OSmart: Whitebox Program Option Fuzzing

Kelin Wang[*†]
Mengda Chen[*]
Liang He[‡]
Purui Su[‡§¶]
TCA, Institute of Software, Chinese Academy of Sciences
Beijing, China

Yan Cai
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China

Jiongyi Chen
Bin Zhang
Chao Feng
Chaojing Tang
College of Electronic Science and Technology, National University of Defense Technology
Changsha, Hunan, China

Abstract

Program options are ubiquitous and serve as a fundamental mechanism for configuring and customizing software behaviors. Given their widespread use, testing program options becomes essential to ensure that the software behaves as expected across various configurations. Existing option-aware fuzzers either mutate options as if they were standard program inputs or employ NLP techniques to deduce relationships among options from the documentation. However, there has not been a whitebox approach that generates option combinations by capturing the inherent execution logic of the program.

This paper presents OSmart, a whitebox approach designed to systematically extract program options and effective option combinations that precisely encapsulate the intrinsic execution logic of the program, incorporating both data dependency and control dependency. OSmart successfully inferred 12,560 option combinations from 56 programs. Additionally, OSmart uncovered that more than 67% of evaluated programs have undocumented options. By integrated with AFL++, OSmart discovered 40.3% more paths, which led to the detection of 51 new bugs and the assignment of 18 CVE IDs. Finally, we also compared OSmart with four state-of-the-art option-aware fuzzers on a public benchmark and our tool achieved higher line coverage in 66.7% (20/30) of the evaluated programs.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

option extraction; option impact; option group

[*]The two lead authors contributed equally to this work.
[†]Also with Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University.
[‡]corresponding authors.
[§]Also with Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences.
[¶]Also with School of Cyber Security, University of Chinese Academy of Sciences.

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690228

ACM Reference Format:
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 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690228

1 Introduction

Fuzzing has been yielding promising results in identifying security vulnerabilities by exploring program paths and triggering crashes. While the fuzzing tools typically pursue coverage-driven strategies, they often employ a fixed set of program options that specify certain runtime program behaviors and result in biased execution logic. This is particularly notable when the programs are provided with abundant program options. As an example, `nasm` [35] accepts 44 options that can drive users to different functionalities.

On the basis of existing fuzzing frameworks, option-aware fuzzing provides varied option combinations and explores program paths associated with different options. Its objective lies in strategically combining options by accurately capturing the dependency among them, without exhaustively enumerating all possible combinations. While maintaining a manageable number of test cases during fuzzing, awareness of program options could enable effective navigation to more diverse execution paths.

In order to achieve option-aware fuzzing, several studies have been presented to generate option combinations [2, 22, 28, 49, 52, 53]. One involves constructing option dependency by observing the differences in execution paths under different option configurations [28]. AFL adopts the most straightforward solution that indiscriminately mutates program options and program inputs [2, 22]. CrFuzz tries to enumerate all documented options and their combinations, which unavoidably suffer combination explosion problems in the presence of tens or hundreds of program options [45]. ConfigFuzz mutates options and input simultaneously based on option relationships extracted from documents manually [52, 53]. CarpetFuzz [49] is the first work adopting NLP-based techniques [8, 18, 36] to identify the potential relationships between options. *So far, option-aware fuzzing has not been accomplished in a whitebox manner. Existing solutions all fail to precisely model the option dependency by capturing the intrinsic characteristics from the program itself, leaving a large input space for option combinations.*

In this paper, we present OSmart, a whitebox approach to systematically extract valid options and effective program combinations based on precise data-dependency and control-dependency analysis. Firstly, we use heuristic rules to precisely extract program options and their affected variables as the sources of dependency analysis. Then, to generate effective option combinations that accurately encapsulate program execution logic, we perform a fine-grained inter-procedural analysis on a new graph-based program representation, *Option Impact Graph*. Additionally, we take steps on option propagation and option combining, such as decomposing the conditional statements, tracking the dependencies between global and structure variables, and applying the Cartesian product to ensure more precise combinations, which mitigates the combination explosion problem. In the end, the option combinations are integrated into fuzzers with two mutation strategies.

We implemented a prototype called OSmart and evaluated it on 56 programs. OSmart successfully inferred 12,560 effective option combinations and uncovered that more than 67% of evaluated programs have undocumented options. We also evaluated OSmart in terms of path coverage by integrating OSmart into AFL++. The results show that OSmart discovered 40.3% more paths than AFL++, which led to the detection of 51 new bugs and the assignment of 18 CVE IDs. Compared with four state-of-the-art option-aware fuzzers on a public benchmark, OSmart outperformed 66.7% (20/30) programs in line coverage.

In summary, the main contributions of this work include:

- **New Perspective.** To the best of our knowledge, this paper presents the first whitebox approach to extract effective option combinations that converge dependencies to achieve option-aware fuzzing.
- **New Techniques.** We present several new techniques in the paper, decomposing option parsing on AST, fine-grained inter-procedural flow analysis, and sound option combination methods, which are operated on a new graph-based program representation called option impact graph.
- **Implementation and Evaluation.** We implemented a prototype system, OSmart, and evaluated it on 56 real-world programs. The evaluation results demonstrated the efficiency and effectiveness of our methods. We will release the source code of our prototype tool at https://github.com/osmart-source/osmartsource.

Responsible Disclosure: When we found the bugs, we immediately reported them to the developers. We provided root cause analysis for some vulnerabilities as well, so that the developers could fix them faster. At the time of this writing, 22 bugs have been fixed.

2 Background

In this section, we explain the concepts used in this paper with a code example, illustrate why existing approaches fail, and give the problem scope of this research.

2.1 Preliminaries

In Figure 1, the demo code contains three functions: `help`-function listing the documented option, `sum`-function calculating the summation of two variables, and `main`-function presenting the option

```
1. void help() {
2.   //Documented Options
3.   printf("-a: set debug flag and string name");
4.   printf("-t: add delta length");
5. }
6. int sum(int x, int y) {
7.   return x+y;
8. }
9. void main(int argc, char** argv) {
10.    char name[2048], fileName[1024];
11.    int  len, flag=0, doSum=0, delta=0;
12.    //Option Parsing
13.    while ((c=getopt(argc, argv, "a:st:"))!=-1) {
14.       switch (c) {
15.          case 'a':
16.            flag = 1;
17.            strncpy(name,optarg,2048);
18.            break;
19.          case 's': /*Undocumented Option!*/
20.            doSum = 1;
21.            break;
22.          case 't':
23.            delta = atoi(optarg);
24.            break;
25.          default:
26.            help();
27.       }
28.    }
29.    //Option Impacts
30.    if (flag) {
31.       len = sizeof(fileName);
32.       if (doSum) {
33.          /*Integer Overflow with [-a, -s, -t]!*/
34.          len = sum(len, delta);
35.       }
36.       /*Buffer Overflow with [-a, -s, -t]!*/
37.       snprintf(fileName, len, "%s", name);
38.    }
39. }
```

Figure 1: Motivating Example.

parsing statements and option impact areas. The core concepts are given below:

- **Documented Options (DOs) and Undocumented Options (UOs).** In Lines 12-28, the program accepts three options, "-a", "-s", and "-t". Specifically, programmers construct a *while*-loop structure containing a *switch-case* sub-structure to pick an option through `argv` by the `getopt`-function [13]. Here, we refer to the options as *documented options*, e.g., "-a" and "-t", listed in the `help`-function. However, for some specific reasons (e.g., *internal testing*, explained by the developers [31]), the option "-s" is missed in `help`-function, and we refer to it as an *undocumented option*.
- **Option Types (OT).** First, for the options owning option values, we classify them into two types: *numeric* and *string*. Specifically, in our example, we say "-a" is a string option as a `strncpy()` is used to accept its option value `optarg` in Line 17. Second, for an option having no option value, in this work, we say its type is ϵ, e.g., the "-s". Actually, identifying the option types can efficiently facilitate the fuzzing procedure as it can reasonably reduce the searching spaces by providing proper initialization values and selecting the right mutation strategy. Note that while most existing works manually confirm the option types by reading the documents, we

Table 1: A Brief Comparison with Existing Works.

Existing Works	Option Extract	Option Grouping	Documented Option	Undocumented Option	Option Type	Option Impacts	Option Group	Bugs
AFL++-argv* [22]	generation-based	generation-based	✓✗	✓✗	✗	✗	✓✗	✓✗
CrFuzz [45]	document-based	enumeration	✓	✗	Manual	✗	✓✗	✗
POWER [28]	document-based	random	✓	✗	Manual	✗	✓✗	✗
ConfigFuzz [52, 53]	document-based	grammar-based	✓	✗	Manual	✗	✓✗	✗
CarpetFuzz [49]	document-based	NLP-based	✓	✗	Manual	✗	✓✗	✗
OSmart	code-based	impact-based	✓	✓	✓	✓	✓	✓

*✓✗=unpractical or incomplete.

try to automatically infer their types based on some heuristic rules (see §4.2.2).

- **Option Impacts (OI).** Program options impact the assignment of variables and the execution of code through the passing of data and control conditions. Variables that are directly assigned (or altered in the function) in option parsing (i.e., `flag` and `name` in Line 16-17, `doSum` in Line 20, `delta` in Line 23) are referred to as *direct-impact variables* (DIVs) impacted by options. Through data flow and control flow, these variables are then passed on to others, which are referred to as *indirect-impact variables* (IIVs). In Figure 1, with "-a" provided, the `flag` is set to 1, which will ensure the execution of Line 31. Then, we can say "-a" has an impact on Line 31.
- **Option Group (OG).** In Figure 1, we can combine the options, "-a" and "-s", as an option group, considering both of their impacts on the same Line 32. Similarly, we can infer option groups based on the impact analysis, and we will formally elaborate it in §3. Note that for consistency, we also treat a single option as an option group.

Actually, in our example, there exist two security bugs: *integer overflow* in Line 34 and *buffer overflow* in Line 37. As shown in Figure 1, the two bugs can only be triggered if and only if the three options, "-a", "-s", and "-t", are provided as an option group at the same time. Specifically, with "-a" (setting `flag` to 1), we can ensure the execution of Line 31, 32, 37. Besides, with "-s", we can enable the execution of Line 34. Moreover, by providing "-t" with a proper option value, we can finally trigger the two security bugs. In other words, if we cannot provide the three options at the same time during fuzzing, we can hardly identify the bugs.

2.2 Why Current Approaches Fall Short

Currently, researchers have proposed two main types of option-aware fuzzers: the fuzzers, like AFL++-ARGV [2] and ConfigFuzz [52], which mutate the arguments and options together, and the fuzzers, like CrFuzz [45], POWER [28], and CarpetFuzz [49], which only mutate the arguments with a fixed option combination. However, all of the existing methods have limitations in option extraction and option combination. We present the detailed comparisons in Table 1.

- **Unpractical and Incomplete Option Extraction.** The existing ways to extract potential options can be summarized into two categories, i.e., generation-based and document-based. For generation-based methods, generating a legal option is time-consuming. For example, by using AFL++-ARGV [2], it could not produce a proper long option (containing only five characters) within 24 hours. On the other hand, for document-based methods, they directly extract options from documentation. However, when the options in the documentation and the code do not match, they could only extract incomplete options, e.g., "-s" in Figure 1.
- **Imprecise Option Combination.** To achieve high code coverage, existing works adopt various solutions to combine options. Specifically, POWER [28] leverages the relevance strategy to combine option configurations, and CrFuzz [45]'s strategy is to enumerate all potential option combinations. However, they do not take into account option dependencies, and the number of random combinations can be exponential. ConfigFuzz [52, 53] manually extracts the option-related grammars and labels option relationships from the documentation as fuzzing templates, but it relies on manual effort. CarpetFuzz [49] adopts NLP techniques to automatically identify the option relationships in the documentation. However, a few option relationships are documented, and there are incorrect records. In our dataset, only 21/56 programs have dependencies recorded in the documentation. Moreover, triggering the bugs in Figure 1 requires option combination ['a','s','t'], but the documentation does not reflect such an option dependency. Therefore, coupled with incomplete extraction, these works cannot discover the bugs in Figure 1.

2.3 Research Goals and Problem Scope

As the undocumented option problem exists in most option-aware fuzzers, our *first goal* is to extract the complete option lists automatically. Besides, our *second goal* is to combine options in a more reasonable and fine-grained way. Finally, our *last goal* is to propose new fuzzing strategies to clearly identify the potential impacts of program options. While we only focus on the command-line options, our approach is applicable to other kinds of options, e.g., configuration files or environments. Besides, we only take care of the option types and pick simple initialized option values to set up fuzzing.

3 Option Impact Graph

To facilitate the extraction of options and the conduction of option dependency analysis, in this work, we propose a new *property graph* [5] based representation, called Option Impact Graph, which

is combined with the abstract syntax tree (AST) and the program dependence graph (PDG). The step of option extraction (§4.2) and the step of option impact analysis (§4.3) allow option impact graph to gradually take shape and assign the option groups as its node property. Besides, we also propose graph traversals [16], which can be used to filter the nodes according to specific node or edge attributes, conclude all the nodes impacted by an option, and count option group results from multiple dimensions. Formal definitions of *option impact graph* are given below.

3.1 Definitions

Definition 1. As defined in the work [5], a *property graph* $G = (V, E, \lambda, \mu)$ is a directed graph, where V is the node set, E is the directed edge set, and $\lambda : E \longrightarrow \sum$ is an edge-type mapping function which can assign different labels from $\sum$ to each edge. The key part of the property graph is the function $\mu : (V \times E) \times K \longrightarrow S$, which can attach different properties to nodes and edges, where K is the property keys and S is the property values.

Definition 2. As defined in the work [16], an *AST-transformed* property graph $G_A = (V_A, E_A, \lambda_A, \mu_A)$ is a specific property graph where V_A and E_A are given by AST, and $\sum_A = \{AST\}$, and $K_A = \{code, line_number\}$, and the S_A contains the corresponding *operators*, *operands*, and *natural numbers*.

Definition 3. As defined in the work [16], a *PDG-transformed* property graph $G_P = (V_P, E_P, \lambda_P, \mu_P)$ is a specific property graph where V_P and E_P are given by PDG, and $\sum_P = \{C, D\}$ (control and data dependency), and $K_P = \{symbol, condition, line_number\}$, and the S_P contains the corresponding variable symbols, *true* or *false* condition value, and *natural numbers*.

Definition 4. In this work, we define an *option impact graph* $G_O = (V_O, E_O, \lambda_O, \mu_O)$ as a specific property graph with

- $V_O = V_{OPT} \cup V_{DIV} \cup V'_P$
- $E_O = E_{OPT} \cup E'_P$
- $\lambda_O : E_O \longrightarrow \{OPT, C, D\}$
- $K_O = K_A \cup K_P \cup \{OG\}$
- $S_O = S_A \cup S_P \cup \mathcal{P}(O)$

where V_{OPT} represents *option nodes* containing option names, and V_{DIV} represents the *DIV nodes* containing DIVs (both of them are extracted from V_A), and V'_P represents other nodes extracted from V_P based on DIV dependency analysis, and $E_{OPT} = \{(v_i, v_j) | v_i \in V_{OPT} \,\&\&\, v_j \in V_{DIV}\}$, and E'_P represents other edges extracted from E_P also based on the DIV dependency analysis, and OG represents a new property key, i.e., Option Group, and $\mathcal{P}(O)$ represents the power set of the option set O.

Property: Option Group. As introduced in §2.1, we use an *option group* to represent a combination of options that have specific impacts (control or data dependency) on the same node in the option impact graph. To represent the potential option groups for each node, we introduce a new property key, *OG*. Besides, to be able to describe different option impacts, we represent each option in the *OG* using the format "$o_i : t_i$" where o_i means the i_{th} option in the group, and t_i means the type of o_i.

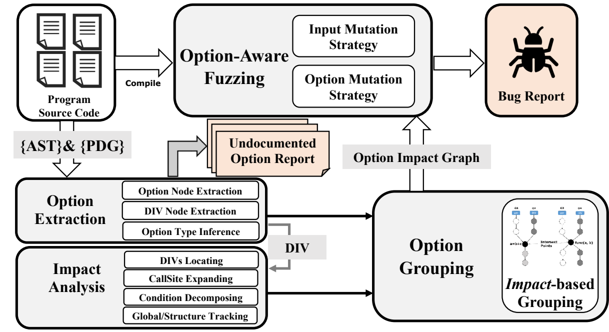

Figure 2: OSmart Framework.

3.2 Traversals

As defined in the work [16], a property graph traversal is a mapping function $T{:}P(V){\longrightarrow}P(V)$, where V is the node set and P is the power set of V. Moreover, one T can be combined with another, and we use $T_1 \circ T_2$ to represent the combination.

In this paper, we utilize the basic traversal and construct three types of traversals. The first kind is the filter traversal, $T_{V(k:s)}$, which is used to filter out the nodes according to the condition $\mu(k) = s$. The second is the predecessor traversals, $T_{Pre}(V)$, and the successor traversals, $T_{Succ}(V)$, to get the predecessor or successor nodes of V. The third is the impact traversals, $T_I(V)$, to traverse from V to all the nodes of the PDG that are within the option impact of V. Their definitions are provided in Appendix B.

4 Design

4.1 Overview

In this work, we propose OSmart framework, which consists of four main components, as shown in Figure 2. First, we will transform the program source code into AST-transformed and PDG-transformed property graphs, i.e., G_A and G_P. Then, in §4.2, based on G_A, we introduce a heuristic-based method to automatically extract option nodes V_{OPT}, followed by the DIV nodes V_{DIV} identification and type inference. In §4.3, we conduct an inter-procedural impact analysis to extract all other nodes and edges of option impact graph. Next, in §4.4, we will introduce our cartesian-product-based algorithm, which can automatically calculate the option groups to complete the option impact graph generation. Finally, in §4.5, we propose two fuzzing strategies to detect potential security bugs triggered by different option groups.

4.2 Option Extraction

After manually dissecting plenty of option parsing implementations, we find out most of the parsing procedures have exposed a general *code pattern*, command line inputs are compared sequentially within a loop structure and operations such as assignments or function calls are performed when matching the option. Based on the pattern, we propose a heuristic rule to extract all valid options automatically.

Meanwhile, we also realize that a specific code block will be executed after matching the corresponding option, e.g., in Figure 1, Line 16-18, Line 20-21, and Line 23-24 after matching the option "-a", "-s" and "-t" respectively. Within these blocks, some critical variables

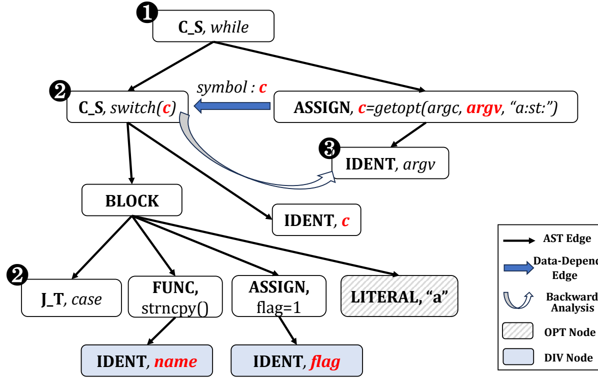

Figure 3: Option Extraction. (Omitting unnecessary nodes.)

are updated by option values or constant values. We identify these assigned variables as DIVs and automatically infer option types based on the assignment modes.

4.2.1 ***Option Node Extraction***. Basically, option parsing is a procedure to deal with the input arguments, *argv*, of the `main` function for C/C++ programs. The corresponding option names and values are arranged adjacent to each other in the *argv* array. Therefore, the key insight of our method is to identify the parsing code that processes *argv* by using a heuristic **Loop-Select-Target** (LST hereafter) rule. Specifically, the option parsing lines (Line 13-28) in Figure 1 can be divided into the following three parts:

❶ **Loop** ["*while*", "*for*"]: is used to iterate all of the items in the *argv* array. To detect the loop structure in the source code, we can use the filter traversal $T_{V_A(code:\text{"}C_S, while/for\text{"})}$ on all nodes in V_A and treat it as a loop node if its *code* attribute is *CONTROL_STRUCT* (**C_S** in Figure 3), which contains *"while"* or *"for"* characters. Then, for each loop node, if the following two kinds of nodes (*select* and *target*) can be detected in the control areas of the loop node, we will confirm it as the root node of the option parsing code.

❷ **Select** ["*switch-case*", "*if-strcmp*"], which can be seen as sequential or nested comparisons, is the way to determine option items provided by most of the programs. Considering the real-world implementations, there can be two types: *switch-case*-based selection and *if-strcmp*-based selection. While small programs usually use an *if-strcmp* structure to select different options, large-scale programs prefer the *switch-case* structure as its concise form. Note that in Figure 3, there are two kinds of AST nodes, i.e., the C_S and JUMP_TARGET (**J_T**), combined into a complete selection. Then, we can use two simple filter traversals, i.e., $T_{V_A(code:\text{"}C_S, switch\text{"})}$ and $T_{V_A(code:\text{"}J_T, case\text{"})}$, to confirm the existence of the selection.

❸ **Target** [*argv*]: If the programs directly use *argv* as the parameter, e.g., *switch(argv)* or *strcmp(argv, "...")*, we can directly extract the corresponding option node, which is compared with the target, e.g., the shadow node (**LITERAL**, "a") in Figure 3. When the target is not *argv*, we can still adopt a backward analysis in the G_P to confirm it. In our example, the variable `c` is used to select user-providing options (Line 14), identified by the *switch-case*-comparison in G_A. Next, we can use the data-dependency of the same variable in the same location (i.e., line number) in G_P to trace backward and find that it is the return value of `getopt()`, which uses the *argv* as its input parameter. After we confirm the existence of option parsing, we can extract the 'a' as one valid option according to the `Select` object. Finally, we will repeat the above steps to find all the options and add them as our option nodes V_{OPT} into the option impact graph.

4.2.2 ***DIV Node Extraction***. After extracting the option nodes, we should recognize the corresponding DIV nodes, i.e., V_{OPT}, and add them into option impact graph. *First*, we need to figure out each option's *control scope*, i.e., a subset in V_A impacted by a specific option during option parsing. Specifically, for the *if-strcmp*-comparison, we recognize each subset under the corresponding *if*-control block(s) using traversals like $T_{Succ} \circ T_{V_A(code:\text{"}C_S, if\text{"})}$. For the *switch-case*-comparison, we recognize the subsets between *case* and *break* nodes. using traversals like $(T_{Succ} \circ T_{V_A(code:\text{"}C_S, case\text{"})}) \cap (T_{Pre} \circ T_{V_A(code:\text{"}C_S, break\text{"})})$. *Second*, as DIVs are updated under the control of each option, in this work, we mainly focus on two kinds of nodes: the destination operands of *assignment* statements, e.g., the blue node (**IDENT**, "flag") under the ASSIGN node in Figure 3 and the *returned* values or *out* parameters of *function* statements, e.g., the blue node (**IDENT**, "name") under the FUNC node in Figure 3.

4.2.3 ***Option Type Inference***. To assign the OG properties for the option and DIV nodes, we need to infer the option types based on how their DIVs are created. Here, we only consider three common option types, *numeric*, *string*, and ε, a little different from a recent OAF [52, 53], which relies on manual analysis of official documents. In this work, we try to automatically infer types according to the assignment behaviors. First, we will check whether there is an option value used to create the DIVs, and if no, we say the option type is ε. Second, if the option value appears, we will infer the types based on the specific API usage. For example, when programmers use `atoi(optarg)` or `strtol(optarg)` to set one of its DIVs, e.g., Line 23 in Figure 1, then we can say the option has a *numeric type* and assign the option group attribute ["$o_i : N$"] for the option node and all its DIV nodes. Similarly, when one of the DIVs is updated through the function `strncpy`, we can reasonably infer its type as a *string* and then attach ["$o_i : S$"] with them. Third, if we cannot find any specific API usage, we just say its type is string.

4.3 Impact Analysis

After extracting options and DIVs from AST, we begin the impact analysis, i.e., propagating the DIVs in the PDGs, extracting impacted nodes, and adding the nodes into option impact graph.

To this end, we can use the impact traversal (defined in §3.2) on the G_P starting from the DIV nodes to find out the data- and control-dependent nodes. However, considering the limitations of the traditional PDG [17], we need to solve the following questions to complete the inter-procedural analysis: **(1)** locating the DIVs in PDG to start impact analysis, **(2)** expanding the function callsites to support inter-procedural analysis, **(3)** decomposing conditions to solve potential false positives, **(4)** tracking global or structure variables to solve potential false negatives.

To overcome the above problems, we have proposed the following ***Rules*** to facilitate the impact analysis.

R1: DIVs Locating. A prerequisite of the option impact analysis is to recognize the DIVs in PDG corresponding to the ones extracted from AST (§4.2.2). In this work, we complete it simply by the *line number*, a common property used both in G_A and G_P (see the definitions in §3.1). By using the filter traversals, like $T_{V_P(line_number:N)}$, we can extract the nodes V_P^{DIV} containing DIVs in G_P. Then, we can use the impact traversal, like $T_I(V_P^{DIV})$, to start the impact analysis.

R2: CallSite Expanding. To support inter-procedural analysis, when meeting a function callsite, say F, we will try to expand it, i.e., connecting the current node with the nodes *copied* from the G_P^F. Specifically, we expand the connection process via the following kinds of nodes from G_P^F:

- **Method Node**: We will first copy this node from G_P^F and connect the callsite node with it using a control-dependency edge if there is any control dependency propagated to the callsite, e.g., the left red control-dependency edge in Figure 4 (a).
- **Parameter Node**: For any parameter in the callsite statement, if it can be reached through the impact traversal, we will copy its corresponding parameter node from G_P^F and add the proper data-dependency edge based on the ones propagated to the callsite, e.g., the middle blue edges in Figure 4 (a).
- **Method_Return Node**: We will copy this kind of node from G_P^F if a return value is held in the callsite. Then, we will connect it back to the callsite with a *RET* data-dependency edge, e.g., the right middle blue edge in Figure 4 (a).
- **Statement Node**: If any of the above nodes are copied to expand the callsite, we will start to traverse the nodes in the G_P^F and extract data- and control-dependent nodes and edges. If the method_return node is copied, we will try to copy all the reachable return-statement nodes and connect them back to the method_return node using *RET* data-dependency edges, which can ensure correctness even when meeting multiple return statements in G_P^F.

Theoretically, we should expand all the callsites for each user-defined function during our impact analysis. Specifically, for each callsite of Function F, we use a bit vector, say BV_F, whose length is *2+#P*, and the *#P* is the number of the parameters. For each function callsite, we set the corresponding bit if the method node, any of the parameter nodes, or the method_return node is copied. Then, we will not expand the function again if its bit vector has been met. Finally, we treat the function-pointer callsites and the third-party-library callsites (e.g., `printf()`) as normal statements, i.e., only considering if there are two or more options that can impact these callsites. If there is any *out* parameter (by reading the third-party-library functions' declarations), e.g., `snprintf()`, OG properties are propagated from input parameters to *out* parameters.

R3: Condition Decomposing. When meeting condition statements containing DIVs or IIVs, we should decompose them to determine the minimum condition that satisfies the judgment. For ones connected by *and*, they should be tracked together, as shown in Figure 4 (b). The impacts carried by these IIVs will be combined and passed on to the nodes controlled by the statement. However, if connected by *or*, they only need to be met on one side and we track

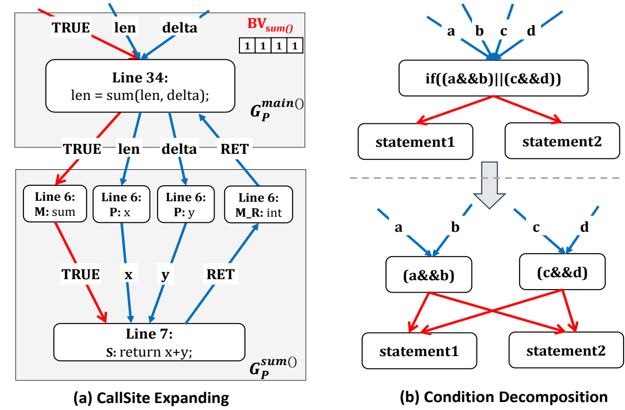

Figure 4: Illustration of Rules for Callsite Expanding and Condition Decomposing.

them independently. The impacts will be separated according to the two sides of *or* statement and passed on. To solve this problem, we decompose the conditions based on the code structure presented in G_A. Specifically, whenever we meet a condition statement in the PDG, we will find and parse the corresponding statement in AST by the line number. Firstly, we collect all of the children nodes, which are sub-trees rooted at the condition statement node. Secondly, we begin our analysis from the leaf nodes containing the variables and recursively merge them based on the *or* and *and* connectors until we meet the root node. Finally, we pass the parsing results, containing the groups of variables, back into PDGs for further analysis.

R4: Global Variable Tracking. In the traditional PDGs, global variables have not been tracked, which will lead to many false negatives, as they can also be global DIVs or IIVs. To solve this problem, we identify all global variables by a public API provided by LLVM [14], i.e., `Module->getGlobalList()`, and store them in a global mapping dictionary, ***GMD***, whose keys are the global variable names and values are the related OGs. During the analysis, we check each node and confirm if it contains any global variable whose name is recorded in the ***GMD***. If the global variable is used as the source value, we get its OG property from ***GMD*** and propagate it. If the global variable is used as the destination value, we will inherit the OG properties from their parent nodes and store them in ***GMD***. Since global variables may be modified with other operands, and the accurate execution order can hardly be statically determined, we propose an optimized rule to reduce the potential false negatives. Specifically, when global variables are assigned by other variables, we do not modify the global dictionary, and we only propagate the modifications inside the function, which will cause some acceptable false positives.

R5: Structure Variable Tracking. To track the DIVs/IIVs hidden in some structure variables, i.e., their *members*, we first record these variables as a special kind of IIVs, i.e., S-IIVs. Then, we track them along with other influenced variables but with field-sensitive analysis. Specifically, we check each access point for these S-IIVs and confirm if the proper *members* are used as source operands; if yes, we will then track the destination operands as new IIVs.

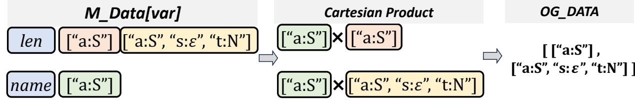

Figure 5: Grouping By Data-dependency. (For Line 37 in Figure 1.)

4.4 Option Grouping

With the above two steps, we obtain an initialized option impact graph G_O^{init}, which contains all the impacted nodes but without *OG* properties. In this step, we propose a method to aggregate the option impact and generate option combinations. The option impacts of the node are propagated through the data and control dependencies. Therefore, we combine the two parts of options. In addition, source operands might be assigned through different paths because of static analysis. Taking path sensitivity into account, we leverage the Cartesian product to put different data-dependent options obtained by a single operand into different option combinations.

4.4.1 ***Grouping Algorithm***. Algorithm 1 gives the details of our grouping method, and its input is the G_O^{init}, and the output is a complete option impact graph G_O. Generally, the algorithm consists of two main steps:

Initialization Stage. In Lines 1-9, we first initialize the G_O with G_O^{init} and then begin to assign the *OG* property to its option nodes and DIV nodes. Specifically, as introduced in §4.2.3, we will infer the type of each option by analyzing the assignment behaviors of all its DIVs, and then construct the *OG* for each option node and its DIV nodes. Finally, in Line 10, we construct a sequence queue, `Q_Node`, storing the nodes in topological order [26]. Note that accessing the nodes in topological order can guarantee that all of the parents of the currently visited node already have the *OG*.

Main-Loop Stage. In Lines 11-27, we present the main body of our algorithm, a large loop-based process that can be split into three *sub-stages* (SS). Note that the *termination* condition of this loop is that the `Q_Node` becomes empty, i.e., having attached the *OG* for all nodes.

- **SS-1: Grouping By Data-dependency.** In Lines 12-21, after fetching the *currently visited* node, i.e., v_{cur}, from the `Q_Node`, if it has no *OG*, we begin the following calculation. First, we prepare an empty dictionary `M_Data` to temporally store all the *OG* values from the parents who have data-dependency on the current node. Specifically, by iterating the parent nodes through the traversal T_{Pre}^{D} (Line 17), we fetch the proper variable `var`, by accessing the *symbol* property of the related edges (Line 18), and use it as a key of the `M_Data` to store the related *OG* inherited from the parent nodes (Line 19), as shown in the *left* of Figure 5. Second, as one node can be impacted by multiple variables, and single variable may have multiple groups, in Line 21, we operate a Cartesian product on `M_Data` to get all potential data-dependency *OG*. We depict this step and the results in the *middle* and *right* of Figure 5.
- **SS-2: Grouping By Control-dependency.** While we also use the `foreach` to iterate all the parent nodes (Line 23), which v_{cur} has control-dependency on, we should know that there is no more than one control-dependency edge based on the formal definition of PDG. Consequently, if there is indeed a control-dependency from its parent, we just use the `OG_CTRL` to inherit the parent's OG.
- **SS-3: Group Generation and Propagation.** In Line 26, we get the option groups for the currently visited node through the second Cartesian product on the `OG_DATA` and `OG_CTRL`. To facilitate the calculation, we add the corresponding two labels, i.e., '`DATA`' and '`CTRL`', for the two kinds of groups. The details of the second Cartesian-product are similar to the first one, which is already depicted in Figure 5.

Algorithm 1: Grouping

Input: G_O^{init}: *an initialized OIG, only containing nodes without OG*
Output: G_O: *a complete OIG, all nodes have OG*

```
1  G_O = G_O^init
2  foreach v_OPT ∈ getOPTNode(G_O) do
3      DIV_Set = getDIVNodes(v_OPT)
4      t = getOPTTypeByDIVType(DIV_Set)
5      v_OPT.OG = [getOPTName(v_OPT): t]
6      foreach v_DIV ∈ DIV_Set do
7          v_DIV.OG = [getOPTName(v_OPT): t]
8      end
9  end
10 Q_Node = getNodesByTopoOrder(G_O)
11 while !Q_Node.empty() do
12     v_cur = Q_Node.pop()
13     if hasOG(v_cur) then
14         continue
15     end
16     M_Data = {}
17     foreach v_parent ∈ T^D_Pre({v_cur}) do
18         var = e_(v_parent,v_cur).symbol
19         M_Data[var].append(v_parent.OG)
20     end
21     OG_DATA = getCP(M_Data)
22     OG_CTRL = []
23     foreach v_parent ∈ T^C_Pre({v_cur}) do
24         OG_CTRL.append(v_parent.OG)
25     end
26     v_cur.OG=getCP({'DATA':OG_DATA, 'CTRL':OG_CTRL})
27 end
```

4.4.2 ***Conflict Identification***. Based on our observations, which are also described in recent works [49, 52, 53], there are many mutually exclusive, i.e., *conflict* relationships between different options. Unfortunately, our current grouping algorithm cannot automatically identify such potential false positives. To solve this problem, besides manual identification based on the documents, we also propose a heuristic post-process based on a common coding behavior, i.e., if the influenced variables of two or more options converge into a control structure node under which there is an *exit* statement, we treat all the involved options as conflict ones, whose relationships will be removed from our grouping results.

4.5 Fuzzing Strategies

Basically, there are two kinds of input data that can trigger security bugs, i.e., normal input and program options. Unlike mutation strategies adopted by existing option-aware fuzzers, i.e., mutating the arguments and options together, or only mutating the arguments

with fixed options, we mutate the input and option separately. Accordingly, after extracting all options and potential combinations, we use two fuzzing strategies, *Option-Mutation*(OM) and *Input-Mutation*(IM), to detect the security bugs. For the OM strategy, for each option of the inferred combinations, we will fix the input and mutate the option values according to the option type (string, numeric, or ε). Specifically, for string and numeric options, the initial content is randomly increased, decreased, or replaced. For ε options, we randomly enable (keep) or disable (remove) them. For the IM strategy, we will fix each inferred option combination with initial option values, e.g., "abc" for string options and "123" for numeric options. We execute dry runs for each combination with the same seed to exclude combinations that will exit early and then mutate the input for each fuzzing round like AFL++.

5 Implementation

We implement OSmart based on the open-sourced JOERN [25], which can generate the AST-transformed and PDG-transformed property graphs from the source code. OSmart contains about 4.3K lines of Python code, including about 1.6K lines of code to implement option extraction and about 2.7K to analyze option impact and generate option impact graph. We also use the latest version of AFL++ [1]. Now, we will introduce several implementation details of OSmart.

Topological Sort. We use `getNodesByTopoOrder()` at Line 10 in Algorithm 1 to get all nodes in the topological order. When calculating the OGs of each node, we need to inherit the OGs from its parent nodes. Therefore, we should arrange the node traversal order and guarantee that all of its parents have been visited and attached to the OG property. While the topological sort is ideal, it only works on a DAG(Directed Acyclic Graph), which is unsuitable for real-world programs because of potential loops. In this work, we use the *line number* to break any potential loop, making each line appear only once, achieving an approximate solution.

OM Fuzzing Strategy. While the IM strategy can be easily implemented based on traditional fuzzing, we should do some more work to implement the OM strategy. Compared with input-mutation, which relies on mutating the content of an input file, option-mutation actually needs to mutate the option value directly. To this end, we slightly transform the option parsing code, i.e., fetching option values from outside files, which are then mutated by existing fuzzing works.

6 Evaluation

We mainly evaluate OSmart on the following aspects:

- **Option Extraction:** Can OSmart efficiently identify documented and undocumented options? How about false positives and false negatives?
- **Option Grouping:** How many groups can be automatically identified by OSmart? Can OSmart infer option groups that cannot be easily identified from documents? How do option groups found by OSmart perform compared with those of CarpetFuzz [49]? Are there any wrong (misleading) group descriptions in official documents?
- **Fuzzing Efficiency:** Can OSmart improve the performance of current fuzzing tools? How many unreachable codes can be found by the option group with an undocumented option? What about the comparison results with existing option-aware fuzzing work?
- **Bug Detection:** Can OSmart discover security bugs triggered through proper option combinations? How about the effectiveness of the two fuzzing strategies? What can we learn from these bugs?

6.1 Experiment Setup

Benchmark. We collected 56 programs, with the latest version of their source code, from 25 popular large-scale packages (e.g., *QEMU*, *OpenSSH*, *PHP*) and third-party libraries (e.g.,*libtiff*, *libxml2*, *libpng*) as our dataset. To keep the fairness of our selection, we tried to cover as many types of option-involved programs as we could. Our dataset contained five types: machine emulators, network servers/tools, script engines, databases, and common tool collections/libraries.

Environment. We evaluated OSmart on a 96-core machine running Ubuntu 20.04.1 LTS with four Intel Xeon Platinum 8268 CPUs and 409 GB memory. We installed JOERN 1.1.1066 [25] and converted the source code to AST-transformed and PDG-transformed property graphs stored in dotfiles [24]. To fetch the global variables, we used LLVM 13.0 [14].

6.2 Option Extraction

6.2.1 *Extraction Results*.
From the *Option Extraction* main column of Table 2, we could see that OSmart unexpectedly found *193* undocumented options not presented in the *help* messages. Interestingly, we discovered undocumented options in popular large-scale programs (e.g., the latest version of *QEMU*, *PHP*) and popular third-party libraries (e.g., *libxml2*, *binutils*). In total, more than 67.9% of programs did not provide all valid options in their *help* messages.

In addition, as shown in Table 2, OSmart extracted 4,924 DIVs from the option parsing procedure. These DIVs were then used as source variables to propagate the option impacts. Besides, we inferred the types of DIVs according to assignment modes. 161 DIVs were numeric, 2,398 were string, and the rest were ε. We used these DIV types to infer 161 *numeric*, 460 *string*, and 1038 ε options.

Finally, based on our manual confirmation, we found that developers mainly chose *while-switch-case-optarg* (WSCO), *for-if-strcmp-argv[i]*(FISA), and their variants to parse options. We found most (83.9%) of developers preferred to *while-switch-case-getopt*. The main reason was that the *switch-case* could make the parsing code more concise and maintainable than the *if-strcmp*.

6.2.2 *A Real Undocumented Option*.
Listing 1 presents a code snippet of the `xmlcatalog` program, which contains an undocumented option, "–convert", identified by OSmart. Specifically, based on the *if-strcmp* rule, our tool identified the "–convert" option and the related DIV *convert*, which was then used as a condition in Line 6. However, this option was not listed in the *help* messages. Consequently, this undocumented option led to the unreachable code, Line 7, which contained a callsite of an important function, `xmlCatalogConvert()`. For existing option-aware fuzzers, like POWER [28] and CarpetFuzz [49], this undocumented option made them miss the chance to examine this function.

Table 2: OSmart Analysis Results. ***(W/F)SC(O/A) = (While/For)-Switch-Case-(Optarg/Argv[i]). (W/F)I(K/S)A = (While/For)-If-(Keymatch/Strcmp)-Argv[i].*** **Doc. = Documented. Com. = Complete (✓) or not (✗). #Un = Number of undocumented options. #Dep/#Con/#CG/#OG = Number of *dep*endency relationships/*con*flict relationships/confilct option groups/final option groups. C-Error=Compilation error. M-Error = No man page found. O-Error = No option can be extracted. Δ= Unique line coverage. ΔUn= Unique line coverage of undocumented options.**

Package	Program	Option Extraction						Option Grouping			Fuzzing			
		Pattern	Doc.	Com.	#Un	DIV	Type Num / Str	Doc.	Carpet #Dep+#Con/#OG	OSmart #OG+#CG	AFL++	ΔCarpet	ΔOSmart	ΔUn
Libtiff 4.5.0	fax2ps	*WSCO*	8	✓	-	12	6/0	2	0+0/64	14+0	5145	22	315	-
	fax2tiff	*WSCO*	29	✗	2	37	3/4	0	0+9/812	49+1	1567	198	501	6
	tiff2ps	*WSCO*	31	✓	-	41	12/5	6	0+8/841	187+159	9750	94	989	-
	tiff2rgba	*WSCO*	7	✗	1	13	3/1	0	0+0/-	20+2	7574	-	1103	450
	tiffcp	*WSCO*	21	✓	-	105	5/55	0	0+2/405	40+11	454	219	749	-
	tiffcrop	*WSCO*	34	✗	4	175	12/15	3	1+3/693	83+21	12075	162	555	564
	tiffinfo	*WSCO*	14	✗	1	17	3/2	0	0+0/116	33+0	5494	33	340	127
Binutils 2.39	as	*WSCO*	81	✗	10	674	5/480	0	C-Error	368+34	13669	-	3552	1253
	gdb	*WSCO*	34	✗	6	30	3/4	1		40+1	x*			
	gprof	*WSCO*	42	✗	1	119	4/45	3		325+79	5343	-	455	331
	objdump	*WSCO*	52	✓	-	194	4/47	12		611+42	20037	-	6676	-
	readelf	*WSCO*	47	✗	2	137	3/16	0		108+20	9790	-	8176	2411
	size	*WSCO*	11	✗	1	20	1/3	0		13+2	4334	-	410	137
Qemu 7.2.0	qemu-edid	*WSCO*	10	✓	-	14	5/9	0	M-Error	14+0	450	-	51	-
	qemu-img bench	*WSCO*	17	✗	1	41	7/18	0		41+5	1019	-	7991	374
	qemu-img convert	*WSCO*	25	✗	1	52	3/33	0		57+6	2361	-	8444	231
	qemu-img dd	*WSCO*	5	✗	1	5	0/3	0		12+3	1040	-	4653	513
	qemu-img measure	*WSCO*	9	✗	1	33	1/26	0		20+3	1035	-	6690	111
	qemu-nbd	*WSCO*	22	✗	1	87	2/59	2		351+74	6820	-	3571	1494
Mupdf 1.21.1	mupdf-x11	*WSCO*	10	✗	1	17	6/7	0	0+0/116	57+16	1037	0	153	115
	muraster	*WSCO*	19	✗	2	42	9/23	0	M-Error	209+91	27487	-	5566	958
	mutool clean	*WSCO*	17	✓	-	19	1/6	0	O-Error	278+51	1412	-	19392	-
	mutool convert	*WSCO*	10	✓	-	13	4/8	0	O-Error	23+6	54	-	5757	-
Openssh v_9_1_P1	scp	*WSCO*	23	✗	3	22	1/4	0	O-Error	237+97	3369	-	160	0
	sftp	*WSCO*	24	✓	-	25	3/8	0		30+9	3529	-	19	-
	ssh	*WSCO*	44	✓	-	140	0/76	1		119+18	3548	-	970	-
	ssh-agent	*WSCO*	9	✗	1	10	0/4	0		17+8	4518	-	18	7
Jasper 4.0.0-rc1	imgcmp	*WSCO*	5	✗	5	12	2/5	0	O-Error	22+3	631	-	1346	42
	imginfo	*WSCO*	8	✗	6	17	3/4	0		16+1	3859	-	409	102
	jasper	*WSCO*	11	✗	6	23	3/8	0		30+16	4047	-	243	66
Httpd 2.4.54	ab	*WSCO*	35	✓	-	85	7/18	2	0+2/592	117+49	114	153	110	-
	htdbm	*WSCO*	15	✓	-	17	0/0	0	1+1/313	17+5	123	107	45	-
Libarchive 3.6.2	bsdcpio	*WSCO*	10	✗	32	54	1/1	0	O-Error	69+34	6936	-	771	2
	bsdtar	*WSCO*	19	✗	66	132	4/1	0		108+51	7711	-	3720	633
Libjpeg 2.1.4	cjpeg	*FIKA*	23	✓	-	34	0/6	1	0+4/583	21+2	3459	79	296	-
	tjbench	*FISA*	35	✗	3	61	4/2	7	0+21/783	328+0	655	140	6089	189
Libxml2 2.10.3	xmlcatalog	*FISA*	8	✗	1	13	0/0	3	0+0/116	39+0	90	110	47	2
	xmllint	*FISA*	64	✗	2	108	2/50	1	0+1/698	102+2	193	883	572	43
Nasm 2.16.01	nasm	*WSCA*	40	✗	4	97	1/21	0	0+6/737	30+2	7220	294	275	14
	ndisasm	*WSCA*	11	✓	-	125	1/88	0	0+0/169	14+2	1127	145	179	-
PHP 8.2.0	php	*WSCO*	28	✗	2	44	1/11	0	O-Error	23+0	25477	-	1427	0
	phpdbg	*WSCO*	21	✗	1	23	0/7	0	0+1/342	89+0	x*			
S2n-tls 1.3.31	s2nc	*WSCO*	30	✓	-	45	4/19	1	M-Error	53+11	904	-	25	-
	s2nd	*WSCO*	28	✗	1	37	4/14	1		42+8	106	-	37	13
Catdoc 0.95	catdoc	*WSCO*	14	✓	-	14	1/4	0	0+1/283	17+0	380	72	482	-
Gifsicle 1.93	gifsicle	*WSCO*	57	✗	4	1088	0/771	0	1+7/-	206+3	2335	-	3090	240
Jhead 3.06.0.1	jhead	*FISA*	45	✗	1	109	0/10	0	0+1/467	1669+434	1028	78	592	35
Libdwarf 0.5.0	dwarfdump	*WSCO*	102	✗	2	159	5/128	5	0+0/1	5894+231	15261	0	834	179
Libming 0.4.8	makeswf	*WSCO*	14	✗	1	53	0/40	0	0+0/240	33+0	2396	567	411	732
Libpng 1.6.39	pngfix	*WISA*	9	✗	2	24	1/3	1	M-Error	10+1	1016	-	162	126
Libsixel 1.8.6	img2sixel	*FSCO*	33	✗	1	116	3/0	6	5+1/678	31+0	1289	311	2314	352
Nginx 1.23.3	nginx	*WSCA*	11	✓	-	26	0/5	0	O-Error	19+2	3235	-	4101	-
Sngrep 1.6.0	sngrep	*WSCO*	17	✓	-	57	2/45	0	0+0/281	21+0	1127	68	114	-
Sqlite3 3.40.1	sqlite3	*FISA*	43	✗	5	38	1/11	1	0+1/381	40+1	25990	0	974	975
W3m 0.5.3	w3m	*WISA*	44	✗	8	67	5/7	1	0+1/633	74+1	4737	0	44	6
Yasm 1.3.0	yasm	*FISA*	31	✓	-	158	0/158	1	C-Error	58+1	13791	-	116	-
Total			**1466**	✗ **193**		**4924**	**161/2398**	**61**	**8+70/10344**	**12560+1531**	**288146**	**3735**	**116081**	**12833**

x*: Gdb and phpdbg are interactive programs, difficult to calculate line coverage.

```
1 // option parse
2 if ((!strcmp(argv[i], "-convert")) || (!strcmp(argv[i], "
      --convert"))) {
3     convert++;
4 }
5 // use the undocumented option
6 if (convert)
7         ret = xmlCatalogConvert();
```

Listing 1: An Undocumented Option in `xmlcatalog`.

6.2.3 ***FP and FN.*** We manually analyzed the program code, found out the options defined, and recognized the type of each option in the program as a benchmark. For option node extraction, we compared artificial results with the options extracted by the LST-rules. The *accuracy*, *precision*, and *recall* of our rule on the benchmark were 80.3%, 82.3%, and 97.0%, respectively. There were two main false positives during our evaluation: *interactive commands* and *sub-program commands*. Specifically, the interactive programs, e.g., *gdb*, received many user commands during debugging or running. Besides, various sub-program commands were commonly used to construct larger programs, e.g., *qemu-img* and *mutool*. In such cases, OSmart would incorrectly recognize them as options. The main reason for *false negatives* is that some programs try to parse partial options in a latent way. For example, when an `option` object is passed into the `getopt_long()` function and its `*flag` variable is not empty [19], the function will directly complete the option parsing task, i.e., `*flag` points to `val`. In such cases, OSmart cannot extract the options.

For option type inference, we confirmed that there were totally 82 false positives. Based on our manual analysis, there were two main reasons that could lead to the false positives. Firstly, the programs chose custom-defined functions to wrap the official APIs, e.g., using *qemu_strtoui* for *aoti()* in *qemu*, which were beyond our rules. Secondly, the programs chose custom-defined variables for the standard ones, like *optarg* and *argv[i]*, to carry the option values in the parsing code, which could also lead to incorrect type inference as we thought there was no option value assigned and treated their types as ε.

6.3 Option Grouping

6.3.1 ***Combinations From OSmart.*** As shown in the second main column of Table 2, compared with only 61 relationships declared in the official documents, OSmart inferred 12,560 option groups in 56 programs. In other words, by using OSmart, we could obtain more than 205× relationships. In addition, OSmart inferred 1669 and 5894 option groups for `jhead` and `dwarfdump` because of the large number of options and frequent usages for their DIVs. By adopting conflict identification, OSmart also identified 1,531 conflict relationships. For example, for `tiffcrop`, whenever providing "-H" and "-W" at the same, it would promote an error message, i.e., "only one of -H or -W to define a viewport", which could help us to identify the conflict, and we would only keep one of them. Besides, when providing "-r auto" with one of the above two options, it would say, "*-r auto* is incompatible with maximum page width/height specified by -H or -W", meaning that the "-r" was mutually exclusive with the option "-H" or "-W" if it took the initialized value *"auto"*.

```
338.  case 'c':
339.        cipher_prefs = optarg;
......
393.  case 'a':
394.        session_ticket_key_file_path = optarg;
......
443.  case 'E':
444.        max_early_data = atoi(optarg);
......
591.  s2n_set_common_server_config(max_early_data, config,
      conn_settings, cipher_prefs,  session_ticket_key_file_path);
```

Figure 6: Grouping Details in `s2nd.c`.

Moreover, we deeply analyzed the dependency type of option groups by OSmart, among which there were 7,505 option groups that could be obtained by control dependency, 6,858 ones by data dependency, and 1,1341 ones by combination of control with data dependency. There was a crossover between the three types. We found the combination of control and data dependency obtained the most option groups. When there were more control structures in the program, and DIVs or IIVs acted as judgment conditions, the number of option groups obtained by control dependency and control with data dependency would increase.

6.3.2 ***Combinations From Documentation.*** In total, there were 12,150 option groups that OSmart could find but could not be directly inferred from the documentation descriptions. For example, for program `s2nd`, the option "-c" meant *"Set the cipher preference version string"*, the option "-E" meant *"Sets maximum early data allowed in session tickets"*, and the option "-a" meant *"Location of key file used for encryption and decryption of session ticket"*. Apparently, we could not conclude the relationship between these options according to the description in the documentation. However, we found the data dependencies between the DIVs of these options, and these DIVs were used as parameters of the same function `s2n_set_common_server_config` in s2nd.c:591, as shown in Figure 6.

Besides, in order to clarify whether OSmart could cover the option groups mentioned in the help documents, we manually identified all the relationships in the documents. We manually found 61 option groups from the documents, as shown in the `Doc.` subcolumn of the second main column of Table 2. By comparing the results of OSmart with the option combinations in the documentation, OSmart missed 21 relationships in the documentation. We analyzed these relationships in depth and found that OSmart missed 11 relationships in 6 programs due to dependencies not being found in the program, missed 8 in 2 programs due to the absence of function pointers analysis, missed one due to complex option parsing and direct DIV assignment not being found, and missed one due to macro definitions leading Joern failed to generate the graphs.

6.3.3 ***Combinations From CarpetFuzz.*** We also compared the grouping results from CarpetFuzz [49] which combined the options through NLP techniques to extract dependency and conflict relationships from *man* pages. In particular, we applied it to all 56 programs. However, it could only work on 17 programs, extracting 8 dependencies and 70 conflict relationships, which produced 10344

combinations. For the remaining 30 programs, the CarpetFuzz failed due to *compile error* (C-Error), *man page not found error* (M-Error), or *no option found error* (O-Error). There were 9 programs that had no relationships in the result, but they could still be utilized due to the N-wise combination strategy adopted by CarpetFuzz. In addition, `tiff2rgba` and `gif2sicle` could not combine options properly. Totally, CarpetFuzz could not apply to 32/56 of our dataset, meaning that it could not be widely used.

We manually analyzed the 8 dependencies extracted by CarpetFuzz. OSmart could find 4 of them and missed 3 relationships in *img2sixel* due to function pointers mentioned in 6.3.2. For the "-b" and "-t" in *htdbm*, we could not figure out any relationship between the two options from document or code. Besides, we also used CarpetFuzz to generate option groups for these programs, which contained the programs that could not be extracted any relationships. Finally, with the extracted relationships, including the dependencies and conflicts, CarpetFuzz totally generated 10344 option combinations.

6.3.4 ***Wrong Relationship Descriptions.*** During our evaluation, it was unexpected that a wrong description existed in the `tiffcrop`'s official document. Based on the *help* explanation, the option "-J" was used to *"Set ... when ... using the -S cols:rows option"*, which implied the group relationship of the two options, "-J" and "-S". However, OSmart found that there was an *exit* when both of them were provided, meaning that this group would be eliminated by conflict identification. In fact, based on our manual review, the function enabled by these two options was an ERROR function ("TIFFError"), so the program would be terminated if the two options were provided. As a result, this kind of wrong description would mislead most of the current option-aware fuzzing works, e.g., the works [49, 52, 53] that relied on official documents.

6.4 Fuzzing Efficiency

6.4.1 ***Improvement Of OSmart.*** To evaluate the improvement of OSmart, We ran each fuzzer on one CPU core for 48 hours with the IM strategy separately and repeated it 5 times to avoid the uncertain impact on all evaluations. Since `gdb` and `phpdbg` were interactive programs, it was difficult to calculate their line coverage.
OSmart Vs. AFL++: In conducting the fuzzing experiments with `AFL++`, we used the least options that would make the program run to start AFL++ and used the option groups obtained for OSmart to run OSmart. In Table 2, `AFL++` represents the line coverage with AFL++, and ΔOSmart represents the unique line coverage obtained by OSmart more than `AFL++`. According to the results for ΔOSmart in Table 2, OSmart could lead to new coverage gains. In total, `AFL++` got 288,146, while OSmart found 116,081 more than `AFL++`, which was 40.3% of the total `AFL++`. Also, for individual programs, in terms of number, OSmart could find up to 19,392 more than `AFL++`, like `mutool clean`, which was 13.7× more than `AFL++`. In terms of rate, OSmart could find 106.6× more line coverage than `AFL++`, like `mutool convert`. Overall, the option groups provided by OSmart are very effective in improving program coverage.
Undocumented Option: To show the impacts of undocumented options, we selected option groups containing these options for fuzzing. ΔUn in Table 2 represents the unique line coverage increasing from option groups including undocumented options compared

Table 3: Option-Aware Fuzzing Comparison on POWER's Dataset.

Program	OSmart	CarpetFuzz	POWER	ConfigFuzz	AFL++-ARGV
avconv	**3518.15**	2442.59	2209.15*	2034.75	825.01
bison	**1269.46**	1133.14	1148.31*	813.15	154
cflow	604.87	565.8	**642.79***	150.42	311.47
cjpeg	**395.37**	118.07	73.46*	60.8	75.92
djpeg	**498.01**	136.37	52.97*	35.88	57.87
dwarfdump	**1771.19**	1746.02	1376.34*	406.57	162.71
exiv2	**2721.53**	1123.37	1402.15*	1202.13	514.36
ffmpeg	**5049.33**	2460.78	2768.37*	2134.4	611.72
gm	**631.38**	515	566.98*	468.18	342.06
gs	**9495.91**	9306.35*	9447.74*	*	3596.22
jasper	282.37	**849.54**	524.47*	479.08	42.34
mpg123	**846.69**	507.62	421.19*	742.34	122.82
mutool	2526.8	**3429.59**	48.33*	738.14	*
nasm	**1023.07**	911.89	898.18*	*	321.13
objdump	833.67	**1104.26**	1012.09*	437.06	67.87
pdftohtml	**3140.29**	1680.15	1536.17*	1159.44	306.86
pdftopng	**1655.06**	1627.17	1592.4*	1011.66	43.02
pdftops	1536.5	**1542.76**	1528.5*	1361.76	34.02
pngfix	192.94	200.99	210*	**300.51**	48.43
pspp	**2112.26**	1080.37*	1021.12*	1859.03	*
readelf	**524.25**	428.78	460.36*	354.48	26.89
size	423.3	310.82	**428.1***	234.04	39.37
tiff2pdf	**722.7**	717.63	661.9*	408.6	26.27
tiff2ps	**505.01**	414.55	376.28*	378.31	20.5
tiffinfo	**399.1**	388.3	331.87*	336.43	14.8
vim	4317.71	**4705.29**	4279.16*	4431.08	1322
xmlcatalog	**928.75**	803.89	748.26*	256.69	18.31
xmllint	855.63	1194.99	**1467.75***	1034.99	53.84
xmlwf	**286.44**	198.38	160.57*	35.78	118.67
yara	422.08	272.53	407.62*	**776.08**	199.34
Total	**49489.82**	41916.99	37802.58	23641.78	9477.82

The numbers mean the average numbers of coverage edges across all test cases [49]. We put the full data in https://github.com/osmart-source/osmartsource/tree/main/compare-tables. * means we failed to compile. N* means we reused the results [49] due to compilation errors.

with `AFL++`. In total, we obtained 12,833 line coverage from undocumented options for 36 programs. The number of line coverage was 4.45% of `AFL++` and 11.06% of the sum of ΔOSmart. And the most coverage could reach 2,411, equivalent to 24.63% of `AFL++`. The coverage achieved by undocumented options shows that these options, although not documented in the help documentation, can make the program execute unique paths and cover more code and discover potential bugs.

6.4.2 ***Option-Aware Fuzzing Comparisons.*** In order to evaluate whether white-box option dependency group could help the option-aware fuzzers have a better performance, we compared OSmart with the state-of-the-art option-aware fuzzer CarpetFuzz. We evaluated CarpetFuzz on OSmart's benchmark, and used the same programs and seeds with OSmart. To keep the fairness, each program was fuzzed by CarpetFuzz for 48 hours and repeated 5 times. The results can be seen in Table 2. **ΔCarpet** represents the unique code coverage of CarpetFuzz more than `AFL++`.

In terms of total line coverage, OSmart was 31.08 × more than CarpetFuzz. And there were several programs that CarpetFuzz could not apply to our dataset. If only comparing the programs that CarpetFuzz had coverage results, OSmart had 12355 new coverage and it was 3.3 *times* of CarpetFuzz. Although it generated more option groups, most of them could not increase code coverage. In total, for 17 programs, the coverage of OSmart was greater

than that of CarpetFuzz, and for the remaining 6 programs, they had similar coverage to that of CarpetFuzz. By manual analysis, we found that if options were not widely used, randomly picking options through N-wise testing used by CarpetFuzz could achieve better results.

6.4.3 ***Comparisons on POWER's dataset.*** We also compared OSmart with other state-of-the-art option fuzzers on the public benchmark released by POWER [28]. We reproduce CarpetFuzz [49] and ConfigFuzz [52, 53] based on the open-sourced code or implementation details described in their papers. For POWER [28], due to the lack of detailed information, we directly reused the results from the manuscript [49]. For AFL++-ARGV [22], the options and their parameters in the official documents were manually extracted as a dictionary to assist its execution. To keep fairness, we ran each fuzzer for 24 hours and repeated it 5 times. The results can be seen in Table 3.

Totally, for 20 of 30 (66.7%) programs, OSmart found new paths more than other option-aware fuzzers. Note that for the *cjpeg*, *djpeg*, *exiv2*, *ffmpeg*, and *pdftohtml*, OSmart could unveil twice the coverage compared to other fuzzers. For the other ten programs, OSmart did not perform as well as the other fuzzers. By manual analysis, we found that if options were not widely used together in the program, randomly choosing options could achieve better results. In addition, we also noticed that AFL++-ARGV did not outperform other fuzzers in all cases, demonstrating the inefficiency of option- and input-undifferentiated mutation.

6.5 Bug Detection and Lessons

6.5.1 ***Bug Details.*** OSmart discovered 51 new security bugs, including 35 highly exploitable bugs, i.e., 8 stack overflows, 14 heap overflows, 4 global overflows, and 9 UAFs. In particular, 33 new bugs were detected by the *input-mutation* strategy, including 25 high exploitable bugs, and 18 new bugs were detected by the *option-mutation* strategy, including 10 high exploitable ones. Besides, while 45 (88%) bugs could be triggered with no more than three options, there were still 6 bugs that could only be triggered with four or more options.

Bugs Detected By CarpetFuzz. To demonstrate the effectiveness of bug detection, we also used the CarpetFuzz to detect the bugs on our dataset. Generally, CarpetFuzz could only find 10 bugs in total, and 5 of them were the same as the ones detected by OSmart as they used the same option combinations to fuzz the same programs. Besides, we also compared the time of discovering the same five bugs found by both OSmart and CarpetFuzz. Totally, OSmart took 226.2s, and CarpetFuzz took 1117.6s. In addition, we also manually confirmed the reason for the other 5 bugs that could only be detected by CarpetFuzz. It turned out to be the failure of option extraction due to the false negatives described in §6.2.3.

Bugs Triggered By Undocumented Options. Unexpectedly, OSmart detected 4 security bugs (2 stack overflows and 1 UAF and 1 FPE) involved with undocumented options. CarpetFuzz did not unveil these bugs and these bugs could not be found by other option-aware fuzzing works that relied on the official documents [28, 49, 52, 53]. Among these bugs, two bugs were triggered by one undocumented option with another documented one, one bug was triggered by one undocumented option with three documented options, and one bug was triggered by four undocumented options.

6.5.2 ***Bug Lessons.*** Here we list all the new bugs details in Table 4, which have been introduced in §6.5. Moreover, by manually analyzing the root causes of these bugs, we present some valuable lessons learned from the option-related security bugs. And we will mainly describe three kinds of *Buffer-Overflow (BO)* bugs and one kind of *UAF* bug detected by our tool.

BO-1. These bugs usually exist in the code blocks that can only be reached through a specific option or option group. And they also need some long-enough input data to overflow the target buffers. For example, we detected such a buffer overflow within the `bsdcpio` program, which needed four options to satisfy all the path conditions to reach the vulnerable code. Then, when providing long-enough input data, a stack overflow would be triggered. It is noted that one of the four options is undocumented in the *help* messages.

BO-2. These bugs can be triggered by a specific option and a long-enough option values which will be finally filled into a small buffer. For example, our tool detected such a bug in the `makeswf` program, in which both of two options, "-I" and "-D", could append a 1023-byte string into the same 1024-character buffer. As a result, when the two options with two long-enough strings were provided in the program, there would be a global overflow.

BO-3. These bugs are very special as they can be triggered only by providing the same option many times without any extra data. For example, our tool detected such a bug in the `tiffcrop` program, in which a 10-character buffer index would be automatically increased whenever one of four specific options was provided. Consequently, when more than 10 such options were provided, a buffer overflow would occur. It is also worth noting that these four options are also undocumented in the *help* messages.

UAF-1. For all the UAF bugs found by OSmart, the main reason is the same, i.e., an extra *free* is invoked by a specific option or option group. For example, our tool detected such a bug in the `img2sixel` program, in which the "-B" option enabled a code block containing an extra *free* invocation. Then before the program existed, normal resource deallocation was invoked, and the freed object pointer would be used again. Besides, our tool also detected another similar bug in the `w3m` program, in which the "-halfload" option led to a *premature* free, and the program would then use the pointer again before it existed. It is noted that the above "-halfload" option is also undocumented in the *help* messages.

7 Discussion and Future Works

Symbolic Execution. Theoretically, symbolic execution [4, 9, 10, 12, 39, 40, 43] can also be used to extract program options for the simple parsing logic, which relies on the string comparison. However, when applied to real-world programs, especially for the ones parsing options by the *getopt()*, we were unable to obtain any result in hours. Nonetheless, we will try to combine the results from OSmart with the program analysis assisted fuzzing works [6, 41, 46, 51] in our future work.

Fine-grained Static Analysis. We have adopted a coarse-grained way to handle indirect functions, resulting in imprecise results to option grouping. A fine-grained approach to solving function pointers will be our future work, for example, integrating existing

function pointer identification methods, such as MLTA [32], to obtain the targets of function pointers for subsequent analysis. Besides, to properly initialize the value of options, we can extend OSMART with some static value analysis [47] in our future work.

Applicability of LST Rule. While OSMART can automatically extract program options on most of our evaluated programs, it still cannot handle the parsing across multiple functions or is processed implicitly, i.e., the exact parsing code is not presented in the program. As our future work, we will try to extend the rule to cover these code patterns and also apply it to other kinds of options, e.g., the ones contained in the configure files.

Conflict Relationship. We take heuristics based on code behavior to identify conflict option groups, but this causes false positives and the results are not minimized. Adding dynamic methods to strengthen the identification, such as parsing output messages or detecting early exits, is left as our future work.

Binary Program. Considering the difficulty in constructing ASTs and the lack of symbolic information, we cannot directly apply OSMART to binaries. Proposing heuristic rules to extract options and building an option impact graph from binary programs is also our main future work.

8 Related Work

8.1 Option-Aware Fuzzing

Fuzzing [3, 6, 11, 20, 21, 23, 29, 33, 37, 38, 41, 46], has been proven an effective method to discover security vulnerabilities. COVERSET [42] has pointed out that it is a possible direction to infer command line options and handle dependent arguments for fuzzing. CrFuzz [45] tries to enumerate option combinations, which can improve the efficiency of fuzzing. However, it only focuses on input mutation. POWER [28] designs a configuration relevance metric to select option combinations. However, it does not mutate the options. CarpetFuzz [49] adopts NLP techniques to automatically identify the dependency and conflict relationships between options in the help document and apply these relationships to fuzzing. However, it can not solve the undocumented option problem. ConfigFuzz [52, 53] picks options that are not mutually exclusive, specifies the type and range of option values, and mutates option values and program inputs. However, these works select option configurations from the official documentation, but we find that the official documentation has fewer options than the program defined. In addition, the works [2, 22] mutate options and input indiscriminately without the help of program documentation. However, it wastes much time in generating a valid option. OSMART can automatically extract options and infer groups, and supports two mutation strategies and missed options.

8.2 Bug Detection with Property Graph

There have been several works proposed to detect vulnerabilities with static graph-based analysis [7, 16, 30]. CPG [16] combines AST, PDG, and CFG to construct code property graphs. It proposes practical traversal methods on the graph and detects security vulnerabilities. However, it does not support inter-procedural analysis. Backes et al. [7] apply CPG to PHP programs and develop a tool named "phpjoern" [34]. ODGEN [30] extends the CPG with object-level data dependencies to detect Node.js bugs. CHKPLUG [44] combines different languages (JavaScript, HTML, PHP, and SQL) as a cross language code property graph structure, tracks the data flows and detects GDPR (General Data Protection Regulation) violations in WordPress plugins. CPGVA [50]utilizes the code property graph to model code lines as features and seeds them to deep learning [27] to discover source code vulnerabilities. Duan et al. [15] extract semantic features from the code property graph and train network using BiLSTM [54] and attention mechanism [48] and mine the vulnerabilities in the target source program. These works are based on graph indexing and traversals to find bugs directly from the graph. While the concept of OFG is inspired by CPG, OSMART uses inter-procedural analysis within the whole program's ASTs and PDGs and infers reasonable potential option groups, which are then used to improve program fuzzing.

9 Conclusion

We propose OSMART to automatically extract options and reasonably infer relationships within them. OSMART incorporates several practical techniques to build option impact graphs, which plays an important role in our whitebox approach. We prototype OSMART and evaluate it with 56 real-world programs. The experimental results show that OSMART is effective and practical in automatically extracting options and inferring option groups to improve program security.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful feedback. This research was supported, in part, by National Natural Science Foundation of China (Grant No. 62372437, 62232016, 62302508, 62302506), Research Project of Institute of Software Chinese Academy of Sciences (ISCAS-JCZD-202301, ISCAS-ZD-202402), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006), and the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. Y2021041).

References

[1] AFLplusplus/AFLplusplus. 2024. *AFLplusplus.* https://github.com/AFLplusplus/AFLplusplus.
[2] AFLplusplus/AFLplusplus. 2024. *AFLplusplus-argvfuzz.* https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/argv_fuzzing.
[3] american fuzzy lop (2.52b). 2024. *American Fuzzy Lop.* http://lcamtuf.coredump.cx/afl/.
[4] angr. 2015. *angr.* https://angr.io.
[5] Marko A.Rodriguez and Peter Neubauer. 2011. The Graph Traversal Pattern. In *Graph Data Management: Techniques and Applications.*
[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/
[7] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *IEEE European Symposium on Security and Privacy.*
[8] bjascob/LemmInflect. 2022. *Lemminflect.* https://github.com/bjascob/LemmInflect.
[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted And Automatic Generation Of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation.*
[10] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/

2408776.2408795

[11] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725. https://doi.org/10.1109/SP.2018.00046

[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform For In-vivo Multi-path Analysis Of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[13] Wikipedia Contributors. 2024. *getopt*. https://en.wikipedia.org/wiki/Getopt.

[14] LLVM Developers. 2024. *The LLVM Compiler Infrastructure*. https://www.llvm.org.

[15] Xu Duan, Jingzhen Wu, tianyue Wu, Mutian Yang, and Yanjun Wu. 2020. Vulnerability mining method based on code property graph and attention BiLSTM. *Journal of Software* 31, 11 (2020), 3404–3420.

[16] Yamaguchi Fabian, Golde Nico, Arp Daniel, and Rieck Konrad. 2014. Modeling and Discovering Vulnerabilities with Code Property Graph.. In *Proceedings IEEE Symposium on Security and Privacy*.

[17] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[18] AI2 Allen Institute for AI. 2024. *AllenNLP*. https://allenai.org/allennlp.

[19] gnu c library. 2024. *getopt_long*. https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y.Levin. 2008. Grammar-based Whitebox Fuzzing.. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*.

[21] Patrice Godefroid, Michael Y.Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing.. In *Proceedings of the Network and Distributed System Security Symposium*.

[22] google/AFL. 2024. *AFL-argvfuzz*. https://github.com/google/AFL/tree/master/experimental/argv_fuzzing.

[23] google/syzkaller. 2024. *Syzkaller*. https://github.com/google/syzkaller/.

[24] Graphviz. 2022. *DOT Language*.

[25] Joern. 2023. *JOERN-The Bug Hunter's Workbench*. https://joern.io.

[26] Arthur B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562. https://doi.org/10.1145/368996.369025

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[28] 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*.

[29] 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 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 48:1–48:12. https://doi.org/10.1145/3597503.3623315

[30] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining node. js vulnerabilities via object dependence graph and query. In *Proceedings of the USENIX Security Symposium*.

[31] libjpeg-turbo/libjpeg turbo. 2022. *Libjpeg-turbo*. https://github.com/libjpeg-turbo/libjpeg-turbo/issues/621.

[32] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis.. In *Proceedings of the 26th ACM Conference on Computer and Communications Security*.

[33] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[34] malteskoruppa/phpjoern. 2024. *Parser utility to generate asts from php source code suitable to be processed by joern*. https://github.com/malteskoruppa/phpjoern.

[35] netwide assembler/nasm. 2024. *nasm*. https://github.com/netwide-assembler/nasm.

[36] NLTK. 2024. *Natural Language Toolkit*. https://www.nltk.org/index.html.

[37] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 697–710. https://doi.org/10.1109/SP.2018.00056

[38] Barton P.Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[39] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[40] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*.

[42] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium*. 861–875.

[43] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331. https://doi.org/10.1109/SP.2010.26

[44] Faysal Hossain Shezan, Zihao Su, Mingqing Kang, Nicholas Phair, Patrick William Thomas, Michelangelo van Dam, Yinzhi Cao, and Yuan Tian. 2023. CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-language Code Property Graph. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/chkplug-checking-gdpr-compliance-of-wordpress-plugins-via-cross-language-code-property-graph/

[45] Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. 2020. CrFuzz: Fuzzing Multi-purpose Programs through Input Validation. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[47] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. https://doi.org/10.1145/2892208.2892235

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[49] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. 2023. CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing. In *Proceedings of the USENIX Security Symposium*.

[50] Xiaomeng Wang, Tao Zhang, Runpu Wu, Wei Xin, and Changyu Hou. 2018. CPGVA: Code Property Graph based Vulnerability Analysis by Deep Learning. In *10th International Conference on Advanced Infocomm Technology, ICAIT 2018, Stockholm, Sweden, August 12-15, 2018*. IEEE, 184–188. https://doi.org/10.1109/ICAIT.2018.8686548

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

[52] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2022. Registered Report: Fuzzing Configurations of Program Options. In *International Fuzzing Workshop (FUZZING)*.

[53] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* 32, 2 (2023), 53:1–53:21. https://doi.org/10.1145/3580597

[54] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. 2016. Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 2: Short Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/V1/P16-2034

A Bug Finding

The bugs found by OSmart are listed in Table 4.

B Graph Traversals

Filter Traversal. We define $T_{V(k:s)}$ to select the specific nodes whose value of k is s from the node set V.

$$T_{V(k:s)} = \{v | v \in V \ \&\& \ \mu(v_{cur}, k) = s)\}$$

Table 4: Vulnerabilities discovered by OSmart. RED represents bugs triggered by undocumented options. UAF=Use After Free, SEGV=Segmentation Violation, NPD=Null Pointer Dereference, FPE=Floating Point Exception.

Strategy	Program	Status	Type	OptNum
fix-option	gdb	confirm(issue-30176)	Memory Leak	1
	gdb	confirm(issue-30322)	Memory Leak	1
	gdb	submit(issue-30323)	Heap Overflow	2
	gdb	submit(issue-30638)	UAF	1
	gdb	submit(issue-30639)	Stack Overflow	1
	gdb	fixed(CVE-2023-39129)	UAF	1
	gdb	fixed(CVE-2023-39130)	Heap Overflow	1
	gdb	submit(issue-30667)	Global Overflow	1
	gdb	submit(issue-30668)	UAF	1
	gprof	fixed(issue-30324)	SEGV	3
	gprof	fixed(issue-30657)	Heap Overflow	1
	makeswf	submit(CVE-2023-31978)	UAF	1
	phpdbg	fixed(issue-10715)	Heap Overflow	1
	php	fixed(issue-9709)	NPD	2
	catdoc	submit(CVE-2023-31979)	Global Overflow	1
	img2sixel	submit(CVE-2023-31980)	FPE	1
	nidsasm	submit(issue-3392856)	Stack Overflow	3
	sngrep	fixed(CVE-2023-31981)	Stack Overflow	3
	sngrep	fixed(CVE-2023-31982)	Heap Overflow	3
	sngrep	fixed(CVE-2023-36192)	Heap Overflow	2
	yasm	submit(CVE-2023-30402)	Heap Overflow	3
	yasm	submit(CVE-2023-31973)	UAF	3
	yasm	submit(CVE-2023-31974)	UAF	3
	yasm	submit(CVE-2023-31972)	UAF	3
	yasm	submit(CVE-2023-31975)	Memory Leak	1
	tiffcrop	submit(issue-590)	Heap Overflow	2
	tiffcrop	submit(issue-593)	Heap Overflow	9
	tiffcrop	submit(issue-594)	Heap Overflow	9
	tiffcrop	submit(issue-595)	Heap Overflow	9
	w3m	confirm(issue-274)	UAF	2
	bsdcpio	fixed(issue-1934)	Stack Overflow	4
	bsdcpio	fixed(issue-1935)	SEGV	4
	gifsicle	confirm(issue-193)	FPE	2
fix-input	makeswf	submit(CVE-2023-31977)	Stack Overflow	2
	makeswf	submit(CVE-2023-31976)	Stack Overflow	2
	phpdbg	fixed(issue-9709)	Memory Leak	1
	jhead	fixed(issue-55)	Stack Overflow	1
	jhead	fixed(issue-54)	SEGV	1
	img2sixel	fixed(issue-65)	Double Free	1
	img2sixel	submit(issue-174)	SEGV	1
	tiffcrop	fixed(issue-450)	Stack Overflow	4
	fax2ps	fixed(issue-475)	Heap Overflow	1
	qemu-edid	fixed(issue-1249)	FPE	1
	qemu-img	fixed(issue-1629)	Heap Overflow	1
	gifsicle	fixed(CVE-2023-36193)	Heap Overflow	1
	htdbm	submit(issue-66637)	SEGV	2
	htdbm	submit(issue-66638)	SEGV	1
	htdbm	submit(issue-66639)	SEGV	1
	sqlite3	fixed(CVE-2023-36191)	SEGV	1
	catdoc	submit(issue-10)	Global Overflow	2
	cflow	fixed(CVE-2023-6031)	Global Overflow	2

Predecessor and Successor Traversals. We define the following *predecessor* and *successor* traversals to visit all predecessor and successor nodes within a property graph for the current visiting node, v_{cur}.

$$T_{Pre}(\{v_{cur}\}) = \{v_{pre} | (v_{pre}, v_{cur}) \in E\}$$

$$T_{Succ}(\{v_{cur}\}) = \{v_{succ} | (v_{cur}, v_{succ}) \in E\}$$

Impact Traversal. We define the *impact* traversal T_I to visit all paths starting from a specific node, e.g., the option node. In fact, this recursive definition is inspired by the *Traversal TNODE* defined in the work [16], which was used to identify all the AST child nodes of any root node.

$$T_I(\{v_{cur}\}) = \bigcup_{v \in T_{Succ}(\{v_{cur}\})} \left(v \cup \left(\bigcup_{v' \in T_{Succ}(\{v\})} T_I(\{v'\}) \right) \right)$$

Inspired by the work [16], we also extend predecessor and successor traversals by using edge label r and attributes k:s, i.e., T^r_{Pre}, $T^{r[k:s]}_{Pre}$, T^r_{Succ} and $T^{r[k:s]}_{Succ}$. Then, they can be used to filter the nodes by confining the edge e, i.e., satisfying $\lambda(e) = r$ or $\mu(e, k) = s$. Similarly, we also extend the impact traversal T_I with the T^r_I and $T^{r[k:s]}_I$.