seed queue of the GBF. Note that the data-configuration co-execution is tightly coupled with the existing GBF's mechanisms. In this algorithm, an interaction with these mechanisms is presented as a call to GBF. Line 3 calculates the total number of executions for each configuration, *totExecs*, using the following equation:

$$\text{totExecs} = min(priority \times N, M) \tag{1}$$

In Equation 1, N is a constant $\geq$ 1 and M is a large constant defining the upper bound. This is similar to the power schedules [5] defined in GBF, but ensures a selected presence condition with higher priority is fuzzed longer. In our implementation, we set the tunable parameters N to 5000 and M to 100,000 after empirically exploring the tradeoffs.

Lines 4-14 detail the data-configuration co-execution logic. This execution continues until the *totExecs* is exhausted (line 4). In each iteration, the GBF is queried to obtain the data file d (line 5). This query reuses GBF's seed selection, scheduling, and mutation strategies to generate data inputs. An important idea of the data-configuration co-execution is that executions of a data input under a set of configurations are considered in parallel, instead of in sequence (which GBF and existing configuration-aware fuzzers implement) where the previous execution's result affects the assessment if the current execution result is interesting. This parallel execution mechanism mimics the variability-aware execution, enabling our variability-aware feedback analysis to efficiently analyze the impact of the presence conditions on the execution traces. Therefore, in line 6, we initialize an empty trace t_{tmp} to track the coverage results running d on all the configurations in C. Then for each configuration c_i in C (line 7), we execute d under c_i and return its coverage trace $t_{(c_i,d)}$ (line 8), illustrated under *Coverage Traces* in Figure 1. This trace is then used to update the c_i entry of the aggregated traces in line 9. Specifically, the trace mapped from c_i in T (empty at the start of this algorithm) is aggregated by adding any additional counts of the number of times a program branch executes. We can see an illustrative example of this aggregation in Figure 1 where the traces obtained from data inputs 1 and i are aggregated (blue lines) to produce the aggregated trace for each configuration. If this execution results in a trace is considered interesting (which typically means discovery of new coverage or crash) by the GBF, compared to the current snapshot of the global coverage bitmap b_{cur}, the data input d is added to the favored data of c_i (lines 10-11). The interestingness of the traces is depicted in Figure 1 as green tick marks or red crosses. Line 12 updates t_{tmp} by aggregating $t_{(c_i,d)}$. Once the executions of d under all the configurations in C complete, the trace aggregated over all these executions of d, t_{tmp}, is used to update the global coverage bitmap and data queue kept by the GBF and b_{cur} is updated (line 13). As illustrated in Figure 1, the aggregated traces from all configurations are sent to the GBF via the *Data Agg. Trace* arrows.

Algorithm 2: Presence Condition Feedback Analysis

Input: Configurations $C = \{c_1, \ldots, c_n\}$; aggregated traces $T = \{c_1 \rightarrow t_1, \ldots, c_n \rightarrow t_n\}$; presence-condition queue Q_{pc}; global bitmap b_g; optional presence condition pc
Output: Updated presence-condition queue Q_{pc}

1 $weights_g \leftarrow getBitWeights(b_g)$
2 **foreach** $c_i \in C$ **do**
3 $\quad t_i \leftarrow T.get(c_i)$
4 $\quad priority \leftarrow calcPriority(t_i, weights_g)$
5 $\quad Q_{pc}.update(genPC(c_i) \rightarrow (priority, t_i))$
6 **if** *pc exists* **then**
7 $\quad OS \leftarrow getAllOptSettings(pc)$
8 $\quad$ **foreach** $os_i \in OS$ **do**
9 $\quad\quad T_{present} \leftarrow getPresentTraces(os_i, T, C)$
10 $\quad\quad T_{absent} \leftarrow T \setminus T_{present}$
11 $\quad\quad t_{pres\_va} \leftarrow getVATrace(T_{present})$
12 $\quad\quad t_{abs\_va} \leftarrow getVATrace(T_{absent})$
13 $\quad\quad priority_{pres} \leftarrow calcPriority(t_{pres\_va}, weights_g)$
14 $\quad\quad priority_{abs} \leftarrow calcPriority(t_{abs\_va}, weights_g)$
15 $\quad\quad Q_{pc}.update(genPC(os_i) \rightarrow (priority_{pres}, t_{pres\_va}))$
16 $\quad\quad Q_{pc}.update(genPC(os_i, neg) \rightarrow (priority_{abs}, t_{abs\_va}))$
17 **return** Q_{pc}

3.3 Variability-Aware Feedback Analysis

The feedback analysis component of VAFuzz processes the results obtained from the data-configuration co-execution stage and uses them to associate presence conditions with (1) their *priority values* which predict the likelihood of resulting in new coverage if fuzzed, and (2) the *favored data seeds* that should be used if the corresponding presence conditions are fuzzed. The feedback analysis is divided into two parts, presence condition feedback analysis and data association mapping, which we describe below.

3.3.1 Presence Condition Feedback Analysis. There are two goals to this analysis: (1) identify the branches that are impacted by specific presence conditions, and (2) assign a priority value to each presence condition based on the impact of the presence condition on the execution traces. Algorithm 2 outlines the *presence condition feedback analysis* process. The algorithm takes the following inputs. (1) The set of configurations C used in the data-configuration co-execution. (2) The aggregated traces T obtained from the co-execution. (3) The global coverage bitmap b_g which is updated by the co-execution. (4) An optional presence condition pc that was selected to mutate and generate the configurations C. This is optional because not all mutators use a presence condition (Section 3.4.1). (5) The presence condition queue Q_{pc}. This queue stores the presence conditions that are generated by the feedback analysis. The presence conditions are ranked based on a *Priority Value* which is computed by this analysis. For each presence condition, the queue also stores an associated trace, representing which bits may be triggered by this presence condition. Algorithm 2 updates this queue with new presence conditions, their priority values and associated traces, providing the fuzzer with the information of which presence conditions to select for fuzzing.

To compute the priority value mentioned above, VAFuzz first needs to understand the importance of triggering each bit. This is so that the fuzzer can prioritize bits that are less frequently triggered by selecting presence conditions that are more likely to trigger these bits. In line 1 of Algorithm 2, we utilize b_g to calculate the weight of each bit in the global bitmap and store them in $weights_g$.

$$weights_g[i] = \frac{b_g[i]}{255}, \quad \forall i \in \{0, 1, \ldots, |b_g| - 1\} \tag{2}$$

Equation 2 shows how the weight of each bit is calculated in our implementation. In AFL++ [14], which VAFuzz is built on top of, the global bitmap is initialized with a value of 255 for each bit and for every execution, the value is decremented by 1. Once the value reaches 0, the bit is considered to be very frequent and is no longer decremented. Hence, the range of the count for each bit is between 0 and 255. To calculate the weight of each bit, the number of times this bit is triggered is divided by 225. Therefore, we assign a weight between 0 and 1 for each bit, where 1 represents a bit that has not been reached and 0 represents a very frequently reached bit.

The feedback analysis then starts to iterate over each configuration in C (line 2). For each configuration c_i, the algorithm retrieves the mapped aggregated trace, t_i, from T (line 3). Line 4 calculates the priority value for c_i.

$$\text{calcPriority}(t, w) = \left\lceil \frac{\sum_{i=0}^{|t|-1} isReached(t[i]) \cdot w[i]}{|t|} \times 100 \right\rceil \tag{3}$$

where

$$isReached(t[i]) = \begin{cases} 1, & \text{if } t[i] \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

In Equation 3, the priority value is calculated by summing the weights of the bits (of the global bitmap) that are reached in the aggregated trace t_i and then dividing by the total number of bits. Therefore, the priority value is between 0 and 100 with higher value indicating the importance of the bits covered by the corresponding trace. Line 5 updates the presence condition queue Q_{pc}. We generate a presence condition from the configuration c_i, which introduces the logical *AND* operation among all the option settings in the configuration. The entry of this presence condition, if it exists in Q_{pc}, is updated with the priority value and the aggregated trace; otherwise, a new entry is added. In our implementation, we use the Z3 [12] constraint solver to efficiently reason about presence conditions. We use Z3 to compare the equivalence of the presence condition to existing presence conditions in Q_{pc} to avoid duplicates in the queue.

Once each configuration has been analyzed, the algorithm then checks to see if there is an optional presence condition, pc, that was used to generate the configurations (line 6). If pc is present, we perform additional feedback analysis. The algorithm extracts all the option settings present in pc and stores them in a set, OS (line 7). We then iterate over each option setting os_i available in OS. For each os_i, we find each configuration from C where os_i is present. We then store the aggregated traces from T for these configurations in $T_{present}$ (line 9). Hence, $T_{present}$ represents the traces of the configurations where the option setting os_i is present. We also store the aggregated traces from T for the configurations where

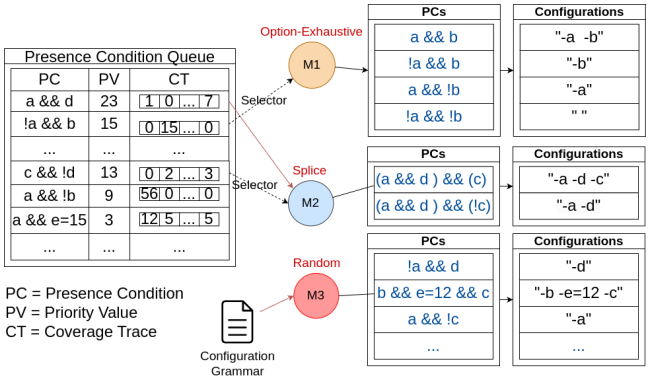


Figure 2: Presence condition mutation example.

os_i is absent in T_{absent} (line 10). Using the two sets of aggregated traces, we estimate which bits are only impacted by the presence and absence of the option setting os_i, resulting in variability-aware traces (lines 11-12).

$$\text{getVATrace}(T) = \Big\lfloor \min\{T[i][j] \mid i \in \{1, 2, \ldots, |T|\}\}, \quad \forall j \in \{0, \ldots, |T[0]| - 1\} \Big\rfloor \tag{4}$$

Equation 4 shows the *getVATrace* function: for each bit, we take the minimum value of the bit in the set of traces. With these variability-aware traces, we calculate priority values using Equation 3 for the present and absent traces (lines 13-14). Finally, we generate two presence conditions, one representing the option setting os_i and the other representing the absence of os_i. These presence conditions, along with the priority values and the aggregated traces, are updated in the presence condition queue (lines 15-16).

3.3.2 Data Association Mapping. The data association mapping is responsible for tracking the relationship between presence conditions and data seeds. The goal is to select suitable data seeds for each presence condition when it is fuzzed. Our insight is that a change of the presence condition may change the context of which data file is interesting. We designed the data-configuration co-execution to retain the information (i.e., the favored data I output from Algorithm 1) needed to establish such an association

Our implementation of the data association mapping associates each presence condition, pc, with a set of data seeds, D. Empirically, we have seen that this mapping is the most effective when the presence condition contains a single option setting. Therefore, for each configuration c_i, we look up I and find its favored data seeds from the data-configuration co-execution. Then for each option setting os_j in c_i, these data seeds are added to the set mapped from os_j to a list of data seeds in the data association mapping.

3.4 Presence Condition Selection and Mutation, and Data Seed Favoring

We now discuss how VAFuzz selects and mutates presence conditions to generate a set of configurations, and how the corresponding favored data seeds are decided.

3.4.1 Presence Condition Selection and Mutation. We define three *presence condition mutators*, $M1 - M3$, with the goal of producing diverse sets of valid configurations. These mutators are randomly chosen with customizable weights. In our implementation, the probabilities of choosing M1, M2, and M3 are 30%, 30%, and 40%.

M1: Option-exhaustive. This mutator generates configurations based on a single presence condition *pc*. The presence condition is selected from the presence condition queue, either based on the priority value (i.e., selecting the presence condition with the highest priority value) or randomly. The reason we sometimes want to select a random presence condition is to improve diversity where some presence conditions and their associated options are well fuzzed while the other options are under-explored. We allow customizing the probability of performing the random presence condition selection. In our current implementation, we set the presence condition selection strategy to have 70% probability of using the priority value; otherwise, perform random selection.

With the selected presence condition, M1 then extracts all the options used in *pc*. The mutator generates the presence conditions by considering all possible combinations of the settings in the extracted options. For Integer and Real options, the mutator uses the setting in the selected presence condition and also generates two random values within the ranges specified in the configuration grammar. The key insight of M1 is to generate a diverse but related set of configurations, including not only those satisfying the selected presence condition that is likely to produce new interesting results (based on the priority value), but also those that allow exploring around the presence condition (i.e., combinations of settings that use the same options). To avoid too many configurations to be executed by the data-configuration co-execution step, all three mutators mutator will sample a total of 50 configurations if the generated set exceeds 50. This limit is set based on empirical observation (similar to Section 3.1) that exceeding this threshold does not significantly improve coverage but increases runtime. We note that this is based on the results of the target programs used in our experiments, and may not hold for all programs.

Figure 2 shows examples of the mutators. As shown, M1 may randomly select the presence condition *!a & b*. The mutator then extracts the options *a* and *b* from the presence condition and generates configurations by considering all possible combinations of the settings of these options. This results in the four presence conditions in Figure 2 (linked from M1) and their corresponding configurations.

M2: Splice. This mutator generates configurations based on two presence conditions, pc_1 and pc_2, with the goal of combining them to form configurations that involve more options. In M2, the first presence condition pc_1 is always the presence condition with the highest priority value in the presence condition queue. The second presence condition pc_2 is selected using the same presence condition selection strategy described in M1 (i.e., the presence condition with the second highest priority value or randomly).

M2 first extracts the distinct options used in pc_2 but not in pc_1, and generates all possible combinations of the settings in the extracted options, similar to M1. pc_1 is then appended to each of the generated presence conditions through a logical *AND* operation. The resulting configurations are then generated based on the new presence conditions. The idea is that we have seen that pc_1 performs well and we are interested to know how it performs when combined with other presence conditions. In Figure 2, the option used in pc_2 but not in pc_1 is *c*, so the presence conditions mutated from M2 are *a & d & c* and *a & d & !c*. Recall that the feedback analysis may accept a presence condition as input. For M2, the merged presence condition (e.g., *a & d & c*) is used.

M3: Random. This mutator generates configurations directly from the configuration grammar, instead of taking any presence condition(s) as input. Specifically, M3 generates configurations by randomly selecting options and their settings from the configuration grammar. Traditionally, randomness in mutation, such as AFL++'s *Havoc* stage [14], can yield strong fuzzing results [51]. M3 is designed to mimic that randomness. The goal of this mutator is to introduce new presence conditions to our presence condition queue that may not be reached by the other mutators. As shown in Figure 2, M3 utilized the configuration grammar to randomly generates configurations like *"-d"*, *"-b -e=12 -c"*, and *"-a"*.

Recall that these mutators determine the priority value input to Algorithm 1. As M1 uses one presence condition, the priority value of that presence condition is used. For M2, the maximum priority value of the two presence conditions is used. For M3, the priority value is user-defined (default set to 5000 based on our experiments).

3.4.2 Data Seed Favoring. After generating a set of configurations, VAFuzz uses the data association mapping to select data seeds from the entire data queue for fuzzing during data-configuration co-execution. This mapping associates each option setting with a set of relevant data seeds. VAFuzz takes the union of these sets based on the selected option settings to determine which data seeds to prioritize. For M2, the merged presence condition guides which data seeds are favored. These prioritized data seeds are then used in the data-configuration co-execution step, ensuring that the fuzzer focuses on data inputs from the queue that have been shown to be effective for the current presence condition.

3.5 PC Queue Optimization

One practical challenge of using the presence condition queue is that as VAFuzz continues adding the presence condition entries into the queue, it may become large such that operations such as updating and selecting from the queue become inefficient. To efficiently manage the presence condition queue, we develop an optimization that reduces the size of the queue when it exceeds a threshold after the variability-aware feedback analysis updates the queue. Overall, this optimization first conducts a coverage-based analysis of the traces in the queue and then performs clustering to remove entries similar in their coverage traces.

Specifically, our approach first identifies traces for each presence condition containing unique coverage. These entries are preserved in the optimization. This ensures that no critical program coverage is lost during the optimization. Second, we process the remaining traces in the queue using k-means clustering [34], grouping similar coverage traces together. The number of clusters is determined based on the diversity of the traces, with the goal of producing the number of clusters close to half the number of remaining traces. A representative of each cluster is selected.

In our implementation, we set the threshold to trigger this optimization to be 50 presence condition queue entries. The impact of

this optimization is clear, with the fuzzing campaigns of several program hanging without it. For example, fuzzing xmllint [46] without this optimization has led to the analysis component to take over 10 minutes to analyze the presence condition queue each time.

4 Evaluation

In this section, we evaluate the performance of VAFuzz, answering the following research questions:

- **RQ1:** How effective is VAFuzz compared to state-of-the-art configuration-aware fuzzers?
- **RQ2:** How do the individual components of VAFuzz contribute to its overall performance and do these components produce any overhead?

4.1 Experimental Setup

Implementation. We implemented VAFuzz on top of AFL++ 4.04c by adding 13,959 new lines of code. Specifically, we modified the *common_fuzz_stuff()* function to integrate our co-execution mechanism. Additionally, we modified the AFL++ instrumentation to enable the change of the argument vector (argv) of AFL++'s forkserver. To enable the feedback analysis and presence condition mutations we used Python, Z3, and the *Python.h* library to interact with AFL++. To ensure fairness, all approaches were evaluated running on a single thread. We simulate parallel execution on VAFuzz by recording coverage traces separately for each configuration-input pair, delaying bitmap updates until all executions for a data file complete. This avoids interference and ensures isolated evaluation.

Baseline fuzzers. We compared VAFuzz against three configuration-aware fuzzers: ZigZagFuzz [25], ConfigFuzz [54], and AFL++-argv [14]. As discussed in Section 2, both ZigZagFuzz and ConfigFuzz attempt to fuzz configurations throughout the fuzzing campaign, which are the most relevant baselines to our approach. We believe AFL++-argv (*Argv*) is a good baseline to compare against, as it adds configuration awareness to AFL++, which our implementation is based on. In addition, we evaluated five black-box sampling baselines reflecting common SPL strategies [33]. For fair comparison, we integrated these baselines with AFL++ adopting the same strategy introduced in ConfigFuzz [54]: allocating each configuration an equal share of a 24-hour budget while reusing the seed queues across the configurations. The baselines are: *most-enabled (ME)* and *most-disabled (MD)* use a single configuration with all options enabled or disabled, respectively; *random sampling (Rand)* and *diversity-based sampling (DB)* generate 50 valid configurations (with DB maximizing Hamming distance [35]); and *pairwise sampling (2W)* generates configurations using a two-way covering array [4]. A preliminary experiment confirmed that varying the number of configurations in *Rand* and *DB* did not significantly impact coverage, so we fixed it at 50. Unlike VAFuzz, which adapts configurations using runtime feedback, the orders of fuzzing the configurations in these sampling baselines are pre-determined.

Other than these baselines, POWER [24], OSmart [50], and CarpetFuzz [48] are other recent configuration-aware fuzzers. We did not include POWER because ZigZagFuzz was developed and improved based on POWER and ZigZagFuzz's evaluation [25] demonstrated consistent improvement over POWER. We did not empirically compare to OSmart [50], and CarpetFuzz [48] because: (1) the improvements made in OSmart and CarpetFuzz focused on extracting options and their relationships before the fuzzing campaign, which is complementary to VAFuzz's contributions; (2) we were unable to get OSmart running despite of the effort of communicating with the developers and CarpetFuzz could not parse the documents of many target programs that we used in our evaluation.

Target Programs. We target a diverse set of 25 programs, particularly those with large configuration space. Columns 1-2 in Table 1 show the target programs and their numbers of options. These programs have been commonly used in fuzzing research. For each program, we manually traced through the source code and the documentation to identify all the configuration options and build the configuration grammar that is provided to VAFuzz and ConfigFuzz. In the case of ZigZagFuzz, we used this grammar to construct a list of options that ZigZagFuzz takes as input. Note that while we have put in significant effort to ensure that the grammar files are complete, there may still be some options that we may have missed as the documentation of some programs may not be up to date, and tracing through the source code can be challenging.

Metrics. We assessed the fuzzers using two metrics: (1) *Code coverage:* This was done by collecting edge coverage information using *AFL-showmap*. After the completion of each experiment, we replay the generated test cases to collect the coverage information. (2) *Bug finding capability:* To evaluate the effectiveness of each fuzzer in detecting vulnerabilities, we collected and manually analyzed all crashes. Each crash was classified based on its root cause (using debugging tools such as `gdb` [15] and `ASan` [22]), and duplicates were filtered to ensure that only unique bugs were counted.

Experimental environment. To ensure fair comparison and reproducibility, we encapsulated each target program–fuzzer pair in a separate Docker [13] container assigned to an isolated CPU core, allowing concurrent execution without interference. Each fuzzer ran for 24 hours, repeated over 5 trials to account for randomness [21, 38]. Experiments were performed on a server with an AMD Ryzen Threadripper PRO 5975WX (32 cores, 64 threads, 3.6 GHz) and 128 GB RAM, running Ubuntu 22.04.

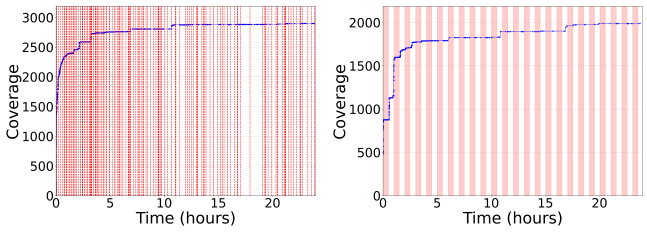
4.2 RQ1: Coverage and Bug Detection

4.2.1 Code Coverage. The C columns in Table 1 show the code coverage results for VAFuzz and the baseline fuzzers. Overall, we observe that VAFuzz outperforms all the baselines in 21 out of 25 target programs. For all target programs, VAFuzz achieved an average coverage of 2366 edges; ZigZagFuzz, ConfigFuzz, and AFL++-argv achieved average coverages of 2053, 1686, and 625, respectively. The sampling baselines, ME, MD, Rand, DB, and 2W achieved average coverages of 1512, 1173, 1327, 1275, and 1282, respectively.

VAFuzz performed exceptionally well on certain target programs, such as djpeg, where it performs 31% better than the next best, and jpegtran, where it outperforms the next best by 134%, and nasm and pdfinfo where it is better by 26% and 46%, respectively. To understand these significant improvements, we performed a deep investigation of the results of jpegtran and djpeg. Figure 3 shows the coverage overtime plots for VAFuzz and ZigZagFuzz on djpeg. We visualize the time periods ZigZagFuzz spent on mutating configurations (red shaded regions in Figure 3b) and the time periods

Table 1: Coverage (C) and bug detection (B) results for different fuzzers across the target programs.

Program	Opts	VAFuzz		ZigZagFuzz		ConfigFuzz		Argv		DB		Rand		2W		ME		MD	
		C	B	C	B	C	B	C	B	C	B	C	B	C	B	C	B	C	B
djpeg	22	2621	1	2002	0	448	0	420	0	1771	0	1971	0	1571	0	1980	0	1492	0
cjpeg	12	4162	0	3680	0	3105	0	932	0	794	0	602	0	575	0	1559	0	1143	0
jpegtran	10	5292	0	2254	0	477	0	340	0	1020	0	1327	0	1250	0	2200	0	1302	0
fax2ps	15	2275	1	1903	1	2075	1	115	0	2005	0	1932	0	1954	0	2192	0	1835	0
fax2tiff	9	1586	0	1485	0	1463	0	540	0	375	0	728	0	703	0	1048	0	821	0
tiff2pdf	13	2087	0	1592	0	1968	0	118	0	1603	0	1101	0	1204	0	1391	0	1075	0
tiff2ps	14	1522	0	1380	0	648	0	154	0	1148	0	870	0	932	0	1239	0	1179	0
tiffcp	16	2648	0	207	0	2335	0	237	0	1114	0	1394	0	1184	0	1521	0	1050	0
tiffcrop	8	1564	0	1797	0	1578	0	171	0	1605	0	1541	0	1482	0	1601	0	1311	0
tiffinfo	10	1240	0	1206	0	1043	0	124	0	1174	0	1159	0	1098	0	1137	0	1043	0
gif2png	3	414	2	413	2	321	2	318	2	213	2	207	2	203	2	357	2	320	2
nasm	5	4268	3	3388	0	3739	1	2342	1	950	0	892	0	832	0	1287	0	750	0
ndisasm	4	877	1	808	0	833	0	498	0	401	0	432	0	412	0	464	0	385	0
nm	2	291	0	287	0	213	0	205	0	203	0	97	0	112	0	256	0	101	0
objdump	7	2952	0	2625	0	1147	0	979	0	712	0	1713	0	1503	0	1434	0	1302	0
readelf	18	4924	0	5270	0	5402	1	1103	0	4875	0	4888	0	4759	0	4937	0	4136	0
size	1	151	0	150	0	110	0	131	0	130	0	132	0	130	0	135	0	121	0
pdfimages	8	1807	1	1379	0	1486	0	498	0	320	0	315	0	302	0	274	0	251	0
pdfinfo	9	1958	1	1333	0	376	0	501	0	1192	0	1228	0	1272	0	1093	0	912	0
pdftohtml	11	2330	0	1805	0	879	0	500	0	1176	0	1357	0	1250	0	1405	0	1203	0
pdftoppm	7	1521	0	1797	1	1916	0	546	0	1216	0	1499	0	1328	0	1360	0	1143	0
pdftops	9	2968	1	2503	0	2456	0	815	0	1358	0	1898	0	1701	0	1928	0	1284	0
pdftotext	10	1749	0	1685	0	1726	0	537	0	1194	0	1519	0	1402	0	1268	0	1023	0
xmllint	25	3676	1	6561	1	5988	1	1959	0	2865	0	2516	0	2901	0	3136	0	2382	0
xmlcatalog	14	4271	0	3821	1	440	0	1553	0	2482	0	1860	0	2003	0	2621	0	1778	0
Total Bugs			12		6		6		3		2		2		2		2		2



(a) VAfuzz **(b) ZigZagFuzz**

Figure 3: Coverage over time plots for djpeg.

when VAFuzz went through the feedback analysis and mutated presence conditions (red lines in Figure 3a). Compared to ZigZagFuzz which uses a pre-defined schedule to mutate configurations for 30 minutes every hour, VAFuzz goes back and forth between analysis with different scheduling, and fuzzes the data seed queue more frequently which may be the reason for the significant improvement in coverage results. We observe a similar trend in jpegtran.

There were 4 target programs (tiffcrop, readelf, pdftoppm, and xmllint) where VAFuzz did not perform as well as some baselines. Among these, the performance drop in xmllint is the most significant, with VAFuzz achieving a coverage of 3676 edges, while ZigZagFuzz and ConfigFuzz covered 6561 and 5988 edges, respectively. We therefore investigated this outlier result, with the support of the visualization similar to the ones presented in Figure 3. For xmllint, we observe that VAFuzz produced lower coverage in the initial stages (first 15 minutes). This may be due to the fact that VAFuzz's initial configurations are not effective in exploring the code paths of xmllint, while ZigZagFuzz always aggressively explores the configuration space at the beginning. We also observe that while VAFuzz frequently performs the feedback analysis, this analysis did not yield significant improvements in the coverage of xmllint, especially in the early stage of the fuzzing campaign.

4.2.2 Bug Detection. The B columns in Table 1 show the numbers of unique bugs found by each fuzzer in our evaluation. VAFuzz found a total of 12 bugs across all the target programs, while ZigZagFuzz, ConfigFuzz, and AFL++-argv found 6, 6, and 3 bugs, respectively. All sampling-based baselines found 2 bugs each, both of which are in gif2png, and are found by all the experimented fuzzers. Among these bugs, 7 were only found by VAFuzz, 2 of which are previously unknown bugs in xmllint and tiff2pdf. Recall that these programs are popular fuzzing targets and therefore well tested; the capability of detecting new bugs in VAFuzz highlights the significance of this result. The bug found in tiff2pdf is a denial of service bug, where the program continues to run indefinitely for a prolonged period of time. We have responsibly reported this to the developers of tiff2pdf, and they have confirmed the bug and pushed a fix. The bug found in xmllint is a segmentation fault, where the program crashes when processing a crafted XML file.

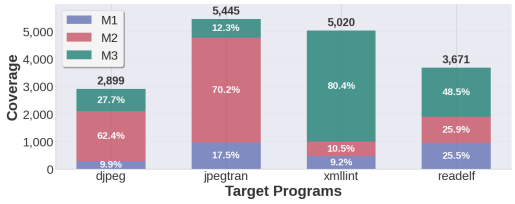


Figure 4: Mutator coverage contribution on four programs.

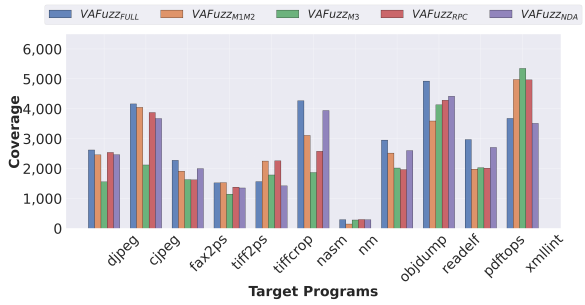


Figure 5: Contribution of VAFuzz components.

The bug has been confirmed and fixed by the developers. Notably, the two unique bugs found by VAFuzz are variability bugs, which are bugs that only occur when certain configuration options are enabled. Details of all the detected bugs, including the information regarding overlapping bugs (bugs found by multiple fuzzers), are available in our artifact [1].

4.2.3 Contribution of Mutators and Presence Conditions. To understand the contribution of VAFuzz's mutators, we analyzed their coverage contribution in djpeg, jpegtran, readelf, and xmllint, shown in Figure 4. We find that mutator effectiveness varies by program. M2 achieved the highest gains for djpeg (62.4%) and jpegtran (70.2%), while M3 performed the best for xmllint (80.4%) and readelf (48.5%). This suggests that VAFuzz mutators are complementary yet program-sensitive. We also examined the impact of presence conditions in these four programs. For djpeg, jpegtran, and xmllint, a small number of presence conditions accounted for most coverage (e.g., 6 conditions yielded 92.3% coverage in djpeg), while readelf required more conditions (18 for 58%). This suggests that leveraging a few important conditions can yield substantial coverage, underscoring the importance of presence conditions in VAFuzz.

4.3 RQ2: Ablation Study

In this section, we perform ablation study to analyze: (1) contribution of different components, (2) impact of initial sampling approach, and (3) overhead of the components on fuzzing performance.

4.3.1 Contribution of Different Components. We evaluated four VAFuzz variants that disable specific components: (1) **VAFuzz**$_{M1M2}$ uses only presence condition-based mutators (M1, M2), omitting M3. (2) **VAFuzz**$_{M3}$ uses only the random grammar-based mutator M3. (3) **VAFuzz**$_{RPC}$ selects presence conditions randomly, ignoring priority values. (4) **VAFuzz**$_{NDA}$ omits data association mapping.

Table 2: Coverage results for different initial configurations.

Program	1E	ME	2W	Rand	DB
djpeg	**2621**	2615	2604	2620	2598
cjpeg	4162	**4165**	4154	4160	4152
jpegtran	5292	**5330**	5281	5299	5299
fax2ps	2275	2263	**2276**	2271	2268
fax2tiff	1586	**1594**	1589	1576	1579
tiff2pdf	2087	2100	2081	2094	**2105**
tiff2ps	1522	1515	**1526**	1521	1510
tiffcp	2648	2632	2644	2640	**2652**
tiffcrop	1564	1556	1561	**1569**	1563
tiffinfo	1240	1235	1231	1239	**1258**
gif2png	**414**	**414**	412	410	411
nasm	4268	4275	4271	**4300**	4261
ndisasm	877	872	**881**	879	872
nm	291	291	289	284	**295**
objdump	**2952**	2945	2950	2947	2944
readelf	4924	4917	**4927**	4921	4925
size	**151**	150	**151**	149	145
pdfimages	1807	1801	**1812**	1789	1795
pdfinfo	1958	1951	1950	**1964**	1948
pdftohtml	2330	2327	2321	2325	**2333**
pdftoppm	1521	1524	1515	**1529**	1510
pdftops	2968	2970	**2972**	2955	2959
pdftotext	1749	1740	1759	**1780**	1739
xmllint	3676	**3752**	3680	3682	3671
xmlcatalog	4271	**4289**	4285	4269	4251

For this ablation, we selected 11 programs with 20+ options (Figure 5). VAFuzz$_{FULL}$ achieved the highest coverage in 7 of 11 programs, supporting the hypothesis that combining variability-aware feedback, mutators, and data mapping improves exploration of configuration spaces.

VAFuzz$_{M1M2}$ outperforms VAFuzz$_{M3}$ in 7 programs, averaging 1.2× more coverage, highlighting the importance of presence condition-based mutators. VAFuzz$_{M3}$ is slightly better on pdftops, readelf, and xmllint, and significantly on nm (1.9×). VAFuzz$_{FULL}$ outperforms both variants on average (1.2× and 1.4×), showing the mutators' complementarity. Notably, VAFuzz$_{FULL}$ underperforms on xmllint and tiffcrop, consistent with RQ1, suggesting program-specific limitations.

Comparing VAFuzz$_{RPC}$ to VAFuzz$_{FULL}$, the latter outperforms in 8 programs, but tiffcrop favors random selection, surpassing even ZigZagFuzz by 26%. This indicates that priority-based selection is generally effective but can be program-dependent.

Finally, VAFuzz$_{FULL}$ consistently outperforms VAFuzz$_{NDA}$, confirming the benefit of data association mapping.

4.3.2 Impact of Initial Sampling Approach. In order to understand the impact of initial configuration generation (Section 3.1), we compared the default sampling approach, one-enabled (1E), against the following sampling-based initial configuration generation approaches: most-enabled (ME), two-way covering array (2W), random sampling (Rand), and diversity-based sampling (DB) using Hamming distance [35]. We used the same sets of configurations from the corresponding baselines introduced in Section 4.1, but using them as the initial configurations of VAFuzz in this ablation study. The coverage results of this experiment is shown in Table

2. Notably, the impact of the initial sampling approach on the performance of VAFuzz is not significant, the average coverage for all variants are very similar: between 2361 to 2368; the maximum variance was 42.5 with standard deviations ranging from 5.8 to 7.1. This result provides further evidence that the variability-aware analysis of VAFuzz is capable of discovering significant presence conditions, despite using different initial configurations.

4.3.3 Overhead of VAFuzz Components. We calculated the overhead introduced by the components of VAFuzz. Specifically, we calculated the time taken by the *Variability-Aware Presence Condition Analysis*, *Variability-Aware Data Seed Analysis*, and *Presence Condition Selection and Mutation* components. Overall, VAFuzz adds minimal overhead, averaging 31.7 minutes in 24 hours of fuzzing, with the Variability-Aware Presence Condition Analysis incurring the most overhead (on average 30.5 minutes). This suggests that the components of VAFuzz do not incur significant overhead. Detailed per-target results are available in our artifact [1].

4.4 Threats to Validity

Our evaluation faces several threats to validity. First, fuzzing is inherently random. To mitigate this, we ran each fuzzer for 24 hours over 5 trials, following prior work [21, 38], and used the same set of target programs to ensure a fair comparison. While bugs in our VAFuzz implementation could influence results, we mitigated this through extensive testing. Second, our findings may not generalize beyond the selected targets. We evaluated 25 programs commonly used in the fuzzing community; however, no standard benchmark for fuzzing program configurations exists. Developing such a benchmark is left as future work. Finally, the accuracy of the configuration grammar could affect our results.

5 Related Work

In this section, we discuss related work that presents variability-aware analysis and additional work that tests configurable software.

Variability-aware analysis and execution. Variability-aware analyses have been developed to detect different types of bugs in configurable software. These analyses often construct specialized variability-aware data structures (e.g., variability-aware AST [31, 37]) and develop analyses based on these data structures to associate presence conditions with the detected bugs.

Varex [31] is a PHP interpreter that utilizes variability-aware execution to test plugin-based systems, which are highly configurable. Varex was the first to introduce the concept of variability-aware execution in a real-world setting and is one of our core inspirations for this work. Kim *et al.* introduced shared execution [20], which extended the interpreter of Java Pathfinder [47] into a variability-aware interpreter. This technique was applied to test software product lines and have shown to be very effective in speeding up the testing process. Our work is the first to apply lightweight variability-aware analysis during fuzzing to associate the presence conditions with the feedback of the fuzzer.

Testing program configurations in SPL. The rich research in SPL addresses the problem of exploring the configuration space through many different techniques [9, 10, 33]. For example, T-wise testing [4] attempts to cover all possible combinations of t parameters by guaranteeing that each combination of t parameters is covered at least once. These techniques often test the target program in a black-box fashion, while we apply grey-box fuzzing to explore the configuration space. There has also been ample work on white-box approaches to explore the configuration space of SPLs through static analysis, symbolic execution, or model-based approaches. Kim et al. [18, 19] propose static analyses to reduce the number of configurations needed for testing or monitoring. Their methods analyze feature relevance or identify configurations that provably cannot violate a specified safety property. These approaches assume a complete feature model and utilize static program analysis to identify relevant features or instrumentation points. Shi et al. [40] introduce a compositional symbolic execution technique for analyzing feature interactions, using code-based dependency analysis and symbolic execution. These white-box approaches may incur scalability limitations through pre-analysis or symbolic execution if applied to fuzzing, and have focused on applications beyond dynamic bug finding such as test completeness and integration testing. In contrast, VAFuzz's dynamic process enables it to effectively find runtime bugs in the configuration space. Tartler et al. [43, 44] quantify and maximize configuration coverage in compile-time configurable software like the Linux kernel. They propose methods to automatically derive configurations for static checkers based on CPP directives and KCONFIG models. While effective for code coverage in static analysis, their focus is not on dynamic runtime bug finding, and complements our focus on run-time configurations.

In addition to the fuzzers discussed in Section 2, several others target program configuration spaces. Eclipser [7] leverages symbolic execution to track both data bits and their programmatic constraints, and can mutate configurations. TOFU [27] is a directed fuzzer that applies structural mutations to guide execution toward target basic blocks, requiring a specification (similar to our configuration grammar) to describe the configuration space. ECFuzz [26] explores configurations of large-scale systems using a multi-dimensional generation strategy and custom mutators, validating configurations via unit tests to reduce resource consumption. Unlike VAFuzz, these approaches do not exploit variability-aware execution and analysis to guide fuzzing.

6 Conclusions

In this paper, we propose *VAFuzz*, a novel variability-aware fuzzer that integrates dynamic variability-aware analysis within the fuzzing process to enhance configuration space exploration. We identify and address key limitations of existing configuration fuzzing approaches; *VAFuzz* utilizes configuration-data co-execution and variability-aware feedback analysis of presence conditions to effectively guide the fuzzing process in the configuration space. Our evaluation of *VAFuzz* on a diverse set of configurable fuzzing targets show that *VAFuzz* outperforms the state-of-the-art configuration-aware fuzzers in terms of code coverage and bug detection. In addition, our ablation study demonstrates the effectiveness of each component of VAFuzz.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-2008905, CCF-2047682, and CNS-2346528.

References

[1] 2025. Artifact for Variability Aware Fuzzing. https://github.com/UTD-FAST-Lab/VAFuzz

[2] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 3, Article 10 (Jan. 2018), 34 pages. https://doi.org/10.1145/3149119

[3] Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury. 2023. Efficient Greybox Fuzzing to Detect Memory Errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/3551349.3561161

[4] Eduard Baranov, Sourav Chakraborty, Axel Legay, Kuldeep S. Meel, and Vinodchandran N. Variyam. 2022. A scalable t-wise coverage estimator. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/3510003.3510218

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[6] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. 2010. Improving the testing and testability of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond* (Jeju Island, South Korea) *(SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 241–255.

[7] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747.

[8] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[9] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. 1997. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444. https://doi.org/10.1109/32.605761

[10] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, and C.J. Colbourn. 2003. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings.* 38–48. https://doi.org/10.1109/ICSE.2003.1201186

[11] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/1273463.1273482

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[13] Docker, Inc. 2024. *Docker: Open Source Container Platform.* https://www.docker.com Version 24.0.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[15] GNU Project. 2024. *GDB: The GNU Project Debugger.* https://www.gnu.org/software/gdb/ Version 14.2.

[16] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 805–824. https://doi.org/10.1145/2048066.2048128

[17] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 773–792. https://doi.org/10.1145/2384616.2384673

[18] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development* (Porto de Galinhas, Brazil) *(AOSD '11)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/1960275.1960284

[19] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. 2010. Reducing Configurations to Monitor in a Software Product Line. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 285–299.

[20] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE '12)*. IEEE Computer Society, USA, 221–230. https://doi.org/10.1109/ISSRE.2012.23

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[22] Konstantin Kuznetsov, Kostya Serebryany, Dmitry Vyukov, Tatiana Cheindel, Alexander Potapenko, and Alexander Stepanov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*. https://clang.llvm.org/docs/AddressSanitizer.html

[23] Kim Lauenroth, Klaus Pohl, and Simon Toehning. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, USA, 269–280. https://doi.org/10.1109/ASE.2009.16

[24] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program Option-Aware Fuzzer for High Bug Detection Ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 220–231. https://doi.org/10.1109/ICST53961.2022.00032

[25] Ahcheong Lee, Youngseok Choi, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim. 2025. ZigZagFuzz: Interleaved Fuzzing of Program Options and Files. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 39 (Jan. 2025), 31 pages. https://doi.org/10.1145/3697014

[26] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 48, 12 pages. https://doi.org/10.1145/3597503.3623315

[27] Sheng Li, Abhinav Vaswani, Cedric Liang, Ziyan Zhao, Emma Simister, and Michael Chan. 2020. Efficient Near-Exact Shapley Value Computation Using Contributions Marginality.

[28] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476

[29] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[30] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On essential configuration complexity: measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322

[31] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300

[32] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N. Nguyen. 2015. Detecting semantic merge conflicts with variability-aware execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 926–929. https://doi.org/10.1145/2786805.2803208

[33] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. https://doi.org/10.1145/1883612.1883618

[34] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, null (Nov. 2011), 2825–2830.

[35] D.K. Pradhan, D. Kagaris, and R. Gambhir. 2005. A Hamming distance based test pattern generator with improved fault coverage. In *11th IEEE International On-Line Testing Symposium.* 221–226. https://doi.org/10.1109/IOLTS.2005.6

[36] Qualys Security Researchers. 2021. CVE-2021-3156: Heap-based Buffer Overflow in Sudo (Baron Samedit). National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2021-3156 Accessed: 2025-03-14.

[37] Alexander Von Rhein, JöRG Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 18 (Nov. 2018), 33 pages. https://doi.org/10.1145/3280986

[38] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1974–1993. https://doi.org/10.1109/SP54263.2024.00137

[39] Kostya Serebryany. 2015. LibFuzzer - A Library for Coverage-Guided Fuzz Testing. LLVM Project. https://llvm.org/docs/LibFuzzer.html

[40] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–284.

[41] Charles Song, Adam Porter, and Jeffrey S. Foster. 2012. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *2012 34th International Conference on Software Engineering (ICSE)*. 903–913. https://doi.org/10.1109/ICSE.2012.6227129

[42] Robert Swiecki. 2021. Honggfuzz: A Security Oriented Fuzzing Tool. GitHub Repository. https://github.com/google/honggfuzz

[43] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, Berkeley, CA, USA, 421–432.

[44] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review* 45, 3 (Jan. 2012), 10–14. https://doi.org/10.1145/2039239.203924

[45] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[46] Daniel Veillard and contributors. 1999–present. libxml2 - The XML C parser and toolkit. http://xmlsoft.org/

[47] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. 2000. Model checking programs. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. 3–11. https://doi.org/10.1109/ASE.2000.873645

[48] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. 2023. CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1919–1936. https://www.usenix.org/conference/usenixsecurity23/presentation/wang-dawei

[49] Dawei Wang, Geng Zhou, Li Chen, Dan Li, and Yukai Miao. 2024. ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 735–749. https://doi.org/10.1145/3658644.3690231

[50] Kelin Wang, Mengda Chen, Liang He, Purui Su, Yan Cai, Jiongyi Chen, Bin Zhang, Chao Feng, and Chaojing Tang. 2024. OSmart: Whitebox Program Option Fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 705–719. https://doi.org/10.1145/3658644.3690228

[51] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1634–1645. https://doi.org/10.1145/3510003.3510174

[52] Yi Xiang, Han Huang, Yuren Zhou, Sizhe Li, Chuan Luo, Qingwei Lin, Miqing Li, and Xiaowei Yang. 2022. Search-based diverse sampling from real-world software product lines. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1945–1957. https://doi.org/10.1145/3510003.3510053

[53] Michal Zalewski. 2017. American Fuzzy Lop (AFL). Online. https://lcamtuf.coredump.cx/afl/

[54] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 53 (March 2023), 21 pages. https://doi.org/10.1145/3580597