ALGERNON: A Flag-Guided Hybrid Fuzzer for Unlocking Hidden Program Paths

Peng Deng, Lei Zhang, Jingqi Long, Wenzheng Hong, Zhemin Yang, Yuan Zhang, Donglai Zhu, and Min Yang
Fudan University
Shanghai, China
pdeng21@m.fudan.edu.cn, {zxl, jqlong20, wzhong20, yangzhemin, yuanxzhang, zhudl, m_yang}@fudan.edu.cn

***Abstract*— Fuzz testing is a widely used method for finding security issues in software. However, certain code paths can only be explored under specific program states. Flag variables, which represent internal states, are crucial in influencing program behavior through flag-guarded branches. Unfortunately, existing fuzzing tools struggle to efficiently explore them due to the implicit data dependency between flag variables and the input. As a result, they commonly lack awareness of the dependency between program input and the assignments of critical flag variables, leading to a blind or random approach to satisfy flag-checking constraints, which greatly impacts the fuzzing efficiency.**

To address this issue, this paper proposes a dynamic flag-guided hybrid fuzzing approach, which automates the identification of flag variables and provides guidance for fuzz testing. Specifically, we first design a pre-fuzzing program analysis to recognize flag variables and a novel data structure to present how flag variables guard code branches. Then, we propose a new constraint-solving approach by separating complex flag-checking constraints into a set of atomic ones and sequentially solving them by traversing our FDG to locate execution paths that could assign the flag variables with the desired values.

We implement a prototype tool, called ALGERNON, and evaluate it on 20 popular open-source programs. Across all tested programs, ALGERNON outperforms QSYM, Angora, AFL++, and INVSCOV in terms of both code coverage and vulnerability discovery, demonstrating the effectiveness of our approach. During our experiments, ALGERNON successfully found 30 zero-day vulnerabilities with 11 CVE IDs assigned.

I. INTRODUCTION

Fuzz testing (or fuzzing) is a widely adopted technique for automatically generating inputs and exploring code logic, making it essential for uncovering vulnerabilities and enhancing software security. However, certain code paths—and the vulnerabilities they contain—can only be exercised when the program is in specific internal states [1]. While some state-of-the-art fuzzers [2], [3], [4], [5] have been developed to target program states annotated by analysts or hinted at in documentation, these approaches cover only a small portion of the possible state space. In reality, many programs rely on complex internal states that directly influence their behavior. This paper focuses on a common and influential type of internal state, known as *flag variables*.

A flag variable is a binary or multi-state variable representing specific conditions or system states within a program. These variables control execution flow through the *flag-guarded branches*, which are prevalent in real-world software. Critically, some of the most difficult-to-reach vulnerabilities lie within these flag-guarded paths. To assess the impact of flag variables on software security, we analyzed CVEs from the past two years across 10 widely used programs in the UniFuzz benchmark [6]. Of the 102 CVEs examined, 64 were directly associated with flag variables. This finding highlights a significant security gap: conventional fuzzing techniques often fail to effectively navigate the complex input-to-flag dependencies, leaving critical vulnerabilities undiscovered. Addressing this challenge requires targeted techniques that explicitly identify and guide exploration through flag-guarded branches.

```
1 if (check_input()) { assign_mode(); }
2 if (mode == COMPRESSION_JPEG) { assign_type(); }
3 if (type == PHOTOMETRIC_RGB) { do_sth(); }
4 if (mode == COMPRESSION_JPEG && type == PHOTOMETRIC_RGB) { do_sth(); }
```

Fig. 1: A simplified example of flag variables.

Consider a simplified example in Figure 1. The variables `mode` and `type` are flag variables employed to track the compression and photometric modes, respectively. At lines 2, 3, and 4, branches are guarded by these flags to control program flow. Specifically, line 1 checks the input data and assigns a value to `mode`, while line 2 sets `type` based on the value of `mode`. Line 4 shows a more complex case, where execution depends on both `type` and `mode`. In such scenarios, specific combinations of flag values are necessary to trigger or disable certain program features.

It's important to note that both `mode` and `type` have no explicit data dependency on the input [7]. To explore the branch at line 4, the program must be in a specific state where `mode` and `type` hold the required values. Unfortunately, capturing this implicit dataflow is challenging and often leads to over-tainting or under-tainting issues [8]. Existing state-driven fuzzing approaches [9], [1], [10], [11], [5], [4], [3], [12], [13] struggle to explore flag-guarded branches by simply mutating seeds to satisfy flag-checking constraints. Overcoming this limitation requires a comprehensive strategy—one that can accurately identify flag variables, resolve their associated constraints, and guide the fuzzer through complex flag-dependent logic. Achieving this presents several key challenges.

Firstly, flag variables have implicit dependencies on input, making it difficult for existing input tracking methods to locate them efficiently. Moreover, since there is typically no documentation or specification for flag variables, developers may use arbitrary names and types, further complicating their

identification among all program variables. To tackle this challenge, we initiate our approach with a pre-fuzz program analysis that leverages semantic and structural characteristics to identify flag variables. While these variables generally lack unique syntactic patterns, they often exhibit recognizable traits in their value sets, most notably, holding predefined compile-time constants. Unlike conventional implicit flow analysis that rely heavily on taint analysis, our method employs static program analysis. This makes our method both efficient and scalable in recognizing flag-guarded branches.

Secondly, to satisfy the conditions of flag-guarded branches, it is necessary to know the values of the corresponding flag variables and identify execution paths that can pass through their assignment points, as only these paths can fulfill the required conditions. However, most existing techniques are unable to even detect flag-guarded branches, let alone perform constraint-solving or targeted mutations. Moreover, flag variables can have numerous combinations, but only specific ones can lead to new code paths. The rest often yield no benefit, resulting in inefficient exploration. Due to the vast number of possible combinations, existing fuzzers struggle to meet the requirements for precise value assignments. To address this, we construct a new data structure called the FDG, which captures the dependencies between flag-checking statements and their corresponding flag assignments. By leveraging the FDG, we enhance the constraint-solving capabilities of the concolic engine to more effectively guide execution through flag-guarded logic.

Finally, in real-world programs, flag-checking statements can be nested, and zero-day vulnerabilities usually reside in such "deep" code logic. As shown in our motivating example, a flag-checking statement can be control-dependent on other flag-checking statements, meaning that satisfying it requires fulfilling earlier flag conditions first. The key challenge lies in recognizing these dependencies and determining the correct sequence of flag assignments. To address this, we use FDG to record the control dependencies, allowing us to guide the fuzzer to generate inputs that trigger flag assignments in the correct order, ultimately satisfying the nested conditions. Our approach enables the fuzzer and concolic engine to first reach the necessary flag assignments, ensuring they can satisfy subsequent flag-checking conditions. To further improve efficiency and precision, we introduce a novel feedback metric called *flag edge coverage* and an innovative seed scheduling strategy to prioritize promising inputs.

We implement our prototype of ALGERNON and evaluate its effectiveness on a benchmark of 20 popular real-world programs. ALGERNON successfully discovered 30 unique zero-day vulnerabilities, all of which were responsibly disclosed to the respective developers. To date, 11 of these vulnerabilities have been assigned CVE IDs. We compared ALGERNON with the state-of-the-art hybrid fuzzers, including QSYM [14], Angora [15], and general fuzzers like AFL++ [16] and INVSCOV [17]. The results show that ALGERNON outperforms these tools in terms of both code coverage and security vulnerability detection. For example, ALGERNON discovers 1.41 times more unique branches than Angora, and 2.14 times more unique crashes than QSYM.

The main contributions of our work are as follows.

- We systematically introduce the concept of flag variables and flag-guarded branches, highlighting their significance in software security and fuzzing.
- We propose a lightweight and efficient algorithm to recognize flag variables, and construct the FDG to guide the fuzzer to reach flag-guarded branches. We implement a dynamic flag guided hybrid fuzzer, called ALGERNON.
- We comprehensively evaluate ALGERNON against state-of-the-art (hybrid) fuzzers, demonstrating its efficiency in program exploration and vulnerability detection.

II. PROBLEM STATEMENT

A. Definition

We define a *flag variable* as a variable that represents a binary or multi-valued internal state, often set based on earlier input-dependent logic and later used to control execution flow. A *flag-guarded branch* is a conditional branch whose execution is determined by the value of a flag variable. These branches often gate access to critical functionality or error-prone code regions and are common in real-world programs.

B. Motivating Example

In this section, we present a simplified example from a popular MP4 parser to illustrate the challenges posed by flag-guarded branches, shown in Figure 2. The code contains two flag variables, `type` and `mode`. Lines 11, 12, and 13 assign different values to `type` based on `input->tag`, allowing for 128 possible values of `type`. The value of `type` is then checked to determine the corresponding behavior at various points, such as lines 14 and 17 (referred to as *flag-guarded branches* in this paper). At line 15, we skip over 20 possible values for `mode`, with one specific value checked at line 20. This setup creates 2,560 possible combinations of `type` and `mode`, yet only one specific combination can reach the vulnerable function. The vulnerable function, located at line 21, is protected by nested flag-guarded branches at lines 14, 19, and 20. To reach this function, `type` must be set to `TYPE_SENC` and `mode` to `MODE_A`. However, existing approaches struggle with this due to their lack of awareness of flag variables and their dependencies. They cannot *solve* for inputs at flag checks because the required flag values must be set far earlier in the control flow. For example, to explore the branch at line 19, a fuzzer must ensure that line 12 is executed with `type` set to `TYPE_SENC`. Standard techniques lack this proactive path guidance.

Applying Existing Approaches. To meet the condition at line 12, a fuzzer using dynamic taint analysis could locate input bytes affecting `input->tag` and mutate or solve for them. However, this approach fails for flag-checks, as the control flow may have already set `type` to an undesired value before reaching the check. Achieving the correct flag-check requires proactive path control much earlier in execution.

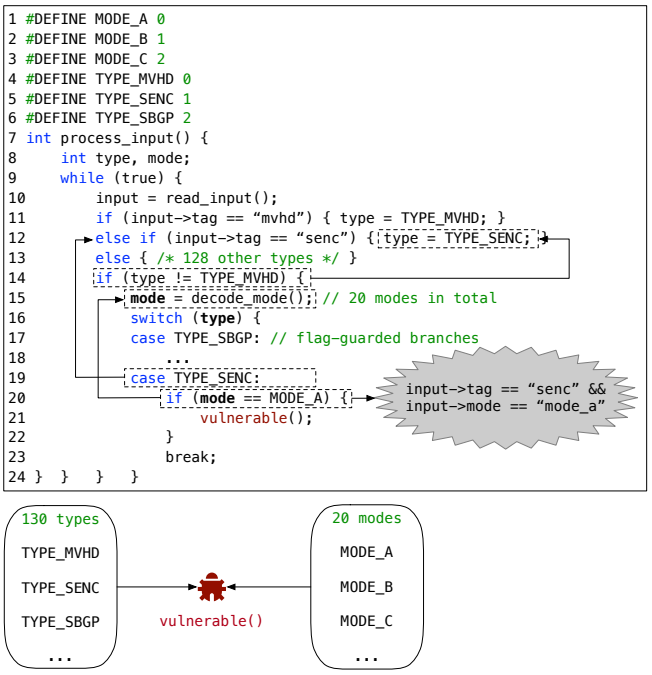


Fig. 2: A motivating example of flag-guarded branches.

Out of the 2,560 possible combinations of `type` and `mode`, only a single combination can lead to line 21, making it extremely difficult for existing techniques to efficiently explore such code locations. Without considering such dependencies, fuzzers may waste resources on useless mutations in an attempt to cover unreachable flag-guarded branches. Consider the scenario where the combination of `TYPE_SBGP` and `FLAG_A` is deemed invalid. In this context, the absence of any corresponding code segments associated with this specific combination implies the lack of relevant program logic. Nevertheless, the presence of such unexplored combinations introduces inefficiency in the fuzzing process. The reason behind this inefficiency lies in the possibility that the assignment statement might be triggered by these specific values, leading to the generation of new code coverage. Consequently, the fuzzer expends valuable computational resources on exploring these unproductive cases, resulting in a waste of energy.

To validate the effectiveness of existing approaches, we further constructed 120 valid combinations in the example and applied several open-source fuzzing techniques, including AFL [18], AFL++ [16], Angora [15], QSYM [14], and INVSCOV [17], against the motivating example. After 24 hours, none of these tools were able to cover all 120 combinations or reach the vulnerable function, highlighting their limitations in exploring flag-related code. In contrast, ALGERNON covered all 120 combinations and triggered the vulnerability in under an hour. This result demonstrates the efficiency and effectiveness of our approach in navigating flag-guarded branches and reaching deep logic protected by complex internal state combinations.

C. Our Solution

STEP#I: Recognizing flag variables based on value sets and operations. Unlike variables tainted by input data, flag variables' values are usually from a restricted collection of compile-time constants. This characteristic ensures that flag variables maintain consistent values throughout program execution, contributing to code reliability and predictability. This finite and predetermined range of potential values provides a distinct semantic and structural characteristic that sets flag variables apart from other variables in a program.

On the other hand, as analyzed above, the flag variables should not be "tainted" by input, which means that compared with other variables that can hold the input data, the data transformation on the flag variables is simpler. For example, in our motivating example in Figure 2, the `type` variable is left untouched during execution until it is compared with a constant. This highlights the distinctive behavior of flag variables compared to other variables in terms of data manipulation. We will discuss the technical details in Section IV-A1.

STEP#II: Resolving flag-guarded constraints by navigating to required flag variable assignments. For example, the flag-guarded branch at line 19 of the motivating example compares the value of the flag variable with the flag constant `TYPE_SENC` (which we call *flag-checking statements*). In order to satisfy the branch condition, there must be an assignment statement somewhere in the program, which we call *flag assignment* statement, and the right value of the assignment is `TYPE_SENC`. In the motivating example, the flag assignment statement is located at line 12. It is worth noting that the mentioned corresponding relationship depends entirely on whether the flag check and the flag assignment statement use the same flag constant, which does not involve complex program data flow analysis. This method is reasonable because of the unique propagation method of flag variables mentioned above, that is, the program directly propagates the value of the flag variables without any modification.

STEP#III: Proactively guiding fuzzer to explore flag-guarded branches using FDG. As illustrated before, satisfying a flag-checking statement is very different from solving a conditional statement which directly examines the value of input bytes. The difficulty comes from the requirement for reverting some branches that have already been executed. For example, if the execution takes the "wrong" path between line 11 and line 13 in Figure 2, e.g., letting `type` equal to `TYPE_SBGP`, it will have no way to get to the vulnerability anymore when reaching the line 21 statement (because the `type` already holds the "wrong" concrete value and the branch to be taken is determined). Such a problem is also known as over-constraint [14] in the symbolic execution research area.

Trying to revert an executed path when we reach the flag-checking statement is hard because it can cause unexpected changes in the control flow (the changed execution path may not reach the flag-checking statement we try to satisfy in the first place). Even for a simple function like ours which only contains two-level nested flag-checking statements of

`type` and `mode`, it still requires careful manipulation of the execution path to reach two way points (line 12 and corresponding branch at line 15 `decode_mode()` to set the value of `mode`) to finally reach the vulnerability. Instead of expecting the fuzzer to "happen to" reach the waypoints to set the value of flag variables, or reverting the path only when we reach some flag-checking statements, we need to "foresee" the required flag assignments to solve the flag-checking statements and guide the execution to proceed as expected. This is why proactive guidance of the control flow is the key to exploring the flag-guarded branches.

III. Overview

In this section, we present the overall design in Figure 3 to efficiently explore the flag-guarded branches. To achieve this target we need to solve two challenges: identifying flag variables and, based on that, satisfying flag-checking statements.

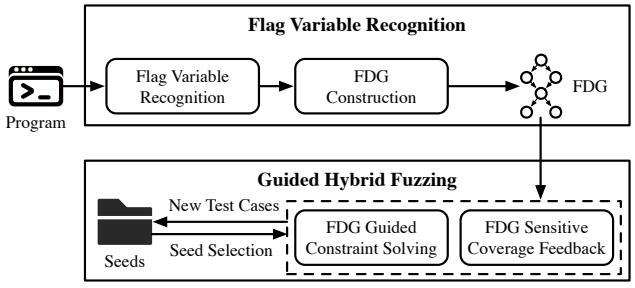


Fig. 3: Overall architecture of ALGERNON.

As analyzed in Section II-C, identifying as many flag variables as possible before fuzzing is essential for our approach. However, existing input-tracking methods, regardless of accuracy, can only passively detect a limited subset of flag variables that are already accessible via fuzzer-generated inputs. To address this, we introduce a pre-fuzzing static analysis to locate candidate flag variables. Starting with all program variables, we leverage insights into the unique characteristics of flag variables to filter candidates, then instrument the code to monitor their values during fuzzing.

Using the identified flag variables, we construct a specialized representation called the *Flag Dependency Graph (FDG)* to model the dependency relationship between flag-checking statements and their corresponding flag assignments. With this structure, we can plan execution paths to efficiently explore flag-guarded branches.

We further propose a *Dynamic Flag-guided Hybrid Fuzzing* technique to guide input generation toward target branches. Instead of traditional distance-based guidance, we incorporate a concolic execution engine for effective guidance. The engine is initialized with the FDG and continuously matches each input's execution path with the FDG to identify the closest unreached flag-checking statements. It then flips specific branches to generate inputs that are more likely to reach the target. Additionally, we design a new feedback metric and seed scheduling strategy to enhance hybrid fuzzing efficiency.

IV. Design

In this section, we dive into the design of ALGERNON.

A. Flag Variable Recognition

We begin our flag-guided fuzzing process by conducting a pre-fuzzing static analysis to gather essential information. Specifically, in this process, we try to collect two classes of information: 1) all the flag variable candidates in the program, and 2) the mapping between flag-checking statements and the flag assignments (which give the checked flag variables the desired values to satisfy the conditions).

In this section, we explain our analysis methodology in detail, outlining the steps taken to accurately identify flag variables and establish the necessary mappings between flag-checking statements and corresponding flag assignments.

1) Flag Variable Recognition with Candidate Filtering: As analyzed above, guiding the exploration of flag-guarded branches requires comprehensive knowledge of all flag variables. Instead of passively identifying flag variables by modifying inputs during fuzzing, we proactively acquire this information beforehand. To achieve this, we start by considering all program variables as potential flag variables. Then, using a filtering mechanism based on our insights, we exclude variables that are clearly not flags, resulting in a refined set of flag variable candidates. We present the overview of this module in Figure 4.

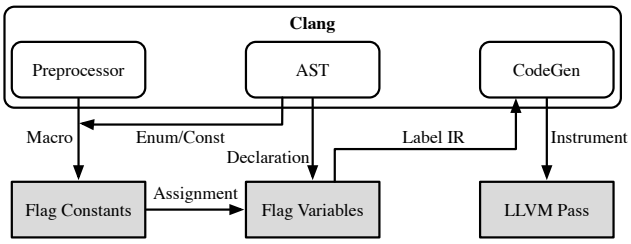


Fig. 4: The overview of recognition of flag variables.

Filtering Flag Variables. In this paper, we define a flag variable as one that is implicitly dependent on input and only hold a set of compile-time constants (referred to as *flag constants*). This means that along any execution path in the program, flag variables should not be tainted by input data.

Generally, the values of program variables may depend on inputs or system environment variables, making their range dynamic and unpredictable at compile time. In contrast, flag variables have distinct characteristics. Their assignments are *predetermined* and remain *constant* throughout program development. This fixed range of values makes flag variables easier to analyze and track reliably, as they are not influenced by dynamic changes. Based on this, we exclude certain program variables using two approaches.

First, we exclude variables that can be assigned input data. We apply a static dataflow analysis that starts with predefined input APIs, such as file operations and stream processing APIs, to trace the flow of input data and its assignment to variables. Variables directly influenced by input data are marked as

tainted and excluded from the candidate set of flag variables. Notably, to keep this process conservative, we do not track data propagation through pointers. Although this approach may lead to some false negatives during variable filtering, it is an acceptable trade-off for later analysis steps.

Second, we exclude variables used as operands in arithmetic operation. Typically, flag variables are not involved in arithmetic operations but are primarily used in bitwise (e.g., AND, OR, XOR) and logical operations (e.g., equality) within branch conditions, as depicted in Figure 5. Developers commonly use flag variables for readability or to enforce coding standards, and using them in arbitrary type conversions or arithmetic operations can undermine their intended purpose and complicate future development and maintenance.

```
1 if (flag1 & FLAG) {} // Bit-wise operation
2 if (flag2 == PREDEFINED_CONSTANT) {} // Logical operation
```

Fig. 5: Two typical ways how flag variables affect control flow.

Filtering Flag Constants. After excluding variables that do not match flag characteristics, we obtain a set of flag variable candidates. We then refine this set by analyzing both the code *structure* and code *semantics* associated with flag constants. This two-part filtering process is designed to generalize across different codebases while capturing common patterns in real-world software.

(1) Structural Filtering. Flag constants associated with a specific flag variable typically share the same data type and are defined in a uniform code structure for clarity and maintainability. In practice, these constants are commonly declared using enumerations, macro definitions, or constant variables grouped together in header or configuration files. For example, in Figure 2, lines 1–3 define constants for the *mode* variable using macros, while lines 4–6 define constants for *type*. To generalize this filtering across programs, our method scans for: (1) Enumerated types (*enum*) grouped under a single definition; (2) Macro definitions (*#define*) grouped by proximity and datatype; (3) *const* variables declared consecutively with the same type and naming structure. By identifying such clusters of related constants, we can reliably associate them with their corresponding flag variables.

(2) Semantic Filtering. Flag constants often follow naming conventions that signal their grouping and purpose. These include shared prefixes or suffixes (e.g., `MODE_`, `TYPE_`), which reflect internal coding standards such as Hungarian Notation, CamelCase, or PascalCase. For instance, in Figure 2, the constants assigned to *mode* all begin with `MODE_`, making them semantically identifiable as a group. To leverage this, we apply prefix/suffix pattern matching to flag constants grouped during structural filtering. A group of constants is retained only if a significant majority of them share a consistent prefix/suffix, indicating semantic cohesion.

Together, these structural and semantic heuristics form a generalizable and lightweight approach to accurately identifying flag constants, even across diverse codebases. This enables the downstream construction of the FDG and the precise guidance of fuzzing efforts.

2) FDG Construction: Based on the analysis of the program source code, we extract the flag variables and the values of the corresponding flag constants. In this section, we introduce the construction of the FDG.

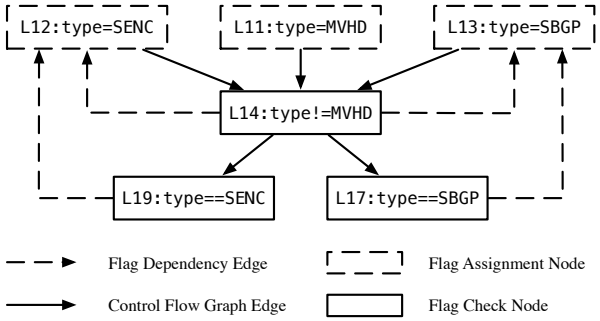


Fig. 6: The simplified FDG for our motivating example.

Based on the recognized flag variables and their associated constants, we identify two key types of program statements: *flag-assignment* and *flag-checking* statements, and we analyze the dependencies between them. The core idea is that to reach and satisfy a flag-checking condition, the program must first execute a corresponding flag-assignment that sets the flag variable to the required value. For example, in our motivating example, the branch at line 19 checks whether the flag variable *type* equals the constant `TYPE_SENC`—this is a flag-checking statement. To satisfy this condition, a flag-assignment statement must assign `TYPE_SENC` to type earlier in the execution. In this case, that occurs at line 12. Importantly, we observe that the dependency between a flag-assignment and a flag-checking statement can be established by comparing their use of the same flag constant. This approach avoids complex data flow analysis, as flag variables are typically assigned and propagated directly without transformation or further computation.

To formally capture these dependencies, we introduce a new data structure called the Flag Dependency Graph (FDG) to model dependencies between *flag-assignment* and *flag-checking* statements. We define FDG as follows:

1) Nodes in the FDG represent conditional branches that involve a flag variable, either through assignment or checking.
2) A flag-assignment node is a statement where a flag variable is explicitly assigned a constant.
3) A flag-checking node is a conditional statement where a flag variable is compared to a constant.
4) An edge is added from a flag-assignment node A to a flag-checking node C if: (1) Both A and C involve the same flag variable; (2) The constant assigned in A matches (or can satisfy) the condition in C; (3) A dominates or can precede C in at least one execution path within the CFG.

Therefore, the edges in FDG represent a satisfiability dependency, indicating that executing A is a prerequisite for satisfying the condition at C.

Figure 6 presents a simplified FDG for the motivating example, with prefixes omitted for clarity. Line 14 checks if `type` is not equal to `TYPE_MVHD`, which can be satisfied by taking either line 12 or line 13. Furthermore, line 19 checks if `type` equals `TYPE_SENC`, which requires line 12 to be executed first. If line 12 is bypassed, the condition at line 19 cannot be met. This illustrates a stricter dependency, where only a specific flag assignment enables a particular flag check.

B. Dynamic Flag-guided Hybrid Fuzzing

1) Fuzzing with Flag Edge Coverage Feedback: To guide the fuzzer more effectively toward flag-guarded branches, we introduce a novel feedback metric called *flag edge coverage*. This metric is designed to focus exploration on paths that involve flag variables.

Flag edge coverage. Unlike standard edge coverage, we label any basic block containing a *flag-assignment* or *flag-checking* statement as a flag-related block, and any edge linked to a flag-related block as a flag edge. In our approach, inputs covering new flag edges or increasing the count of covered flag edges are prioritized. This feedback metric enables ALGERNON to guide the fuzzer toward prioritizing basic blocks related to flag-guarded branches, enhancing exploration efficiency. It is important to note that flag edge coverage supplements standard edge coverage, prioritizing flag-guarded branch exploration. ALGERNON also tracks standard edge coverage. In programs with few flag variables, ALGERNON gracefully degrades, primarily leveraging standard edge coverage, while maintaining effectiveness similar to that of other state-of-the-art fuzzers.

Data-Sensitive Feedback. In addition to the flag edge coverage, ALGERNON also considers data flow information as feedback to drive the fuzzing process. This is because even if the program reaches the same flag-guarded branch, changes in the flag variable value indicate new exploration opportunities. Therefore, ALGERNON includes changes in the flag variable value as a data-sensitive feedback metric. By instrumenting the program to monitor the value of the flag variable, ALGERNON considers it a new exploration when there is any value change.

Mitigating Seed Explosion. It should be noted that there are typically a large number of flag variables in the program (as shown in Section V), and each may have a large set of possible values. As a result, tracking changes in flag values can lead to an excessive number of preserved seeds, which can impact the efficiency of later mutations. To address this, ALGERNON samples and preserves a subset of inputs that trigger new flag values. Specifically, inputs triggering unobserved flag variable values are always preserved. For previously observed values, inputs are randomly sampled and preserved. This trade-off balances thorough exploration with runtime efficiency, ensuring that fuzzing remains scalable even in complex programs with many flag variables.

2) Dynamic Hybrid Fuzzing Guided By FDG: Using FDG, ALGERNON can accurately determine the execution paths needed to satisfy specific flag-guarded branches. For each input in the queue, ALGERNON collects the flag-assignment and flag-checking nodes covered during execution, assessing how well each input aligns with targeted branch conditions.

In the concolic execution module, ALGERNON uses a topological sort of the FDG to identify the shallowest unsatisfied flag-checking statement as the primary target, enabling a systematic and efficient exploration process. As satisfying a flag-checking condition requires first reaching the related flag-assignment statement, ALGERNON applies a seed scheduling strategy (detailed in Section IV-B3) to prioritize inputs that have already reached the relevant flag-assignment statement. If no such input exists, ALGERNON sets the corresponding flag-assignment statement as the new target, selecting inputs that have executed nodes upon which the flag-assignment statement control depends, as determined by the FDG.

Once an input is selected, ALGERNON activates the concolic execution engine to trace its execution path. Unlike traditional concolic methods, ALGERNON focuses exclusively on solving constraints for branches leading directly to the target guided by the FDG, thus minimizing unnecessary constraint solving and improving efficiency. The concolic engine then flips branches on the current execution path to generate inputs more likely to reach the target flag-checking condition, which are then returned to the scheduler.

After exploring a selected flag-guarded branch, ALGERNON dynamically shifts focus to the next shallowest unexplored node using topological sorting, continuing exploration in an organized, structured manner. This iterative process ensures thorough coverage of flag-guarded branches, reducing redundant paths and minimizing performance overhead. By leveraging FDG-guided execution and targeted branch-solving, ALGERNON enables more directed and efficient exploration, significantly enhancing its ability to navigate complex program structures and uncover hidden vulnerabilities.

3) Seed Scheduling: ALGERNON optimizes exploration efficiency by dynamically prioritizing seeds based on the exploration status of flag-guarded branches. Seeds that sufficiently explore a specific flag-guarded branch have lower priorities, allowing ALGERNON to shift focus and resources toward unexplored branches. Additionally, seeds that uncover new flag edge coverage are labeled with higher priority, ensuring they are selected more frequently for mutation. This labeling helps ALGERNON focus on seeds likely to reveal untouched program paths or new flag-guarded branches, especially those involving unique flag variable combinations.

To improve the precision and effectiveness of this approach, ALGERNON periodically assesses seeds for potential relevance to unexplored flag-guarded branches, using FDG guidance to identify seeds covering critical paths toward target branches. If no current seed adequately reaches a target flag-assignment node, ALGERNON selects seeds that cover dependent nodes, maximizing the likelihood of progressing toward unexplored flag-checking conditions.

In the concolic execution module, ALGERNON employs the same priority-based strategy, where labeled seeds are prioritized to track and solve input dependencies related to flag variables. By focusing on labeled seeds during concolic

execution, ALGERNON efficiently directs exploration towards flag-dependent branches, enhancing the likelihood of reaching deeper, flag-related code paths. This integrated approach between fuzzing and concolic execution allows ALGERNON to dynamically adjust its targets and refine its seed pool, effectively balancing exploration coverage and precision to boost overall testing efficiency.

V. EVALUATION

In this section we evaluate the effectiveness of ALGERNON.

A. Experiment Setup.

Prototype Implementation. We implement ALGERNON using over 5,000 lines of C/C++ code and 500 lines of Python. We modify the clang-10 preprocessor and AST construction modules to extract flag constants from source code. Flag variable recognition is implemented in a custom LLVM Pass. We modify the CodeGen module to label flag variables in the LLVM IR, enabling us to instrument these labeled locations. We implement flag coverage feedback in AFL (commit-ID 6103710) to track changes in flag variable values. Finally, we use IDA Pro to construct the FDG of the target binary and modify QSYM to perform selective solving based on the FDG.

Baselines. We compare ALGERNON with QSYM [14], Angora [15], AFL++ [16] and INVSCOV [17]. It is noteworthy that current state-driven fuzzing tools are not included in our experiment, because they either are not open-source or only work on the protocol programs and are not applicable for normal programs. Due to the unavailability of the source code for GREYONE [19] and PATA [20], a comparison with these tools could not be conducted.

Benchmark. We chose programs from Unifuzz [6] and OSS-Fuzz [21], which are widely adopted for evaluating state-of-the-art fuzzers. The primary selection criterion is to cover various domains, including the processing of video, image, audio, PDF, and binary files. As a result, our benchmark consists of 20 popular programs with diverse categories.

Test Bed and Initialization.. We carry out all the following experiments on a Ubuntu 18.04 server with Intel Xeon CPU (Gold 5218, 64 cores) and 256 GB RAM. All tools are given the same timeout and the same initial corpus.

B. Flag Constants Identification Effectiveness

To the best of our knowledge, no public data set of flag constants exists, so we invited relevant domain experts to manually sample and verify the flag constants identified by ALGERNON. For each program, the experts randomly sampled 20 flag constants from ALGERNON's results. Two experts independently verified each sampled constant, and a third expert reviewed their findings. This process achieved a Cohen's Kappa [22] coefficient of 0.962, indicating strong agreement.

In Table I, the "Recognized" column indicates the number of flag constants successfully identified by ALGERNON. The "Sampled" column represents the number of flag constants reviewed by the experts. The "TP" column denotes the number of true flag constants verified by the experts, accompanied by the corresponding percentage. Across the 20 programs evaluated, ALGERNON successfully recognized a total of 18,250 flag constants. Experts sampled 400 of these, verifying 375 as true flag constants, yielding an accuracy rate of 93.75%.

TABLE I: Effectiveness of identified flag constants

Program	Version	#Recognized	#Sampled	#TP	%TP
pixdata	gdk-pixbuf 2.31.1	537	20	20	100%
imginfo	jasper 2.0.12	1,176	20	20	100%
lame	3.99.5	862	20	19	95%
mp3gain	1.5.2	125	20	20	100%
flvmeta	1.2.1	390	20	20	100%
mp42aac	Bento4 1.5.1-628	3,641	20	20	100%
cflow	1.6	799	20	20	100%
infotocap	ncurses 6.1	2,218	20	17	85%
jq	1.5	321	20	18	90%
mujs	2.0.2	362	20	16	80%
pdftotext	xpdf 4.00	785	20	19	95%
nm	binutils 5279478	491	20	20	100%
objdump	binutils 2.28	504	20	18	90%
tcpdump	4.8.1	2,266	20	18	90%
jhead	3.00	44	20	18	90%
faad2	1d53978	618	20	20	100%
faust	2.63.0	312	20	17	85%
fdkaac	03c3c60	88	20	17	85%
libxls	e3719f3	140	20	19	95%
picoc	3.2.2	572	20	19	95%
All	-	18,250	400	375	**93.75%**

Based on manual analysis, we found that the main reason for false positives in ALGERNON is the presence of consecutive constants that share the same prefix or suffix. This similarity can sometimes lead to misrecognition. For example, `LOG_ERROR` and `LOG_INFO` might be identified due to the common `LOG_` prefix. However, these typically represent configurations or external conditions, making them input-independent rather than internal program states with implicit input dependencies. It is important to emphasize that these false positives do not have a significant impact on the subsequent exploration process. While false positives may result in the recognition of additional flag variables, they do not cause any omissions in the program's control flow. Many of these false positives have little or no effect on the control flow at all, making the added overhead acceptable without hindering ALGERNON 's guidance for exploration

It is essential to balance precision and efficiency in the flag recognition. ALGERNON strives to accurately identify flag variables while minimizing the risk of missing potential flag-guarded branches, ensuring effective program exploration with an acceptable level of false positives.

C. Code Coverage

We ran all fuzzing tools with a 24-hour timeout and repeated each experiment three times to ensure reliability and reduce random variations. Table II shows the line and branch coverage achieved by each tool, along with ALGERNON 's improvement over the others. Among the tested programs, *jq* could not be compiled with INVSCOV due to a clang error in INVSCOV, so we omitted *jq* results with INVSCOV in this and the following experiments. Abnormal *jq* results with INVSCOV are also excluded from Table II and the related figures.

TABLE II: Code coverage of ALGERNON compared with other fuzzers

Program	Line Coverage					Branch Coverage				
	# ALGERNON	# Angora	# AFL++	# QSYM	# INVSCOV	# ALGERNON	# Angora	# AFL++	# QSYM	# INVSCOV
pixdata	4758	2815(+69.02%)	4446(+7.02%)	3818(+24.62%)	4187(+13.64%)	2608	1504(+73.40%)	2499(+4.36%)	2083(+25.20%)	2303(+13.24%)
imginfo	4042	3200(+26.31%)	3961(+2.04%)	3200(+26.31%)	3411(+18.50%)	2494	2034(+22.62%)	2453(+1.67%)	2400(+3.92%)	2174(+14.72%)
lame	5136	5061(+1.48%)	5068(+1.34%)	5139(-0.06%)	5060(+1.50%)	2646	2614(+1.22%)	2636(+0.38%)	2684(-1.42%)	2626(+0.76%)
mp3gain	2142	2086(+2.68%)	2092(+2.39%)	2093(+2.34%)	2095(+2.24%)	1070	1017(+5.21%)	1043(+2.59%)	1044(+2.49%)	1046(+2.29%)
flvmeta	655	113(+479.65%)	655(+0%)	655(+0%)	655(+0%)	322	58(+455.17%)	322(+0%)	317(+1.58%)	321(+0.31%)
mp42aac	3921	2694(+45.55%)	2524(+55.35%)	3714(+5.57%)	2389(+64.13%)	2589	1945(+33.11%)	1634(+58.45%)	2529(+2.37%)	1564(+65.54%)
cflow	1993	1948(+2.31%)	1992(+0.05%)	1976(+0.86%)	1993(+0%)	1248	1194(+4.52%)	1248(+0%)	1235(+1.05%)	1245(+0.24%)
infotocap	1886	1365(+38.17%)	1830(+3.06%)	1459(+29.27%)	1834(+2.84%)	1490	993(+50.05%)	1424(+4.63%)	1110(+34.23%)	1442(+3.33%)
jq	3501	3410(+2.67%)	3490(+0.32%)	3398(+3.03%)	N/A	2144	2032(+5.51%)	2128(+0.75%)	2032(+5.51%)	N/A
mujs	4705	3402(+38.30%)	4685(+0.43%)	4366(+7.76%)	4595(+2.39%)	2860	1857(+54.01%)	2828(+1.13%)	2556(+11.89%)	2728(+4.84%)
pdftotext	11038	9096(+21.35%)	10781(+2.38%)	9359(17.94%)	10182(+8.41%)	8008	6218(+28.79%)	7375(+8.58%)	6790(+17.94%)	7340(+9.10%)
nm	6420	6678(-3.86%)	5657(+13.49%)	5871(+9.35%)	3824(+67.89%)	3929	4100(-4.17%)	3458(+13.62%)	3433(+14.45%)	2216(+77.30%)
objdump	7940	5770(+37.61%)	7529(+5.46%)	9017(-11.94%)	5851(+35.70%)	5049	3412(+47.98%)	4870(+3.68%)	5584(-9.58%)	3508(+43.93%)
tcpdump	20141	7693(+161.81%)	18065(+11.49%)	13153(+53.13%)	17127(+17.60%)	14331	4605(+211.21%)	12471(+14.91%)	8570(+67.22%)	11518(+24.42%)
jhead	715	667(+7.20%)	313(+128.43%)	328(+117.99%)	313(+128.43%)	555	463(+19.87%)	213(+160.56%)	218(+154.59%)	212(+161.79%)
faad2	7253	6935(+4.59%)	7142(+1.55%)	7043(+2.98%)	7138(+1.61%)	3175	2991(+6.15%)	3096(+2.55%)	3062(+3.69%)	3090(+2.75%)
faust	2867	2831(+1.27%)	2868(-0.03%)	2701(+6.15%)	2829(+1.34%)	3482	3470(+0.35%)	3497(-0.43%)	3461(+0.61%)	3434(+1.40%)
fdkaac	1531	1360(+12.57%)	1375(+11.35%)	1453(+5.37%)	1376(+11.26%)	796	627(+26.95%)	647(+23.03%)	741(+7.42%)	646(+23.22%)
libxls	1358	1201(+13.07%)	1324(+2.57%)	1307(+3.90%)	1343(+1.12%)	876	790(+10.89%)	842(+4.04%)	790(+10.89%)	866(+1.15%)
picoc	3291	3036(+8.40%)	3287(+0.12%)	2722(+20.90%)	3092(+6.44%)	1864	1695(+9.97%)	1865(-0.05%)	1501(+24.18%)	1766(+5.55%)
Arithmetic Mean	4764.65	3568.05(+33.54%)	4454.20(+6.97%)	4164.65(+14.41%)	4173.37(+15.76%)*	3076.80	2180.95(+41.08%)	2827.45(+8.82%)	2609.65(+17.90%)	2633.95(+18.68%)*

The results in Table II demonstrate ALGERNON's significant advantage in terms of program code coverage. Overall, ALGERNON outperforms other tools across most tested programs, with improvements in line coverage of 6.97%, 15.76%, 14.41%, and 33.54% over AFL++, INVSCOV, QSYM, and Angora, respectively. It also improves branch coverage by 8.82%, 18.68%, 17.90%, and 41.08%, respectively.

In *objdump*, ALGERNON 's coverage is lower than QSYM's. This discrepancy stems from the nature of objdump's code paths and the limited flag-guarded branches in the running commands we used. Since ALGERNON is designed to identify and explore flag-guarded branches, programs with fewer such branches present fewer opportunities for guided exploration. In *objdump*, this scarcity of flag-guarded branches led to more time spent traversing the FDG, increasing overhead without a significant coverage improvement over QSYM's native exploration. Similarly, ALGERNON shows limited improvement in *flvmeta* and *lame*, performing similarly to QSYM and Angora. Manual analysis reveals that while these programs contain many flag variables, most flag-guarded conditions use custom-defined functions for comparisons rather than direct comparisons between flag constants and variables. This limits ALGERNON 's ability to identify and guide exploration effectively, resulting in only marginal gains.

Table III shows the p-value of the Mann-Whitney U test, comparing line coverage achieved by ALGERNON with QSYM, Angora, AFL++, and INVSCOV in our benchmark projects. The results reveal significant differences ($p < 0.05$) in line coverage for most projects, validating ALGERNON 's superiority over these fuzzers.

TABLE III: Mann-Whitney U test results on the line coverage for 24 hours over 3 runs

Program	QSYM	Angora	AFL++	INVSCOV
pixdata	4.04E-2	3.83E-2	9.52E-2	4.04E-2
imginfo	9.52E-2	3.83E-2	9.52E-2	4.04E-2
lame	9.20E-2	3.83E-2	3.83E-2	3.83E-2
mp3gain	3.83E-2	3.83E-2	3.83E-2	3.61E-2
flvmeta	4.07E-1	3.61E-2	1.77E-1	3.96E-1
mp42aac	9.20E-2	3.83E-2	3.83E-2	3.61E-2
cflow	1.88E-1	1.88E-1	8.88E-2	3.29E-1
infotocap	4.04E-2	3.83E-2	4.04E-2	3.18E-2
jq	1.88E-1	9.20E-2	9.20E-2	N/A
mujs	3.83E-2	3.83E-2	9.20E-2	9.20E-2
pdftotext	4.04E-2	4.04E-2	9.52E-2	3.83E-2
nm	9.52E-2	1.91E-1	9.52E-2	3.83E-2
objdump	3.83E-2	3.83E-2	4.04E-2	3.18E-2
tcpdump	4.04E-2	3.83E-2	9.52E-2	9.20E-2
jhead	3.83E-2	8.88E-2	3.83E-2	2.97E-2
faad2	4.04E-2	4.04E-2	4.04E-2	4.04E-2
faust	4.04E-2	4.04E-2	1.91E-1	4.04E-2
fdkaac	4.04E-2	4.04E-2	4.04E-2	4.04E-2
libxls	4.04E-2	3.83E-2	4.04E-2	4.04E-2
picoc	4.04E-2	4.04E-2	4.04E-2	4.04E-2

D. Flag-guarded Branch Coverage

To evaluate the effectiveness of ALGERNON in improving flag-guarded branch exploration, we specifically measured the coverage of flag-guarded branches in the program. We approximated the number of flag-guarded branches by counting the number of conditional statements marked with flag variables in the LLVM IR generated for each program. The experimental results are shown in Table IV.

For most programs, ALGERNON significantly outperformed baseline tools in flag-guarded branch coverage. On average, it increased flag-guarded branch coverage by 21.06%, 36.13%, 44.48%, and 66.42% compared to AFL++, INVSCOV, QSYM, and Angora, respectively. ALGERNON 's improvements are particularly notable in programs with a high number of flag variables. For instance, when testing *mp42aac*, ALGERNON increased flag-guarded branch coverage by 292.00%, 292.00%, 172.22%, and 237.93% over AFL++, INVSCOV, QSYM, and Angora, respectively. This indicates that ALGERNON effectively guides exploration through code paths involving flag-

TABLE IV: Flag-guarded branch coverage of ALGERNON compared with other fuzzers

Program	# ALGERNON	# Angora	# QSYM	# AFL++	# INVSCOV
pixdata	141	79	97	112	105
imginfo	144	66	88	105	78
lame	91	69	69	69	69
mp3gain	16	12	12	12	12
flvmeta	35	12	25	26	28
mp42aac	98	29	36	25	25
cflow	268	215	217	217	217
infotocap	34	12	12	15	15
jq	248	227	226	227	N/A
mujs	146	54	107	112	109
pdftotext	109	41	49	64	55
nm	16	9	9	9	9
objdump	739	472	601	597	571
tcpdump	1067	327	558	982	814
jhead	89	50	32	30	30
faad2	233	201	194	203	203
faust	216	192	172	201	190
fdkaac	35	13	17	11	10
libxls	110	88	96	93	109
picoc	407	381	319	394	376
Average	212.10	127.45	146.80	175.20	159.21
Increase	-	66.42%	44.48%	21.06%	36.13%*

TABLE V: Unique crashes running for 24 hours achieved by various fuzzers when testing real-world programs

Program	QSYM	Angora	AFL++	INVSCOV	ALGERNON
pixdata	6	0	6	17	23
imginfo	5	0	0	0	0
lame	3	0	4	3	4
mp3gain	7	3	9	6	7
flvmeta	4	0	4	4	3
mp42aac	5	8	2	1	7
cflow	3	0	4	4	5
infotocap	1	0	2	2	1
jq	2	0	0	N/A	0
mujs	1	0	1	2	1
pdftotext	1	0	3	7	14
nm	0	0	0	0	0
objdump	9	1	4	2	2
tcpdump	0	0	1	4	26
jhead	4	2	1	2	15
faad2	0	0	0	0	2
faust	0	0	0	0	1
fdkaac	1	1	1	1	2
libxls	2	1	1	1	7
picoc	17	9	12	10	32
Total	71	25	55	66	152

guarded branches, enabling a more comprehensive analysis of program behavior and increased coverage.

E. Bug Detection

1) Unique Bugs: To assess the impact of the coverage improvements achieved by ALGERNON on vulnerability discovery, we conducted a comparative analysis with other fuzzers. We compiled the programs in our benchmark with AddressSanitizer (ASAN) and fuzzed them for 24 hours. The experiment was also repeated three times to ensure robustness and mitigate the influence of random variations. It is important to note that many crashes reported by the fuzzers were duplicates; we de-duplicated results to extract unique bugs for statistical analysis, following existing work [23].

The results, shown in Table V, demonstrate that ALGERNON outperformed the baselines in terms of unique bug discovery. Specifically, ALGERNON uncovered 152 unique bugs, surpassing the 66, 55, 71, and 25 unique bugs detected by INVSCOV, AFL++, QSYM, and Angora, respectively. Notably, ALGERNON consistently identified the highest number of unique bugs across most of the evaluated programs. The ability of ALGERNON to uncover a larger number of unique bugs highlights its effectiveness in detecting and exposing hidden security weaknesses.

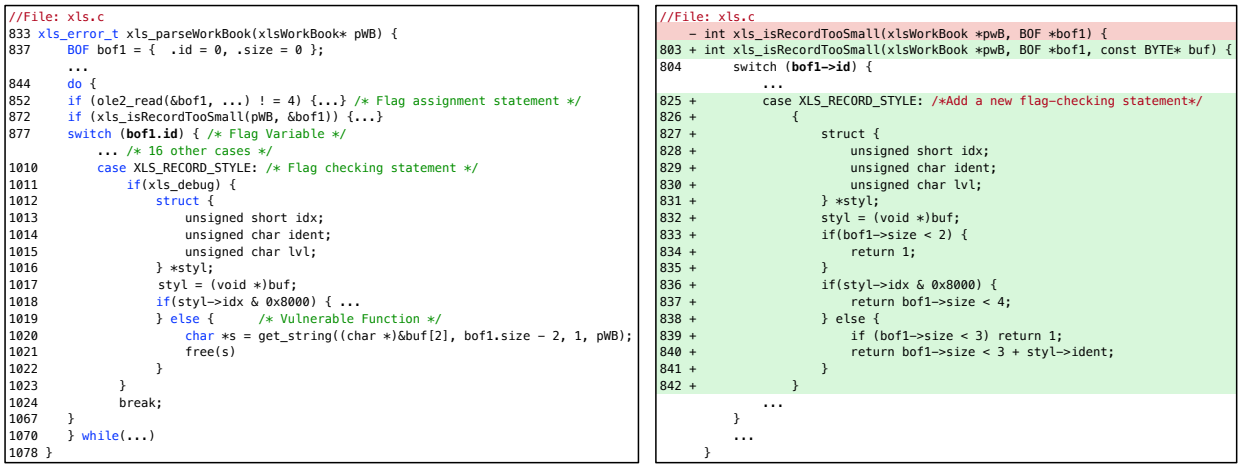
2) Zero-day Vulnerability Discovery: Table VI shows the zero-day vulnerabilities discovered by ALGERNON. It discovered a total of 37 zero-day vulnerabilities, 30 of which were exclusively detected by it. We reported all the vulnerabilities to the respective manufacturers and assisted in remediation. As of the writing of this paper, 11 of these vulnerabilities have been confirmed with CVE IDs. Our manual analysis shows that most of these vulnerabilities are related to flag variables and were challenging for baseline fuzzers to detect.

Conclusion. Our experimental results in Section V-C, V-D, and V-E demonstrate that solving flag-guarded constraints is crucial because it enables the fuzzer to systematically trigger branches that would otherwise be unreachable through random mutation or naive symbolic reasoning. These branches often protect complex logic or rare behaviors, where bugs are more likely to hide. By explicitly reasoning about and satisfying these constraints, fuzzers can significantly increase both code coverage and the likelihood of discovering deep, hidden bugs. To further illustrate how flag variable knowledge aids ALGERNON in program exploitation, we present a case study on one of the zero-day vulnerabilities it uncovered.

Case Study: Heap Buffer Overflow in libxls. Libxls [24] is a widely used library for parsing Excel binary file formats. When tested with ALGERNON, we found seven vulnerabilities. Until the writing of this paper, all have been fixed by the developers, with six assigned CVE IDs. Figure 7(a) shows the vulnerable code for a heap buffer overflow, and Figure 7(b) shows the official patch provided by the developers.

This vulnerability is deeply embedded within the program logic, requiring specific flag variable related conditions to trigger. The input must satisfy several flag-guided branches, with a critical check at line 1010 in Figure 7(a). Line 1010 verifies if the flag variable `bof1.id` is equal to the flag constant `XLS_RECORD_STYLE`. If this condition is met, the program can proceed to the vulnerable function `get_string()` at line 1020, where the vulnerability is triggered. However, reaching this point is complex. The flag-checking statement at line 1010 is within a do-while loop at line 844, and manual analysis indicates that the loop must iterate over forty times—executing over forty flag-checking statements—before reaching line 1010. The security patch shown in Figure 7(b) confirms that the vulnerability relates to the flag variable `bof1.id`, it adds necessary sanity checks before calling the vulnerable function.

Existing fuzzing methods struggle to identify such deeply hidden vulnerabilities because they lack awareness of flag



```
//File: xls.c
833 xls_error_t xls_parseWorkBook(xlsWorkBook* pWB) {
837     BOF bof1 = {  .id = 0, .size = 0 };
        ...
844     do {
852     if (ole2_read(&bof1, ...) ! = 4) {...} /* Flag assignment statement */
872     if (xls_isRecordTooSmall(pWB, &bof1)) {...}
877     switch (bof1.id) { /* Flag Variable */
            ... /* 16 other cases */
1010        case XLS_RECORD_STYLE: /* Flag checking statement */
1011            if(xls_debug) {
1012                struct {
1013                    unsigned short idx;
1014                    unsigned char ident;
1015                    unsigned char lvl;
1016                } *styl;
1017                styl = (void *)buf;
1018                if(styl->idx & 0x8000) { ...
1019                } else {      /* Vulnerable Function */
1020                    char *s = get_string((char *)&buf[2], bof1.size - 2, 1, pWB);
1021                    free(s)
1022                }
1023            }
1024            break;
1067    }
1070    } while(...)
1078 }
```

(a) The vulnerable code

```
//File: xls.c
    - int xls_isRecordTooSmall(xlsWorkBook *pwB, BOF *bof1) {
803 + int xls_isRecordTooSmall(xlsWorkBook *pwB, BOF *bof1, const BYTE* buf) {
804         switch (bof1->id) {
                ...
825 +           case XLS_RECORD_STYLE: /*Add a new flag-checking statement*/
826 +               {
827 +                   struct {
828 +                       unsigned short idx;
829 +                       unsigned char ident;
830 +                       unsigned char lvl;
831 +                   } *styl;
832 +                   styl = (void *)buf;
833 +                   if(bof1->size < 2) {
834 +                       return 1;
835 +                   }
836 +                   if(styl->idx & 0x8000) {
837 +                       return bof1->size < 4;
838 +                   } else {
839 +                       if (bof1->size < 3) return 1;
840 +                       return bof1->size < 3 + styl->ident;
841 +                   }
842 +               }
                ...
            }
            ...
        }
```

(b) The developer's patch to fix the vulnerability

Fig. 7: A heap buffer overflow vulnerability detected by ALGERNON and official patch used to fix it.

TABLE VI: Zero-day vulnerabilities found by ALGERNON

Program	Version	#0-day vulnerabilities	#Fixes	#CVEs
mp42aac	ab5591e	1 (1)	1	0
cflow	1.7	1 (1)	0	0
picoc	3.2.2	23 (17)	0	0
fdkaac	03c3c60	2 (2)	2	2
faust	2.63.0	1 (1)	0	1
faad2	1d53978	2 (2)	2	2
libxls	e3719f3	7 (6)	7	6
Total	-	37 (30)	12	11

[1] Numbers in parentheses indicate those that were found only by ALGERNON.

variables and their dependencies. Through its targeted identification of flag variables and guided hybrid fuzzing, ALGERNON efficiently detected this vulnerability.

F. Limitations

While ALGERNON demonstrates strong effectiveness, it has some limitations that may impact its practical use.

Firstly, in the flag variable identification phase, ALGERNON identifies flag constants and then recognizes flag variables based on their assignments. Although this approach works well for most real-world scenarios, it may have limitations in non-standard cases where developers assign values to flag variables using character constants, integer constants, or bitwise operations. We could address this by using lightweight taint tracking to handle such cases.

Secondly, although our guiding strategy shows better results compared to other tools, it may not fully exploit the potential of flag variables. Future research should focus on developing more efficient guiding strategies for the hybrid fuzzing module. This could include exploring intelligent path selection algorithms or using machine learning approaches to improve the exploration of flag-guarded branches and enhance vulnerability detection.

VI. RELATED WORK

Hybrid Fuzzing. To better solve branch conditions, some works have proposed combining fuzzing with symbolic execution, leveraging the ease of use and efficiency of fuzzing as well as the input-solving capabilities of symbolic execution to meet specific branch conditions. Driller [25] was the earliest attempt to combine the fuzzing tool AFL with the symbolic execution engine angr [26]. However, due to inadequate environmental modeling and poor maintenance, it is not suitable for testing real-world programs. QSYM [14] also combines fuzzing with symbolic execution, while the two parts can run simultaneously, improving testing efficiency and solving the problem of environmental modeling, making it compatible with real-world programs. However, QSYM can only collect the branch constraints explicit dependent on input and cannot solve flag-guarded branch constraints. Angora [15] abandons heavyweight symbolic execution and constraint solving, using taint tracking and gradient descent algorithms to solve constraints. Nevertheless, this approach still encounters the challenge that implicit data flows cannot be traced by taint analysis methods, resulting in the inability to solve flag-guarded branch conditions.

The state-of-the-art works and research of hybrid fuzzing mainly focus on improving the interaction efficiency between fuzzing and symbolic execution and reducing the cost of constraint solving, which are orthogonal with ALGERNON. DigFuzz [27] collects the execution frequency of branches in fuzzing and allows symbolic execution to explore low-frequency branches. Pangolin [28] reduces unnecessary duplicate calculations by reusing values previously solved by the constraint solver, improving solving efficiency. CoFuzz [29] improves hybrid fuzzing's scheduling strategy and coordination mode to achieve more efficiency. Besides, existing constraint-solving techniques [30], [31] do not apply to solving flag-guarded branches. This is due to the implicit data dependency between flag variables and the input, which makes it

difficult for symbolic execution engines to collect constraint information and solve these branches effectively.

State-driven Fuzzing. Existing state-driven fuzzing works [5], [1], [3], [2], [11] mainly focus on testing protocol libraries or the Linux kernel, without paying attention to normal programs. Since the concept of state in normal programs is relatively ambiguous, most works cannot identify the states in them. Flag, as a special state, is naturally difficult to address through existing work. AFLNET [5] and SGFUZZ [1] both require well-defined states in the program, and these works focus on state transitions while ignoring the combination of variables, which makes it difficult to deal with flag-guarded branches. IJON [4] requires manual annotation to label the program states, but the program contains a large number of flags, which makes it difficult to annotate them manually. INVSCOV [17] divides program states by identifying likely invariants in the program, which is similar to our approach. However, INVSCOV focuses on invariant checks violations, while we focus on whether the flag variables and their combinations are satisfied or not. DSFUZZ [13] proposes a directed fuzzing approach to reach deep program states and vulnerabilities, however, it still relies on random mutation strategies to satisfy the state-related branches, which limits its effectiveness.

Coverage Guided Fuzzing State-of-the-art coverage guided fuzzing tools [29], [32], [33], [34], [35] struggle to efficiently explore flag-guarded branches, as flag variables hold a set of compile-time constants that are implicitly dependent on the input but not directly loaded from it. This implicit data flow limits these tools from using taint analysis or symbolic execution to aid in exploration. The input-tracking methods adopted by existing works, i.e., dynamic taint analysis [15], [36], [37] and dynamic taint inference [19], [20], [38], [39], [40], are unaware of the fact that the flag variables are input-determined, so they fail to explore flag-guarded branches. This problem cannot be solved by extending the existing taint analysis since it can easily cause serious over-taint problems. While GreyOne [19] can infer the relationship between input bytes and variables, it cannot capture variable combinations or control dependencies. The reliance of flag variables on input, along with the need to identify precise combinations and control dependencies, highlights the need for advanced techniques to effectively handle these complexities.

VII. Conclusion

In this paper, we propose a novel flag-guided hybrid fuzzing approach ALGERNON. It identifies the flag variables in the program and proposes a new data structure, FDG, to model the dependencies between flag assignment and checking statements. Based on FDG, ALGERNON introduces flag edge coverage as feedback to drive the fuzzer. In the solving module, FDG guides the branch-solving process and assigns priorities to seeds, optimizing flag-guarded branch exploration. We evaluated ALGERNON 's effectiveness through extensive testing on real-world programs. Results show that ALGERNON significantly outperforms popular testing tools. It discovers 37 zero-day vulnerabilities, and 11 have been assigned CVE-IDs.

Acknowledgments

We would like to thank anonymous reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172104, 62172105, 62202106, 62302101, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

References

[1] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3255–3272.

[2] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, 2022.

[3] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "Statefuzz: System call-based state-aware linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3273–3289.

[4] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1597–1612.

[5] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[6] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers." in *USENIX Security Symposium*, 2021, pp. 2777–2794.

[7] Wikipedia contributors, "Data dependency — Wikipedia, the free encyclopedia," 2023, [Online; accessed 14-September-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Data_dependency&oldid=1174645850

[8] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[9] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[10] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols."

[11] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 999–1010.

[12] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 720–730.

[13] Y. Liu and W. Meng, "Dsfuzz: Detecting deep state bugs with dependent state exploration," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1242–1256.

[14] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[15] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[16] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10.

[17] D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[18] M.Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2024.

[19] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing." in *USENIX Security Symposium*, 2020, pp. 2577–2594.

[20] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*, 2022, pp. 154–170.

[21] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," 2017.

[22] wikipedia, "Cohen's kappa wikipedia," https://en.wikipedia.org/wiki/Cohen\%27s_kappa, 2024.

[23] P. Deng, Z. Yang, L. Zhang, G. Yang, W. Hong, Y. Zhang, and M. Yang, "Nestfuzz: Enhancing fuzzing with comprehensive understanding of input processing logic," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1272–1286.

[24] Github, "libxls," https://github.com/libxls/libxls, 2024.

[25] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[26] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[27] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." in *NDSS*, 2019.

[28] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1613–1627.

[29] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and improving hybrid fuzzing," in *Proceedings of the 45th International Conference on Software Engineering, ser. ICSE*, vol. 23.

[30] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 18–35.

[31] X. Liu, W. You, Z. Zhang, and X. Zhang, "Tensilefuzz: facilitating seed input generation in fuzzing via string constraint solving," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 391–403.

[32] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, "Ems: History-driven mutation for coverage-based fuzzing," in *29rd Annual Network and Distributed System Security Symposium, NDSS*, 2022, pp. 24–28.

[33] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "Darwin: Survival of the fittest fuzzing mutators," *arXiv preprint arXiv:2210.11783*, 2022.

[34] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2194–2211.

[35] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen, "Path transitions tell more: Optimizing fuzzing schedules via runtime program states," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1658–1668.

[36] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[37] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[38] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 499–513.

[39] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[40] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 769–786.