FIXX: FInding eXploits from eXamples

Neil P Thimmaiah
University of Illinois Chicago
npemma2@uic.edu

Yashashvi J Dave
University of Illinois Chicago
yashashvidave@gmail.com

Rigel Gjomemo
University of Illinois Chicago
rgjome1@uic.edu

V.N. Venkatakrishnan
University of Illinois Chicago
venkat@uic.edu

Abstract

Comprehensively analyzing modern-day web applications to detect different vulnerabilities and related exploits is challenging and time-consuming. Security researchers spend significant time discovering and creating vulnerabilities and exploiting disclosures. However, such disclosures are often limited to single vulnerability instances and do not contain information about other instances of the same vulnerability in the application. In this paper, we propose FIXX, a tool that can automatically find multiple similar exploits from taint-style vulnerabilities inside the same PHP application. FIXX aims to help web application developers detect all possible instances of a known exploit within the program's code. To do this, FIXX combines novel notions of path and graph similarity over graph representations of code. We evaluate FIXX on 32 CVE reports containing cross-site scripting and SQL injection vulnerabilities associated with 19 PHP applications and discover 1097 similar exploitable paths leading to 10 new CVE entries.

1 Introduction

Taint-style vulnerabilities are a common type of weakness that occurs in a program when malicious data flows from input locations (called *sources*) to places that carry out sensitive operations (called *sinks*), like writing to a database. These vulnerabilities are common among web applications due to their processing input from untrusted sources.

Several approaches have been developed for analyzing taint-style vulnerabilities in web applications [11, 12, 15, 21, 23, 26, 29, 31, 33–35, 45, 46, 48, 49]. These approaches use sophisticated techniques, such as symbolic execution and constraint solving, to find vulnerable source-sink paths and corresponding exploits. Once such vulnerable paths are discovered, they are usually disclosed to developers and patched. However, those disclosures are often focused on a single vulnerability instance. According to Google's Project Zero, one in four 0-day exploits in 2020 were slight variations of previously disclosed 0-days and could have been easily prevented with more comprehensive patching [8]. According to the same post: "A correct patch is one that fixes a bug with complete accuracy, meaning the patch no longer allows any exploitation of the vulnerability. A comprehensive patch applies that fix everywhere that it needs to be applied, covering all of the variants".

A likely reason for this problem is that every analysis approach can have false negatives, i.e., vulnerabilities that are present in an application but are not discovered by that approach. This is often due to approximations in static analysis methods and insufficient coverage in dynamic analysis methods. Another likely reason is due to insufficient information about the vulnerability contained in the disclosure. In fact, often, the information in CVE entries contains only a location in the code and the type of vulnerability without describing the root causes of the vulnerability, and vulnerability disclosures are often inconsistent [17,25]. As a result, as mentioned earlier, patches may not be comprehensive.

Rather than finding new zero-day vulnerabilities, in this paper, we focus on an equally important problem: finding all exploitable instances of a known vulnerability and its variants in an application. Given a vulnerable location present in an application's program code through which the latter can be exploited, we aim to find all such locations that can be exploited in the same way.

Our approach is implemented in a tool called FIXX, which is focused on detecting cross-site scripting (XSS) and SQL Injection vulnerabilities in PHP applications. FIXX's first step is to process CVE reports about such vulnerabilities in PHP applications and extract useful information to reproduce the exploit. Once the exploit has been reproduced, FIXX uses the information from the observed execution to extract the source-sink data flow path of that exploit. Next, FIXX identifies instruction components of that path that are: 1) extensively reused in the rest of the application (e.g., function calls or array accesses), 2) sensitive operations (e.g., sink instructions like `echo` or sanitization functions). The key idea behind these two criteria is based on the intuition that additional vulnerabilities might exist elsewhere in the application because developers reuse the same code, which may contain sensitive

operations or be insufficiently sanitized. Next, FIXX finds instructions elsewhere in the application that have similar components. FIXX's final step after identifying such similar instructions elsewhere in the application is to extract paths from sources to those instructions and from those instructions to sinks and perform a path-based similarity ranking with the path observed during the exploit. Because of the way in which those paths were constructed, it is more likely that those paths are also vulnerable.

At the core of FIXX, is a novel *similarity* measure between instructions, which takes into account not only the names of the instruction components but also their types and structure of the instructions and the similarity of the corresponding data flow paths to the exploitable path. This similarity measure is used in different stages of the approach to identify highly reused instruction components and to identify similar instructions.

FIXX makes the following novel contributions:

- To the best of our knowledge, this is the first approach that detects new vulnerable paths that are similar to a known vulnerable path.
- We define and combine two metrics to identify reusable vulnerable code.
- We design a similarity measure between instructions and components of instructions that takes into account both identifiers and structure of an instruction.
- Finally, we evaluate FIXX on 300 CVEs and discover multiple unreported similar paths, leading to submitting 10 new CVEs.

This paper is organized as follows: Section 2 provides a detailed description of our problem and an overview of the solution. The architecture and details of our approach are presented in Section 3. The implementation details are provided in Section 4. Section 5 contains the evaluation. Sections 6 and 7 contain related work and conclusions, respectively. Finally, Section 8 is the ethics and Open Science Policy section.

2 Challenges and Approach Overview

To illustrate the challenges of the problem, let us consider the running example shown in Listing 1, inspired by CVE-2024-25868 about the PHP application CodeAstro Management System v1.0. After reading the vulnerability and reproducing it, we discover that the exploitable path is in the file `add_type.php`. This path contains two source instructions (Lines 4-5), one of which contains the parameter `membershipType`. The file also contains a database query that inserts this parameter into the table `membership_types` (Line 6). This parameter is retrieved from the database in numerous other files, including the file `list_renewal.php` (Line 16). Finally, the variable `membershipTypeName` created in Line 19 is echoed (Line 20). As can be seen from the example, the values flow unsanitized from the source node (Line 4) to the sink node (Line 20), giving rise to a cross-site scripting vulnerability. The vulnerable path followed is through the line numbers (3, 4, 5, 6, 7, 8, 9, 15, 16, 17, 18, 19, 20). We point out that even though the CVE description only describes the XSS vulnerability, this code also has an SQL injection vulnerability because unsanitized values are also used in the query in Lines 6-7.

```
1 //add_type.php
2 <?php
3 if ($_SERVER["REQUEST_METHOD"] == "POST") {
4     $membershipType = $_POST['membershipType'];
5     $membershipAmount = $_POST['membershipAmount'];
6     $insertQuery = "INSERT INTO membership_types VALUES
          ('$membershipType', $membershipAmount)";
7     if ($conn->query($insertQuery) === TRUE) {
8         $response['success'] = true;
9         $response['message'] = 'Membership type added
              successfully!';}
10    else {
11        $response['message'] = 'Error: '.$conn->error;}}
12 ?>
13 //list_renewal.php
14 <?php
15    $membershipTypeId = $row['membership_type'];
16    $membershipTypeQuery = "SELECT type FROM membership_types
          WHERE id = $membershipTypeId";
17    $membershipTypeResult = $conn->query($membershipTypeQuery);
18    $membershipTypeRow = $membershipTypeResult->fetch_assoc();
19    $membershipTypeName = ($membershipTypeRow) ?
          $membershipTypeRow['type'] : 'Unknown';
20    echo "<td>{$membershipTypeName}</td>";
21 ?>
```

Listing 1: Vulnerability in CodeAstro

```
1 //view_type.php
2 <?php
3 ...prev code
4 if ($_SERVER["REQUEST_METHOD"] == "POST") {
5     $membershipType = $_POST['membershipType'];
6     $membershipAmount = $_POST['membershipAmount'];
7     $insertQuery = "INSERT INTO membership_types (type, amount)
          VALUES ('$membershipType', $membershipAmount)";
8     if ($conn->query($insertQuery) === TRUE) {
9         $successMessage = 'Membership type added successfully!';
10        // header("Location: dashboard.php");
11        // exit();
12    } else {
13        echo "Error: " . $insertQuery . "<br>" . $conn->error;
14    }
15 }
```

Listing 2: Similar vulnerable code in viewtype

In addition to this vulnerability, there are other vulnerabilities present throughout the application that are not present in the CVE description and haven't been identified to this date. Listing 2 shows one such XSS vulnerability in file `view_type.php`. Here, unsanitized values from the source in line 5, can reach the sink in line 13. We point out that there are several similar components to Listing 1 in this code, including the database table name, variable names such as `$insertQuery`, `$membershipType`, etc. This vulnerability, however, is not included in the original or other CVEs. This example illustrates our problem stated below.

Problem statement: Given a proof of a vulnerability as an executable path, can we use the knowledge present in that path to discover similar vulnerable paths?

2.1 Challenges

Code Reusability. One of the main challenges in detecting

similar vulnerable paths is understanding how code is reused across an application. Such reuse may rise from different programming practices, such as object-oriented programming, MVC patterns, or program readability (e.g., using the same variable names across the application for variables that have the same functionality). For instance, the same identifier may appear in multiple places inside the application and play multiple roles inside instructions. In addition, as is often the case, even if two components (e.g., function calls) are semantically the same, syntactically, they may be different (e.g., function calls with different input arguments). To solve this challenge, we need to define a similarity measure that is able to take into account the structure of the instructions together with the identifiers, but that is flexible enough to accommodate syntactic variations between similar instructions.

Selectivity. Another challenge in solving this problem is that of focusing the search for other vulnerable paths in locations that are most likely to be vulnerable. In fact, assuming a similarity measure is available, how do we search for other vulnerable similar paths? Naively searching for where the instructions of the vulnerable path are reused might make the search space too large, as each instruction may be reused in multiple locations. For instance, each instruction in the vulnerable path in Listing 1, appears in multiple other locations in the application, however, only Listings 2 and 3 contain a vulnerable path. To solve this challenge, we need a method that is able to take into account the likelihood that a path is vulnerable.

To address this challenge, our approach must be able to detect not only exact replicas of a known vulnerability but also variants of that vulnerability. Based on these discussions, we define our problem as follows:

Prior work in the area of code similarity detection includes different methods mainly focused on Java and C/C++ applications [10, 19, 28, 30, 38, 40, 44]. A large component of these works aims to detect plagiarized code, that is, code that performs the same operations as a given code fragment [37]. In particular, prior literature has defined 4 types of code clones: (1) formatting and layout (Type I), (2) identifier renaming (Type II), (3) structural changes (e.g., replacing `for` with `while`) or statement deletions, (4) extreme code modifications that preserve semantics but change the syntax significantly (Type IV) [37]. These works, however, consider the code similarity of code fragments as a whole and are not aimed at identifying similar vulnerabilities. As a result, as we will discuss later, they would require some modifications to be able to find vulnerable paths.

2.2 Approach Overview

Our goal is to build an automated approach for uncovering such false negatives. We aim to use the exploits discovered by existing approaches to detect additional instances of those exploits. Our tool called FIXX, works in different phases. First, the exploit is reproduced by installing the vulnerable PHP

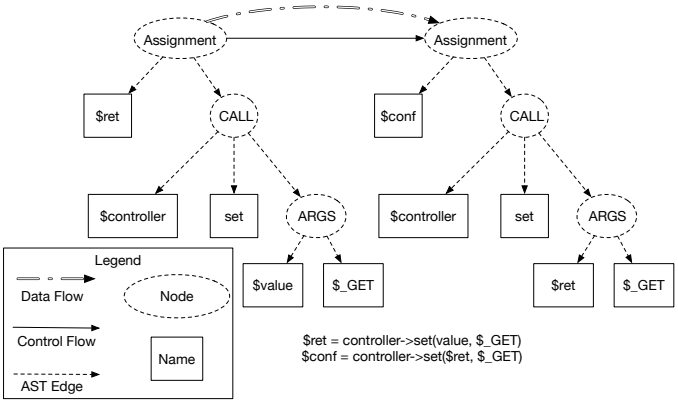


Figure 1: Code Property Graph example.

application and following the instructions present in the vulnerability report. Next, FIXX retrieves the execution dataflow and, from the latter, extracts those *reusable* and *sensitive* sets of components that are most likely to lead to an exploit. For instance, function calls in the trace receiving inputs or wrapper functions around database operations are examples of such sensitive components. To do this extraction, we first define two scoring mechanisms for instructions. These include a *reusability score* and a *sensitivity score*, which capture how often an instruction is reused elsewhere in the application and how important that instruction is in the overall exploit. We use these two scoring mechanisms to extract the most important components of the reported exploitable path. Next, these instructions are used as *seeds* to find new exploitable paths in the application. In particular, these seeds are used to identify new instructions in the application that are similar to them. These new instructions are then treated as starting points to find data flow paths from sources to sinks that contain them. Finally, we measure the similarity of each of these new data flow paths with the original exploitable path reproduced earlier. We thus define two types of similarities, an *instruction* similarity measure, and a *path* similarity measure. Once similar paths are identified, FIXX verifies their exploitability via a semi-automated approach, which involves symbolic execution and dynamic and manual analysis.

Code Property Graphs. We introduce the code representation model used in FIXX as a preliminary discussion. A Code Property Graph (CPGs) is a code representation model that combines different information about the application into the same graph-based abstraction layer, [47]. Most commonly, they combine Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Dependency Graph (DDG), and Call Graph (CG) in the same graph. This enables queries that can combine constraints from all these sources of information. For instance, a CPG can be used to find control flow paths between calls to methods of the same class (information on the AST) where the value of at least one variable that is local to those methods has a data dependency on an HTTP input parameter (information on the DDG). Figure 1 shows a small example of a code property graph. As can be seen

from this figure, each instruction is represented by a node, and the names of the respective components are represented by special *Name* nodes. FIXX uses code property graphs extensively to retrieve paths and compute similarity among instructions.

3 Architecture and Algorithms

In this section, we discuss the details of our approach. An overview of FIXX's architecture is shown in Figure 2. First, an *Information Extractor* module is used to process natural language vulnerability reports and extract information related to reproducing the exploit. Next, the exploit is executed by the *Exploit Executioner* module, which returns exploit subgraphs, a representation of the exploit path and its control dependencies. The exploit subgraphs are further analyzed by the *Seed Identification* module, which identifies special instructions (*seeds*) that are highly reused and sensitive. The *Path Detector* module uses the seeds to find similar nodes in the CPG and, starting from those nodes, find paths similar to the exploit path in the CPG. These paths are then sent to the *Verifier* module to check whether they are actually exploitable. In the rest of this section, we discuss details about each of these steps.

3.1 Identifying Vulnerability Root Causes

The first phase in our approach is used to characterize the vulnerability knowledge and its causes. FIXX does this by executing the exploit present in the CVE description and collecting data and control dependencies along the execution paths from sources to sinks.

3.1.1 Vulnerability Report Knowledge Extraction

The first step in characterizing a vulnerability is extracting as much information as possible about the vulnerability in order to be able to reproduce the exploit. While this task may be executed manually on a case-by-case basis, FIXX is able to perform it at scale with the help of GPT-4 [9], which automatically extracts helpful details such as 1) application name and version, 2) file name related to the vulnerability, 3) parameter names or code elements mentioned in the CVE, e.g., function names, variable names, etc. and 4) actual malicious inputs that exploit the vulnerability. In particular, we fine-tune GPT-4 with a set of 50 CVE descriptions describing the different components that we are interested in, i.e., file names, application names and versions, and weak code components that are responsible for the vulnerability.

The output of this step includes a report containing details about reproducing the exploit. When these details are exhaustive, including the HTTP requests and parameters, the exploit can be reproduced automatically with minimal human effort. In many cases, however, manual effort is needed to install and set up applications and to fill in the missing details for the exploits to work.

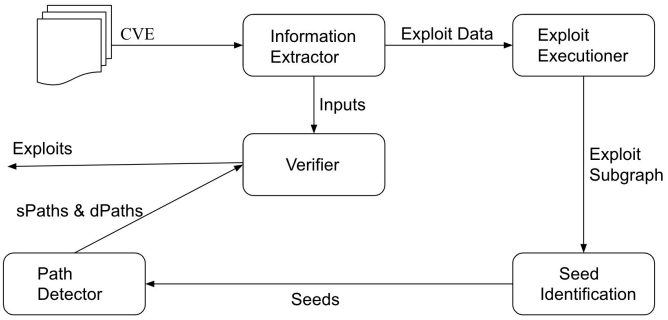


Figure 2: FIXX's architecture.

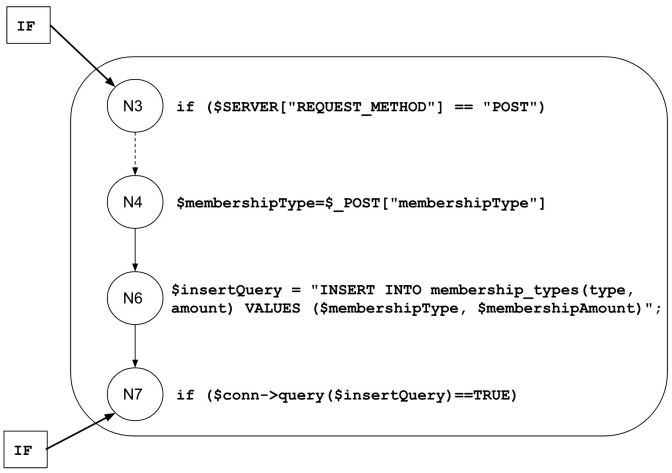


Figure 3: Exploit Subgraph.

3.1.2 Exploit Executioner

After retrieving the details of the vulnerability from its CVE description, we reproduce the exploit and collect a trace of the executed PHP instructions.

Extracting Exploit Dataflow. Next, using the execution trace, FIXX extracts a data flow path D_P from the CPG, which contains only those instructions that are present in the trace. In addition to the data flow path, FIXX uses the CPG to collect the control dependencies of the instructions in that path. For instance, the source statements in lines 4 and 5 in Listing 1 are only executed if the conditional statement in line 3 evaluates to True. In fact, these control dependencies may contain important clues in understanding the possibility of an exploit and looking for similar instances of that exploit. FIXX identifies the conditions under which the execution of the instructions in the exploit data flow D_P occurs by performing a backward traversal of the CPG to identify additional control dependencies.

This model is represented as a subgraph of the CPG containing the dataflow path D_P and related control dependencies whose satisfaction makes that feasible. Each node in this subgraph is annotated with the state of the application observed during the exploit execution. In the rest of the paper, we refer to this subgraph by *Exploit Subgraph*. A portion of the exploit subgraph related to our Listing 1 is shown in Figure 3.

3.2 Detecting Similar Exploits

After obtaining the exploit subgraph, we must find paths or subgraphs that are similar to that subgraph. In our initial implementation of FIXX, we tried using the whole subgraph to find other similar graphs in the application. This approach was similar to AST- and PDG-based approaches for finding code clones [28, 30, 38, 44]. However, after a quick analysis of the results and many false positives, we concluded that a particular vulnerability execution represented by the exploit subgraph is still highly dependent on the application location and unlikely to be repeated in exactly the same or similar structure elsewhere. The main intuition and difference with code cloning approaches is that rather than finding clones of the exploit subgraph, to detect similar vulnerabilities, we need to find similar vulnerable instructions in similar paths. For instance, SQL queries themselves such as N6 in Figure 3 do not appear quite often in the rest of the application but instructions like N4 that contain sensitive superglobals and N7 that contain calls to query functions appear frequently throughout the application. In addition, there are other instructions, such as sanitization functions, and common sink functions, that are particularly important for the exploit.

Therefore, differently from code clone detection approaches, we need to select a small number of "important" instructions from the exploit subgraph that can serve as *seeds* to find other similar paths. We call this task *seed identification*. To identify important nodes that may lead to similar exploits, FIXX uses two criteria, namely *reusability* and *sensitivity*, which capture and combine characteristics of instructions that are more likely to be reused elsewhere in the application and lead us to an exploit.

3.2.1 Seed Identification

Reusability score. Modern object-oriented applications typically reuse a significant portion of their code, especially in the form of function and class method calls [41]. The appearance of these components in the exploit subgraph is a direct indication that they do not perform sufficient sanitization. Thus, their use elsewhere in the application increases the likelihood of a vulnerability threat being present along those paths.

FIXX defines the reusability score of an instruction component C as the number of times that that component appears in the application divided by the total number of times all components in the exploit subgraph E_G appear in the application.

$$R(C) = \frac{C}{\sum_{i \in E_G} C_i}$$

Ranging from 0 to 1, the reusability score introduces a sorting criterion among the components of an exploit subgraph that can be used to select the seeds. In particular, we consider different component types, as shown in Table 1, which are the building blocks of the majority of instructions in PHP. We note that variables include scalar variables, arrays, and arguments of function calls. Because superglobals are usually arrays that contain malicious input during a taint-style exploit, we include them as a (special) component type as well. We also point out that a single instruction may contain multiple components.

Component	Example Name	Reus. Score
Variable	$membershipType	0.007
Superglobal	$_POST	0.04
Constant	"successfully"	0.0001
Method	conn→query($membershipTypeQuery)	0.003

Table 1: Reusable Components

For each instruction containing one or more components in the exploit subgraph and for each component, FIXX scans the CPG of the application and counts the number of components with the same name. It then computes the reusability score for that component. Finally, the reusability score of the instruction is the sum of all the reusability scores of the components in that instruction. The third column of Table 1 shows the score of some of the components present as part of nodes in a CPG. As shown in the table, the `$_POST` method has a higher reusability score than the other two components. An issue that arises when counting occurrences is with regard to components having complex datatypes that encapsulate other components, such as arrays and function calls. For instance, consider the expression `$_POST['membershipType']`. We can count this as an occurrence of the array `$_POST` or as the occurrence of the constant `membershipType` or as the occurrence of the array element represented by the expression. To resolve this issue, we create different rules depending on the context and the types of components in the instructions. The high-level idea behind these rules is to count occurrences of elements that participate in control or data flows at their highest granularity. For instance, references to array elements are counted as occurrences of the array element rather than occurrences of the array, polymorphic function calls are counted as occurrences of the corresponding form depending on the number of arguments, etc.

Sensitivity score. Intuitively, the sensitivity score focuses on the importance of an instruction towards an exploit. In particular, this score captures domain knowledge about instructions that are sensitive and likely to appear in exploit paths. For instance, special source components (e.g., `$_GET`) and sink functions (e.g., `mysql_query`, `echo`) have a higher sensitivity than other instructions because they are usually the attackers' targets. Likewise, the presence of a call to known sanitization functions (e.g., `htmlspecialchars`) in an exploit subgraph is usually more important than access to, say, a user-defined array, as it refers to the quality of sanitization. Therefore, it makes more sense to search for similar components in the application first. If an instruction has multiple sensitive components, the sensitivity scores of those components are added to obtain the sensitivity score of the instruction. In FIXX, the names of the sensitive components must be given as input to FIXX by an analyst. In our current implementation, we use a general list of sensitive components containing superglob-

als, common sink functions such as `echo` and `mysqli_query`, and known sanitization functions.

In addition to these base heuristics for sensitive components, FIXX also considers methods that may serve as wrappers of those components. For instance, a common practice in several applications is to create a class that is responsible for database queries and wraps `mysql_query` inside methods of that class. In these cases, there may be only one call to `mysql_query` in the application but many calls to its wrapper methods. Thus, while the reusability score of the sensitive component is low, the reusability score of the wrapping function is high. Therefore, we need to propagate the sensitivity level of `mysql_query` to the functions that call it.

To propagate the sensitivity level of a sensitive component C to a function F that calls that component, we estimate the likelihood that C is executed when F is called. This likelihood is equal to the number of control flow paths inside F that contain C divided by the total number of control flow paths inside F. For instance, if all a function's paths execute `mysql_query`, then the likelihood that `mysql_query` is executed when the function is called is 100%. We refer to this value by *execution likelihood* of C inside F, $E(C,F)$, in the rest of the paper.

Based on these discussions, we define the sensitivity score S_C of a function or method called F as follows.

$$S_C(F) = \begin{cases} 1: \ \ S_C \in \text{Sensitive components list} \\ \sum_{C_i \in F} E(C_i,F) \cdot S_C(C_i) \ : \ \text{otherwise} \end{cases}$$

A component belonging to the list of highly sensitive components provided by a domain expert has a sensitivity score equal to 1. Starting from these components, the formula can recursively compute the sensitivity score of every function that executes such a component in one or more of its paths. In particular, if the function F executes multiple components C_i with non-zero sensitivity score, then F's sensitivity score is equal to the sum of the products of the execution likelihood of each C_i inside F with the sensitivity score of C_i.

Selecting Seeds. FIXX calculates a final *selection score* value as the product of the reusability score and the sensitivity score for each instruction of the exploit subgraph. Next, the components are sorted in decreasing order of their selection scores and sent to the next phase of the approach.

3.2.2 Finding Similar Paths

After the seed identification extracts the seeds as a ranked list, the *Path Detector* module is responsible for finding paths that are similar to the exploit path. To do so, this module selects the top instructions of the seeds list and uses them as starting points for discovering new paths. In our implementation, we use between 1 and 4 top most reused and sensitive instructions from the seed list to search for similar paths. As mentioned earlier, our goal is to find not only exact replicas of a vulnerability but also similar variants of that vulnerability. To capture these variants, we introduce a "calibration knob" in the calculation of similarity between components.

Definition 3.1 (AST-based similarity modulo X) *Two instruction nodes of a code property graph are said to be* similar modulo X*, where X is a non-negative integer, if their AST subtrees have the same structure and node types, and there are X corresponding* Name *nodes that are different.*

This definition captures a variable notion of similarity that enables FIXX to match components beyond strict identity and thus detect variants of seeds and, by extension, vulnerable paths. It compares the AST structure of different instructions with those of a seed, allowing for X differences in the names of the various components present in the instruction. For instance, two instructions are *similar modulo 0* if they have identical AST subtrees and all the Name nodes are the same. As another example, the two `Assignment` nodes in Figure 1 are similar modulo 2 because they differ by two name nodes, while the `Call` nodes are similar modulo 1 because they differ by only one name node. This definition of similarity allows us to capture different syntactic expressions of potentially similar nodes.

Dataflow-based similarity Using this notion of similarity, the *Path Detector* module finds other similar nodes throughout the application that are similar to each of the top seeds and, using these nodes as starting points, finds data flow paths from sources to those nodes (*sPaths*) and data flow paths from those nodes to sink statements (*dPaths*) as shown in Algorithm 1. In particular, the algorithm iterates over the ranked list of seeds (Line 5) and, for each one of those seeds n, finds similar nodes modulo *mod*, where *mod* is a non-negative integer (Line 6). It then iterates over each node s of these similar nodes and: 1) it retrieves *sPaths*, the set of paths from source statements to the node s (Line 11) and *dPaths*, the set of paths from node s to sink statements (Line 15), 2) for each path in *sPaths* (and *dPaths*) it counts the number of nodes that are similar modulo between that path and the predecessors (or successors) (Lines 14, 18) of the seed n in the exploit subgraph D. It then sorts the two sets of paths *sPaths* and *dPaths* in decreasing order of the number of similar nodes (Lines 19, 20) and returns a set each of *sPaths* and *dPaths* that are greater than a given threshold value (Line 21). This threshold is provided as an input when counting the number of similar nodes, and only those *sPaths* and *dPaths* that have a similarity greater than the threshold are obtained.

It is important to note here that the algorithm does not try to match the sequences of instructions exactly in the exploit subgraph to find similar paths (*sPaths* and *dPaths*). Rather, it sorts the paths it discovers starting from nodes similar to the seeds based on the overlap between those paths and the exploit subgraph. We made this design choice to make the detection of similar paths somewhat robust concerning possible differences in similar data flows in the application. The final output of the algorithm is a set of paths ranked by the

Algorithm 1 Similar Paths Identification.

```
1: Input: Seeds S, Exploit subgraph D, Modulo mod, Threshold
2: Output: Similar paths list
3:
4: Paths ← ∅
5: for n ∈ S do
6:     Sim_nodes = find_similar_nodes(n, mod)
7:     for s ∈ Sim_nodes do
8:         Paths = Paths ⋃ SIM_PATHS(s, n, D, mod)
   return Paths
9:
10: function SIM_PATHS(s, n, D, mod)
11:     sPaths = getSourcePaths(s)
12:     for sPath ∈ sPaths do
13:         anc = ancestors(n, D)
14:         B[sPath] = count_sim_nodes(sPath, threshold, predecessors)
15:     dPaths = getDestinationPaths(s)
16:     for dPath ∈ dPaths do
17:         desc = descendants(n,D)
18:         F[dPath] = count_sim_nodes(dPath, threshold, successors)
19:     Sort(sPaths, B)//decreasing by most similar nodes
20:     Sort(dPaths, F)//decreasing by most similar nodes
21:     return sPaths, dPaths //with similar nodes greater than threshold
```

degree of overlap with the nodes in the exploit subgraph. This set of paths is built starting from seed nodes that are highly reused across the application and highly sensitive according to domain expert knowledge.

3.2.3 Verification

The final step in FIXX is to verify that the paths identified by the path detector module are actual exploits. As shown in the architecture figure, the Verification module receives in input the similar paths found by the path detector module and the malicious inputs present in the original exploit. We then execute the application using those malicious inputs and obtain the execution trace and confirm whether the input reaches the sink. For those cases that do not reach the sink, we first try to obtain new malicious inputs for those similar paths using symbolic execution and constraint-solving step. If this step fails, we manually check the similar paths to determine inputs, execute the paths with those inputs, and see if the inputs led to an exploit. We describe these steps in more detail next.

Symbolic execution. In this step, the Verifier module retrieves the data flow paths for each of the *sPaths* and *dPaths* returned in Algorithm 1 and builds the expanded *exploit subgraph* for each of them. Next, it transforms each *exploit subgraph* it receives into a symbolic formula and augments that formula with additional constraints about malicious variables at the sink. Next, is uses an SMT solver to determine malicious values for the variables at the sources that lead to those malicious values at the sinks. It is important to note here that translating the instructions from PHP to the language of the SMT solver, which in FIXX is Z3 [24], can be quite challenging due to different semantics (e.g., Z3 requires type definitions, while PHP types are resolved at run time) As a result, the solution produced by the SMT solver, which contains the malicious inputs that need to be sent to the application, may contain inaccuracies that must be solved manually.

Dynamic execution. In this step, we execute each path by sending the inputs produced by the SMT solver in the previous step and check whether the malicious payload reaches the similar node and the sensitive sink node, respectively. If so, the execution is recorded.

Manual verification. Finally, if the previous steps fail, a human user manually creates a malicious payload and verifies whether the payload reaches the similarity node and sensitive sink node, respectively.

4 Implementation

Information Extractor. In order to extract the corresponding components mentioned in a given CVE description, we use Open AI's GPT-4 model. We used 50 CVE examples to fine-tune the model using OpenAI's APIs. The examples include the application name, version number, file name vulnerability type, and code components. Next, we request the fine-tuned version of GPT4 to extract the application name, version number, file name, vulnerability type, and code components from the rest of the CVE descriptions.

Execution Traces. To retrieve execution traces, we use the XDebug APIs to perform a step-by-step execution of a PHP file with a specific HTTP request built using information from the CVE description and the Z3 Solver (in the Verifier module). The XDebug API returns the executed lines together with the state of the application at each line.

CPG Construction. Instruction nodes that make up the dataflow subsequence of the malicious file are extracted using the CPG of the application. The CPG of every application is built using PHP Joern [5].

Exploit Executioner. To reproduce the exploit, we need to set up the application and understand the vulnerability. This step involves some manual work because PHP applications usually have different setup procedures. Once the application is set up and the exploit reproduced, the Exploit Executioner is responsible for extracting the exploit subgraph and identifying the seeds using reusability and sensitivity scores. This module is implemented using Cypher queries on a Neo4J graph database representing the code property graph [6, 7]. The Cypher queries retrieve the dataflow paths using the data dependency edges on the graph and the control dependencies (enclosing conditional statements).

Path Detector. This module executes Algorithm 1 using a value of 0, 1 and 2 for the *modulo* parameter and a value of 50% for the threshold to obtain the most similar nodes and

paths, respectively. Similar to Path Detector, this module is implemented using the Neo4J graph database and Cypher queries.

Verifier. The translation of paths into constraint formulas and their augmentation with additional constraints is implemented by a Python module that translates paths into Z3 constraint formulas [24]. With string types being an important aspect of our approach and Z3 supporting this type, we use Z3 as our SMT solver. In the case of false negatives, a human user manually creates a malicious payload and verifies whether the payload reaches the similarity node and sensitive sink node, respectively.

5 Evaluation

Application and CVE selection. To evaluate FIXX, we selected 300 recent CVE entries about XSS and SQL injection vulnerabilities spanning 209 applications. All these CVEs have been published between 2019 and 2025. We obtained these CVE entries using the search function of the CVE MITRE website, where we searched for XSS and SQL injection vulnerabilities for PHP applications. We further processed each of these CVE entries using our fine-tuned GPT-4 model and were able to reproduce the exploits for 132 of these CVEs. We could not reproduce the exploit in the other CVEs for several reasons. First, the reported exploits were associated with applications that were problematic to install. Second, the instructions to reproduce these exploits were vague or contained conflicting information for reproducing the vulnerability.

Results Summary. Among the 132 CVEs for which we could reproduce the exploit, we found and confirmed multiple additional similar paths for 32 of them. Using FIXX, we successfully verified many of these paths as actual exploits discovering 19 applications containing a total of 1097 exploitable paths. We have submitted some of these paths to MITRE and obtained 10 new CVEs (multiple paths have been submitted for a single CVE). We plan to continue submitting additional CVEs.

To evaluate our approach, we executed FIXX with different values for the number of seeds (1–4 seeds), similarity modulo in the instruction similarity measure (0, 1, and 2), and threshold values in finding *sPaths* and *dPaths* (with thresholds of 50% or more). The results, along with the corresponding application names and CVE IDs, are reported in Table 3, which shows the results for the values of 4 seeds, a similarity modulo of 2, and a similarity threshold of 50%. In this table, the first column shows the name of the application followed by the last two digits of the CVE-ID. The version of the application is present in the CVE description in the CVE entry shown in the third column. The second column of Table 3 shows the lines of code of each application. In particular, we test both large and small applications. Column 4 displays the total number of similar nodes found in other locations in the program, and column 5 shows the total number of *sPaths*, column 6 shows the total number of *dPaths*, column 7 contains the total number of exploitable paths obtained by verifying the *sPaths* and *dPaths*, respectively. Column 8 shows the results of an additional experiment in our evaluation. Specifically, to understand the robustness of our approach, we reverse the role of the paths, i.e. we check whether FIXX is able to discover the original exploitable path mentioned in the starting CVE by using newly discovered exploitable paths as exploit examples. As can be seen in Table 3, FIXX is able to discover the original CVE path in most of the applications. Finally, the last column contains the new CVEs that have been published for each application. The results prove that an exploitable path reported in the form of a CVE can lead to multiple similar exploitable paths present throughout the application. We want to point out that the vulnerabilities were not patched in many of the subsequent versions of these applications. In the rest of this section, we provide additional details regarding our evaluation, focusing on the following questions:

Application	Avg Reus.	Avg Sens.	Avg. Sel. Score	sPaths + dPaths per max no. of seed s_1-s_4
SeoPanel02	0.92	200	1.83	s_1-0 | s_2-14 | s_3-0 | s_4-0
Collabtive98	0.087	33.34	0.084	s_1-16 | s_2-0 | s_3-0
CE Phoenix58	2.31	175	4.07	s_1-0 | s_2-0 | s_3-0 | s_4-0
Clansphere10	0.67	125	0.77	s_1-0 | s_2-0 | s_3-19 | s_4-1
Fan-Blog24	0.39	100	0.39	s_1-0 | s_2-0
Fan-Blog12	2.52	100	2.52	s_1-11
Fan-Blog31	2.52	100	2.52	s_1-44
Eng-Onl-83	10.24	100	10.24	s_1-0 | s_2-264 | s_3-1 | s_4-0
Eng-Onl-76	16.95	100	16.95	s_1-121 | s_2-13 | s_3-0 | s_4-119
Hos-Manag-11	3.50	75	2.14	s_1-8 | s_2-23 | s_3-0 | s_4-146
Advocate28	2.28	50.01	0.33	s_1-5 | s_2-0 | s_3-22 | s_4-0
Aut-Enrol-94	11.36	0.01	0.0011	s_1-95 | s_2-48 | s_3-0 | s_4-0
Aut-Enrol-26	6.05	100	6.05	s_1-65 | s_2-0
Tailor60	5.10	33.34	0.81	s_1-63 | s_2-36 | s_3-0 | s_4-0
Tailor73	6.10	125	7.45	s_1-0 | s_2-131 | s_3-0 | s_4-0
Fruit Bazar89	5.93	100	5.93	s_1-0 | s_2-0 | s_3-2 | s_4-2
Fruit Bazar78	4.33	100	4.33	s_1-0 | s_2-0 | s_3-1 | s_4-0
Blood Bank27	7.633	100	7.633	s_1-1 | s_2-0 | s_3-4 | s_4-1
Blood Bank04	8.86	100	8.86	s_1-11 | s_2-83 | s_3-1 | s_4-0
Covid19tms04	8.73	100	8.73	s_1-1 | s_2-44 | s_3-0 | s_4-0
Covid19tms03	8.26	100	8.26	s_1-0 | s_2-0 | s_3-43 | s_4-2
Dairy Farm93	8.61	100	8.61	s_1-0 | s_2-33 | s_3-1 | s_4-0
CodeAstro68	6.20	50.01	2.33	s_1-38 | s_2-1 | s_3-4 | s_4-0
CodeAstro72	4.41	100	4.41	s_1-44 | s_2-1 | s_3-0 | s_4-0
CodeAstro33	4.62	100	4.62	s_1-17 | s_2-1 | s_3-0 | s_4-0
Loan-Man-90	9.46	100	9.46	s_1-22 | s_2-38
Daily-Exp06	8.81	100	8.81	s_1-2 | s_2-10 | s_3-0 | s_4-0
Daily-Exp04	8.95	100	8.95	s_1-0 | s_2-27 | s_3-1 | s_4-0
Vis-Manag-83	10.33	100	10.33	s_1-19 | s_2-2 | s_3-0 | s_4-0
Vis-Manag-60	2.29	75	1.76	s_1-9 | s_2-1 | s_3-0 | s_4-0
Vis-Manag-61	10.33	100	10.33	s_1-19 | s_2-2 | s_3-0 | s_4-0
Keerti00	14.19	100	14.19	s_1-10 | s_2-2 | s_3-0 | s_4-0

Table 2: Sensitivity and Reusability Scores For The Maximum Possible No. of Seeds (s_1-s_4) For Each Application

1. How well do the different steps of FIXX perform, and how does performance vary with the the different parameters in FIXX?
2. How well does FIXX perform in detecting additional similar paths?
3. How does FIXX compare with other approaches?

Application Name	LOC	Original CVE	Sim. Nodes	sPaths	dPaths	Exp. Paths	Re-Discover CVE	New CVEs
SeoPanel02	255.0k	CVE-2021-3002 (XSS)	617	14	0	2	No	-
Collabtive98	180.4k	CVE-2021-3298 (XSS)	19	0	16	9	No	CVE-2024-46240 CVE-2024-48706 CVE-2024-48707 CVE-2024-48708
osCommerce CE Phoenix58	59.8k	CVE-2020-12058 (XSS)	13	0	0	0	No	-
Clansphere CMS10	54.3k	CVE-2021-27310 (XSS)	26	0	20	20	No	-
FantasticBlog12	24.7k	CVE-2022-28512 (SQL)	1	2	9	9	Yes	-
FantasticBlog31	24.7k	CVE-2021-26231 (SQL)	1	2	42	42	Yes	-
FantasticBlog24	24.7k	CVE-2021-26224 (XSS)	4	0	0	0	No	-
Engineers Online Portal83	15.6k	CVE-2023-5283 (SQL)	171	0	265	258	Yes	-
Engineers Online Portal76	15.6k	CVE-2023-5276 (SQL)	83	11	242	52	Yes	-
Hospital Management System11	9.4k	CVE-2021-39411 (XSS)	144	6	171	6	Yes	CVE-2024-46237 CVE-2024-46238 CVE-2024-46239
Advocate Office Management System28	9k	CVE-2024-9328 (SQL)	87	17	10	21	Yes	-
Automated Enrollment94	7.7k	CVE-2021-3294 (XSS)	50	10	133	21	Yes	-
Automated Enrollment26	7.7k	CVE-2021-26226 (SQL)	21	0	65	0	No	-
Tailor Management60	7.7k	CVE-2021-40260 (XSS)	101	10	89	86	Yes	-
Tailor Management73	7.7k	CVE-2020-36073 (SQL)	136	0	131	130	Yes	-
Fruits Bazar89	6.4k	CVE-2022-34989 (SQL)	198	2	2	4	Yes	-
Fruits Bazar78	6.4k	CVE-2022-30478 (SQL)	192	0	1	1	Yes	-
Blood Bank System27	4.8k	CVE-2024-9327 (SQL)	93	5	1	5	Yes	-
Blood Bank System04	4.8k	CVE-2024-9804 (SQL)	100	11	84	84	Yes	-
Covid19tms04	4.1k	CVE-2024-53604 (SQL)	39	1	44	42	Yes	-
Covid19tms03	4.1k	CVE-2024-53603 (SQL)	36	0	45	43	Yes	-
Dairy Farm Management93	3k	CVE-2023-41593 (XSS)	39	0	34	33	Yes	CVE-2024-46241
CodeAstro68	2.6k	CVE-2024-25868 (XSS)	62	4	39	38	Yes	CVE-2024-46236 CVE-2024-48709
CodeAstro72	2.6k	CVE-2024-46472 (SQL)	59	0	45	44	Yes	-
CodeAstro33	2.6k	CVE-2024-2333 (SQL)	42	0	18	17	Yes	-
Loan Management90	2.2k	CVE-2024-9090 (SQL)	6	2	58	40	Yes	-
Daily Expense Tracker06	1.7k	CVE-2020-10106 (SQL)	35	2	10	10	Yes	-
Daily Expense Tracker04	1.7k	CVE-2021-26304 (XSS)	23	0	28	27	Yes	-
Visitor Management83	1k	CVE-2024-22983 (SQL)	19	0	21	21	Yes	-
Visitor Management60	1k	CVE-2020-25760 (SQL)	11	0	10	0	Yes	-
Visitor Management61	1k	CVE-2020-25761 (XSS)	19	0	21	21	Yes	-
Keerti00	968	CVE-2024-1700 (XSS)	19	0	12	11	Yes	-

Table 3: Evaluation Results for a Threshold Value of 50 and Similarity Modulo Value of 2

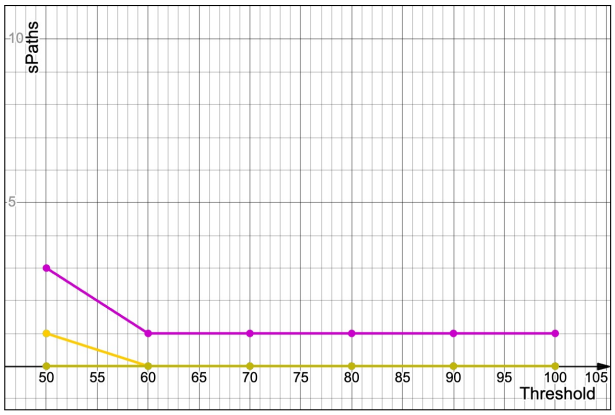
5.1 Steps Evaluation

5.1.1 Seed Selection Results

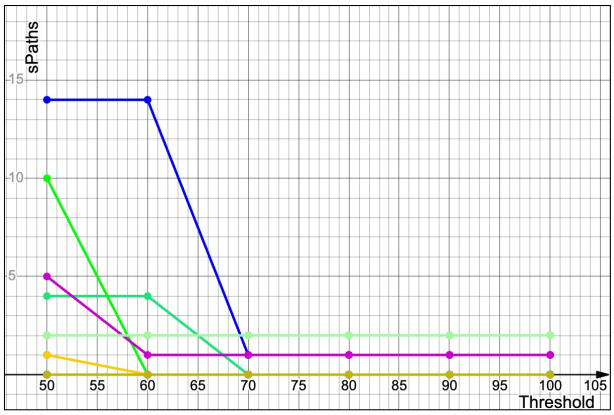
We collect the reusability and sensitivity scores as well as the selection scores for the applications to understand how well FIXX can select seeds that lead to final similar exploitable paths. We show these measurements in Table 2. For each application, the table shows in columns 2, 3, and 4 the average values of the reusability, sensitivity, and selection scores over the top 1–4 seeds, respectively. Due to their respective exploit subgraphs, certain applications like FantasticBlog12 (Fan-Blog-12) and Loan Management90 (Loan-Man-90) tend to have fewer seeds, 1 and 2 respectively, than the rest of the applications. As seen in the table, almost all applications have a high average selection score which shows that the the seeds selected by FIXX are highly sensitive and reused a lot throughout the corresponding application. The last column shows the combined *sPaths* and *dPaths* found per seed denoted by s_i. As can be seen, the top seeds indicated by s_1 contribute to more paths than the lower ranked seeds in most of the applications. In addition, the lowest seeds indicated by s_4 do not contribute to almost any paths in almost all the applications. This is because the latter seeds have a lower reusability and sensitivity score, leading to fewer similar nodes and hence fewer *sPaths* and *dPaths*.

5.1.2 Seeds, Thresholds, and Similarity Modulo

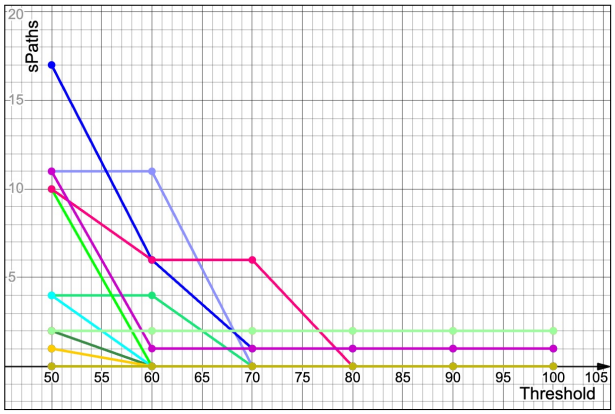
FIXX's three most important parameters are the similarity modulo, the number of seeds we select from the exploit subgraph, and the threshold that determines sPath and dPath similarity. To understand how each of these parameters affects the results, we ran FIXX with different values for these components. In particular, we changed the similarity modulo from 0 (identical instructions) to 2, the number of seeds from 1 to 4, and the threshold from 50% to 100%. We show the results for some combinations of such values in Figure 4 and Figure 5. Due to space reasons, we only include the results for 3 and 4 seeds. In the figure, each line represents one application. As can be seen from the figures, the threshold value significantly affects the number of sPaths and dPaths. The higher the threshold, the fewer paths are discovered. For example, in graph 4(a) which represents the number of *sPaths* detected by FIXX using the top 3 seeds and a similarity modulo value of 0, the purple-pink line represents the application Engineers Online Portal76 which shows that only 3 *sPaths* were discovered when the threshold was set to 50 and 1 *sPath* was found for all higher threshold values. Similarly, the application Engineers Online Portal76 represented by the purple-pink line in graphs



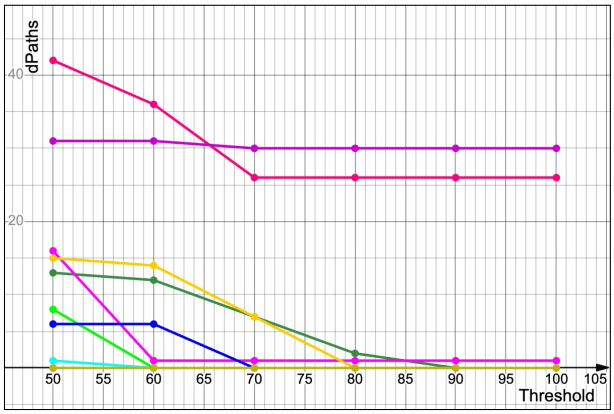
(a) sPaths, 3 seeds, 0 sim modulo



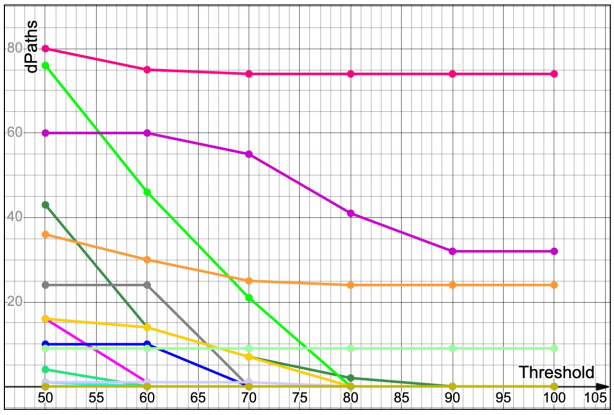
(b) sPaths, 3 seeds, 1 sim modulo



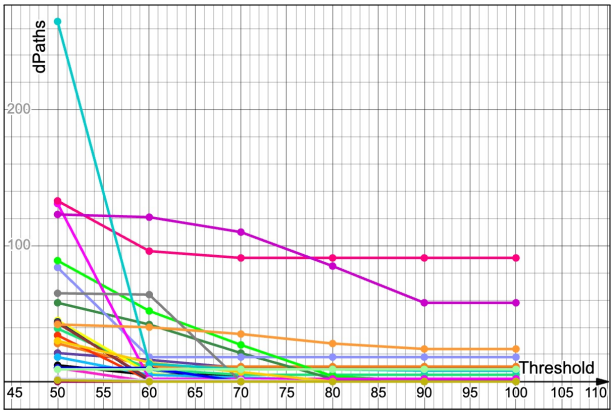
(c) sPaths, 3 seeds, 2 sim modulo



(d) dPaths, 3 seeds, 0 sim modulo



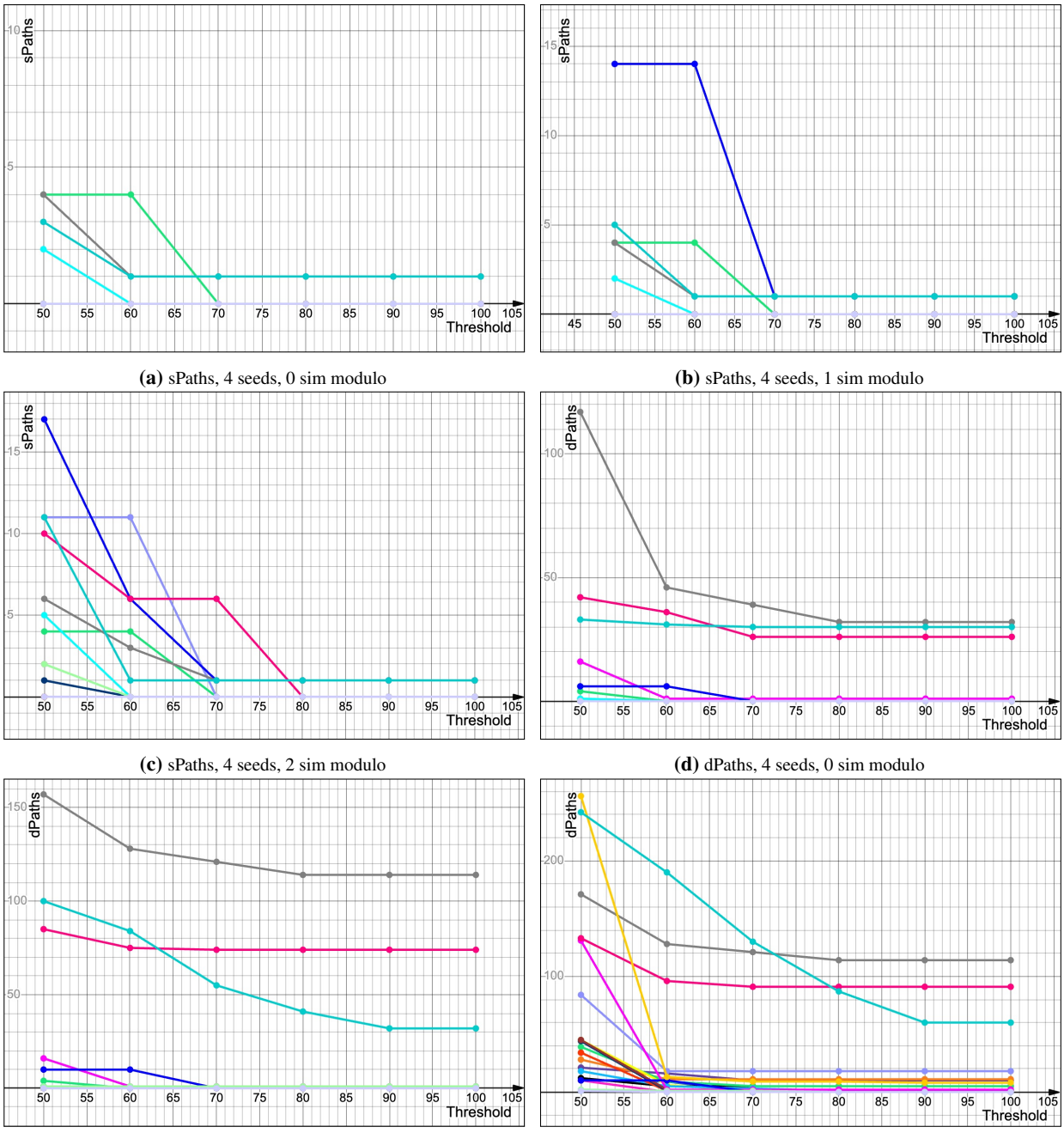
(e) dPaths, 3 seeds, 1 sim modulo



(f) dPaths, 3 seeds, 2 sim modulo

Figure 4: Graphs (a)-(c) represent the sPaths and graphs (d)-(f) represent the dPaths for similarity modulo values 0-2 for different applications using 3 seeds respectively

4(d-f) shows that even though there is an increase in the number of *dPaths* (31 to 60 to 123) by increasing the modulo from 0 to 1 and then to 2, there is always a decrease in the number of *dPaths* as the similarity threshold increases. This shows that the number of *sPaths* and *dPaths* are directly proportional to the similarity modulo value and inversely proportional to the threshold values. We maintain the threshold value at 50% to prevent multiple non-similar *sPaths* and *dPaths* from being counted. However, due to the complexity of a few large applications, it was difficult to obtain many paths that were above the 50% limit. In particular, for the four large applications, SeoPanel, Collabtive, CE Phoenix and Clansphere, shown at the bottom of Figure 6 we had to reduce the threshold even further. Increasing the similarity modulo value also increases the number of sPaths and dPaths. In particular, for the same number of seeds, as the similarity modulo increases,



(a) sPaths, 4 seeds, 0 sim modulo

(b) sPaths, 4 seeds, 1 sim modulo

(c) sPaths, 4 seeds, 2 sim modulo

(d) dPaths, 4 seeds, 0 sim modulo

(e) dPaths, 4 seeds, 1 sim modulo

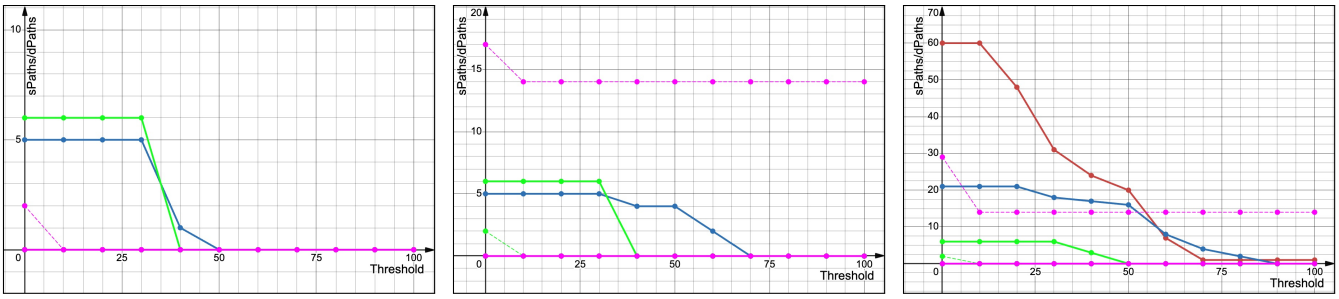
(f) dPaths, 4 seeds, 2 sim modulo

Figure 5: Graphs (a)-(c) represent the sPaths and graphs (d)-(f) represent the dPaths for similarity modulo values 0-2 for different applications using 4 seeds respectively

the height of same colored graph (representing the same application) also generally increases. Finally, as expected, as the number of seeds increases, the number of sPaths and dPaths increases, which can be seen by comparing the respective graphs from Figure 4 and Figure 5.

5.1.3 Evaluation of LLM Techniques

In this section, we give a brief overview of the results of extracting information from CVE descriptions using GPT-4. In order to extract the required details from the corresponding CVE descriptions, we use a fine-tuned version of GPT-4. For all the applications that were evaluated using FIXX, our fine-tuned GPT-4 model extracts the labels of filenames, version numbers, application names, vulnerability types, and various



(a) Large applications, sPaths(dotted)/ dPaths(solid), 4 seeds, 0 sim modulo **(b)** Large applications, sPaths(dotted)/ dPaths(solid), 4 seeds, 1 sim modulo **(c)** Large applications, sPaths(dotted) /d-Paths(solid), 4 seeds, 2 sim modulo

Figure 6: Graphs (a)-(c) represent the sPaths(dotted) and dPaths(solid) for the maximum seeds (3-4) at similarity modulo 2 for the top four large applications

code components, present in the description of the CVEs with perfect precision and recall. Before fine-tuning, the precision and recall were around 95-97%.

5.2 Performance

FIXX's run time performance is shown in Table 4. The experiments were performed on an Asus laptop with 16GB RAM and Intel i7 CPU. For every application that was evaluated using FIXX, column 2 of the table contains the total running times of the Exploit Executioner, the Seed Identification, and the Path Detector modules. As can be seen, the running times are neither proportional to the size of the applications nor the number of PHP lines present in the application and fluctuate significantly. This is because even though the similarity computation depends on the complexity of an application, it also depends on the total number of seeds obtained as well as the number of dataflow paths counted as *sPaths* and *dPaths* respectively. The most time-consuming tasks are the calculation of the sensitivity score of functions wrapping sensitive operations mentioned in Section 3, the computation of every sPath and dPath, and finally, the symbolic execution as well as constraint solving of the similar paths. Finally, the third column combines the time spent by one of the researchers in reproducing the exploits and manually checking the results of the Verifier. Obviously, such times depend on several factors, including ease of reproducibility, understanding of the applications, etc.

5.3 Comparison with Other Tools

In this section, we compare FIXX with two other vulnerability analysis tools, namely RIPS [22] and Navex [12] and a clone detection tool named CloneDr [2]. Unfortunately, other clone detection tools, such as CCFinderX [3], and MOSS [1], do not support PHP.

The comparison results with RIPS and NAVEX are shown in Table 5. Because FIXX is not a 0-day vulnerability discovery approach, we are interested in evaluating the completeness of NAVEX and RIPS and seeing if they detect all the paths detected by FIXX as exploitable. Table 5 contains the number of exploitable paths found by FIXX in the first column. In the

Application	Similarity Time	Manual Effort
SeoPanel02	3.5hr	2hr
Collabtive98	2.5hr	1.5hr
CE Phoenix58	1.5hr	1hr
Clansphere10	57min	45min
Fantasticblog24	51min	30min
FantasticBlog12	59min	30min
FantasticBlog31	1hr	30min
Engineers Online83	39min	1.5hr
Engineers Online76	27min	1hr
Hospital Management11	13min	1hr
Advocate28	13min	30min
Automated Enrollment26	29min	30min
Automated Enrollment94	29min	30min
Tailor60	39min	45min
Tailor73	41min	30min
FruitBazar89	35min	45min
FruitBazar78	35min	45min
Blood Bank27	27min	30min
Blood Bank04	29min	30min
Covid19tms04	13min	35min
Covid19tms03	17min	35min
Dairy Farm93	9min	5min
CodeAstro68	34min	55min
CodeAstro72	37min	55min
CodeAstro33	29min	55min
Loan Management90	10min	30min
Daily Expense06	14min	30min
Daily Expense04	14min	30min
Visitor Management83	21min	25min
Visitor Management60	17min	25min
Visitor Management61	19min	25min
Keerti00	9min	5min

Table 4: Running Times.

second column, it shows the number of exploitable paths that RIPS found out of the total found by FIXX. The third column shows the paths found by NAVEX. As can be seen in the table, using their approach, RIPS and NAVEX are able to find only a subset of the paths found by FIXX, which demonstrates that FIXX can be used in conjunction with these tools to find additional exploitable similar paths.

PHP CloneDr is an AST-based tool that is used to identify and display blocks of code inside an application that are identical or nearly identical to each other to each other [2]. It is based on string-matching operations over the AST. After reading the application's code, CloneDr produces in output a set of HTML pages containing the detected clones organized in clusters. Each cluster is a sequence of one or more lines of code clones of one another. In particular, an AST subtree is

Application	Exp. Paths	Detected by RIPS	Detected by Navex
SeoPanel98	2	0/2	0/2
Collabtive10	9	0/9	2/9
CE Phoenix58	0	0	0
Clansphere10	20	11/20	8/20
FantasticBlog12	9	3/9	2/9
FantasticBlog31	42	30/42	21/42
FantasticBlog24	0	0	0
Engineers Online83	258	229/258	9/258
Engineers Online76	52	49/52	37/52
Hospital Management11	6	5/6	3/6
Advocate28	21	0/21	5/21
Automated Enrollment94	21	14/21	8/21
Automated Enrollment26	0	0	0
Tailor60	86	4/86	51/86
Tailor73	130	17/130	21/130
Fruit Bazar89	4	2/4	1/4
Fruit Bazar78	1	1/1	1/1
Blood Bank27	5	0/5	0/5
Blood Bank04	84	66/84	2/84
Covid19tms04	42	31/42	11/42
Covid19tms03	43	32/43	12/43
Dairy Farm93	33	32/33	6/33
CodeAstro68	38	6/38	0/38
CodeAstro72	44	0/44	0/44
CodeAstro33	17	0/17	0/17
Loan Management90	40	13/40	18/40
Daily Expense06	10	10/10	8/10
Daily Expense04	27	27/27	16/27
Visitor Management83	21	16/21	7/21
Visitor Management60	0	0	0
Visitor Management61	21	16/21	7/21
Keerti00	11	11/11	0/11

Table 5: Comparing the Exploitable Paths Detected by FIXX With RIPS and Navex

considered a clone of another if their string representations are similar above a certain threshold. Consequently, CloneDr cannot detect the similarity among data flow paths whose instructions, in most cases, are not part of the same AST subtree being considered by CloneDr's algorithm. CloneDr does not find any clones in any of the applications discussed in the Evaluation section. In some cases, it is able to find multiple clones of an exploitable dataflow path or of a portion of the path when that path is composed of contiguous instructions.

5.4 Discussion and Limitations

Because FIXX relies to some degree on name overlap, a limitation of FIXX may arise in situations where the components executed during the exploit have fundamentally different names when reused. Currently, FIXX performs exact substring comparisons when trying to match name nodes. If the names of reused components differ by little, FIXX will not be able to match them. We plan to further enhance our similarity comparison by introducing techniques that can perform inexact matching in additional ways (e.g., Locality Sensitive Hashing). Use of LLMs to compare code semantics and identify semantically similar code is another avenue of potential future work.

A second limitation of our implementation may arise when symbolic execution is not precise due to approximations inherent in that technique and, therefore, unable to produce inputs. In these cases, FIXX's verifier may not be able to produce an input string for paths that are actual exploits. We partially address this issue by manually creating inputs and verifying the exploitable paths. A possible avenue for future work includes a more powerful tool for symbolic execution or abstract interpretation of PHP.

6 Related Work

The approach described in this paper has a few intersections with previous work, such as code cloning detection or PHP vulnerability analysis. However, to the best of our knowledge, this is the first work that systematically defines an approach for automatically finding exploits that are similar to a disclosed vulnerability.

PHP Vulnerability Analysis. Several approaches exist for detecting vulnerabilities and exploits in PHP applications [4, 11–13, 16, 20, 22, 23, 32, 36, 39]. Some of these approaches use static analysis techniques such as taint tracking and symbolic execution or dynamic analysis to detect previously unknown vulnerabilities. Others automate the task of finding proof for the vulnerabilities found in the form of exploits. However, they look for zero-day vulnerabilities and exploits and are not tuned to reuse knowledge produced by human testers and developers. As we mention in the introduction such knowledge, even though it is often vague and inconsistent, can provide great insight into the security and inner design of applications. FIXX tries to reuse such knowledge.

Detecting code clones. One area of research in the field of software engineering deals with the detection of code clones [28, 30, 38, 41, 44]. A large portion of these works considers code as text and deals with the problem using text similarity detection techniques. Some of these works use different approaches to detect code clones based on program dependency graph similarity [30, 38, 44] or on abstract syntax tree similarity [28]. However, none of these combine both PDG and AST techniques to determine similar dataflows. Among these works, the closest work to FIXX, with respect to the techniques used to detect similar dataflows, is [30]. However, they search for identical dataflows and do not consider cases where similar dataflows may be different subsequences in larger dataflows. CloneWorks performs code similarity detection based on the Jaccard similarity of sets of words appearing in the code fragments that are compared. While performant, this method does not consider type and path similarity [42]. Another approach that uses PDG-based techniques is implemented for Android applications. This approach, however, works on bytecode and is not aimed at detecting vulnerabilities [18]. Another PDG-based approach for detecting clones is [27]. This approach transforms PDG-s into AST forests to detect isomorphic PDGs. However, this approach is not aimed at detecting similar vulnerabilities and it is not developed for PHP.

Code property graphs. Work based on detecting vulnerabilities using code property graphs includes [12, 14, 47]. These works rely on using graph representations of code to detect vulnerabilities as code patterns on the graphs. Some of these

works target PHP applications as well. However, again, they focus on zero-day vulnerability detection and do not incorporate elements from exploitable traces in their approach. Another related work that uses code property graphs is [43], which searches for code from flawed tutorials in PHP applications. However, this work looks mainly at syntactic similarity to identify vulnerabilities, without delving into dataflow similarity and automated exploit generation.

7 Conclusion

In this paper, we present FIXX, one of the first approaches for detecting new vulnerable paths similar to a known vulnerable path in a given application. We also demonstrated that FIXX, being the first of its kind, successfully identifies 1097 similar exploitable paths to 32 original CVEs spanning 19 PHP applications resulting in the submission of 10 new CVEs.

8 Ethics Consideration

In this section, we discuss the ethical considerations and compliance with the open science policy.

8.1 Vulnerability Threat Reports

The research in this paper has been done ethically. Our approach FIXX aims to find all exploitable instances of a known vulnerability in an application. Once the paths obtained have been executed and the corresponding vulnerabilities have been found, we report the exploit to the corresponding vendor. This step is done to inform the vendor of either an XSS or SQL Injection vulnerability present in the corresponding version of their application so that the vulnerability may be patched.

We compose an email consisting of the type of vulnerability (XSS in the case of our published CVEs), the location of the vulnerable code in the application as well as Proof of Concepts of how the vulnerability can be triggered. We also provide remediations to help the vendors patch the exploit in future versions. We repeat this process for all the applications that FIXX discovered exploitable paths in, highlighting each path separately.

We also report the vulnerabilities found to MITRE Corporation by requesting for a CVE-ID. Since we are interested in showcasing the different exploitable paths discovered by FIXX, we distinguish paths and submit multiple CVEs for a given application. We have created Github repositories for each of the applications and provided more details regarding each of the vulnerabilities that have been submitted to the vendor and MITRE, respectively. At the time of writing this paper, we have received confirmations for 10 published CVEs. We will continue to submit additional CVEs for the remaining exploitable paths found.

8.2 Artifacts

The artifacts responsible for obtaining the results throughout our paper can be evaluated for their availability, functionality, and reproducibility. We have included the source code and the applications selected as part of our evaluation. In addition, we have included the dataset used to fine-tune the GPT4 model discussed in the LLM section of the paper. Finally, we have also disclosed all the published CVEs IDS with details on how the vulnerabilities can be reproduced. The link to our artifact: https://zenodo.org/records/14738532

9 Acknowledgments

This work was funded by Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006, by the US Department of Defense/National Defense Education Program through the Educational and Research Training Collaborative at the University of Illinois Chicago, award HQ00342010037, by NSF awards 1918542 and 2330565, and UKRI, UK award no. EP/Y026233/1. We would like to also thank the following undergraduate students for their help: Ahmad Alawi, Jamiel Impoy, and Khizer Shareef.

References

[1] A System for Detecting Software Similarity. https://theory.stanford.edu/ aiken/moss/.

[2] Clone Doctor: Software Clone Detection and Reporting. http://www.semdesigns.com/products/clone.

[3] the archive of CCFinder Official Site. http://www.ccfinder.net/ccfinderxos.html.

[4] Rips. http://rips-scanner.sourceforge.net/, 2013. Accessed: 2020-10-13.

[5] Phpjoern. https://github.com/malteskoruppa/phpjoern, 2014. Accessed: 2020-10-14.

[6] The neo4j graph platform – the #1 platform for connected data. https://neo4j.com/, 2018. Accessed: 2020-10-15.

[7] Cypher. https://neo4j.com/developer/cypher/, 2020. Accessed: 2020-10-14.

[8] Déjà vu-lnerability. https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html, 2021. Accessed: 2021-11-04.

[9] Gpt4-technical report. https://cdn.openai.com/papers/gpt-4.pdf, 2024. Accessed: 2024-12-27.

[10] Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., and Maqbool, B. A systematic review on code clone detection. *IEEE Access 7* (2019), 86121–86144.

[11] Alhuzali, A., Eshete, B., Gjomemo, R., and Venkatakrishnan, V. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), ACM, pp. 641–652.

[12] Alhuzali, A., Gjomemo, R., Eshete, B., and Venkatakrishnan, V. N. NAVEX: precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018* (2018), W. Enck and A. P. Felt, Eds., USENIX Association, pp. 377–392.

[13] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng. 36*, 4 (July 2010), 474–494.

[14] Backes, M., Rieck, K., Skoruppa, M., Stock, B., and Yamaguchi, F. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)* (2017), IEEE, pp. 334–349.

[15] Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, Association for Computing Machinery, p. 25–35.

[16] Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. Multi-module Vulnerability Analysis of Web-based Applications. In *the 14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.

[17] Chaparro, O., Lu, J., Zampetti, F., Moreno, L., Di Penta, M., Marcus, A., Bavota, G., and Ng, V. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting*

on Foundations of Software Engineering (New York, NY, USA, 2017), ESEC/FSE 2017, Association for Computing Machinery, p. 396–407.

[18] CHEN, K., LIU, P., AND ZHANG, Y. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, Association for Computing Machinery, p. 175–186.

[19] CRUSSELL, J., GIBLER, C., AND CHEN, H. Andarwin: Scalable detection of android application clones based on semantics. *IEEE Transactions on Mobile Computing 14*, 10 (2015), 2007–2019.

[20] DAHSE, J. *Static detection of complex vulnerabilities in modern PHP applications*. PhD thesis, Ruhr University Bochum, 2016.

[21] DAHSE, J., AND HOLZ, T. Simulation of built-in php features for precise static code analysis.

[22] DAHSE, J., AND HOLZ, T. Simulation of built-in PHP features for precise static code analysis. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014), The Internet Society.

[23] DAHSE, J., AND HOLZ, T. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (USA, 2014), SEC'14, USENIX Association, p. 989–1003.

[24] DE MOURA, L. M., AND BJØRNER, N. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.

[25] DONG, Y., GUO, W., CHEN, Y., XING, X., ZHANG, Y., AND WANG, G. Towards the detection of inconsistencies in public security vulnerability reports. In *Proceedings of the 28th USENIX Conference on Security Symposium* (USA, 2019), SEC'19, USENIX Association, p. 869–885.

[26] DOUPÉ, A., BOE, B., KRUEGEL, C., AND VIGNA, G. Fear the ear: discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, Association for Computing Machinery, p. 251–262.

[27] GABEL, M., JIANG, L., AND SU, Z. Scalable detection of semantic clones. In *2008 ACM/IEEE 30th International Conference on Software Engineering* (2008), pp. 321–330.

[28] JIANG, L., MISHERGHI, G., SU, Z., AND GLONDU, S. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)* (2007), pp. 96–105.

[29] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (SP'06)* (2006), pp. 6 pp.–263.

[30] KAMALPRIYA, C. M., AND SINGH, P. Enhancing program dependency graph based clone detection using approximate subgraph matching. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)* (2017), pp. 1–7.

[31] KANG, M., XU, Y., LI, S., GJOMEMO, R., HOU, J., VENKATAKRISHNAN, V. N., AND CAO, Y. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), pp. 1059–1076.

[32] KIEYZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Ardilla: Automatic Creation of SQL Injection and Cross-site Scripting Attacks. http://groups.csail.mit.edu/pag/ardilla/.

[33] KIEYZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering* (2009), pp. 199–209.

[34] LUO, C., LI, P., AND MENG, W. Tchecker: Precise static interprocedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2022), CCS '22, Association for Computing Machinery, p. 2175–2188.

[35] MARTIN, M., AND LAM, M. S. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th Conference on Security Symposium* (USA, 2008), SS'08, USENIX Association, p. 31–43.

[36] NADERI-AFOOSHTEH, A., KWON, Y., NGUYEN-TUONG, A., RAZMJOO-QALAEI, A., ZAMIRI-GOURABI, M.-R., AND DAVIDSON, J. W. Malmax: Multi-aspect execution for automated dynamic web server malware analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 1849–1866.

[37] RAGKHITWETSAGUL, C., KRINKE, J., AND CLARK, D. A comparison of code similarity analysers. *Empirical Software Engineering 23*, 4 (Aug 2018), 2464–2519.

[38] SABI, Y., HIGO, Y., AND KUSUMOTO, S. Rearranging the order of program statements for code clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)* (2017), pp. 1–7.

[39] SADYRIN, D., DERGACHEV, A., LOGINOV, I., KORENKOV, I., AND ILINA, A. Application of graph databases for static code analysis of web-applications.

[40] SARGSYAN, S., KURMANGALEEV, S., BELEVANTSEV, A., AND AVETISYAN, A. Scalable and accurate detection of code clones. *Programming and Computer Software 42*, 1 (2016), 27–33.

[41] SUDHAMANI, M., AND RANGARAJAN, L. Structural similarity detection using structure of control statements. In *Procedia Computer Science* (2015), vol. 46, pp. 892–899.

[42] SVAJLENKO, J., AND ROY, C. K. Cloneworks: A fast and flexible large-scale near-miss clone detection tool. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (2017), pp. 177–179.

[43] UNRUH, T., SHASTRY, B., SKORUPPA, M., MAGGI, F., RIECK, K., SEIFERT, J., AND YAMAGUCHI, F. Leveraging flawed tutorials for seeding large-scale web vulnerability discovery. *CoRR abs/1704.02786* (2017).

[44] WANG, M., WANG, P., AND XU, Y. Ccsharp: An efficient three-phase code clone detector using modified pdgs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (2017), pp. 100–109.

[45] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not. 42*, 6 (June 2007), 32–41.

[46] WASSERMANN, G., YU, D., CHANDER, A., DHURJATI, D., INAMURA, H., AND SU, Z. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, Association for Computing Machinery, p. 249–260.

[47] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 590–604.

[48] YU, F., ALKHALAF, M., AND BULTAN, T. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2010), pp. 154–157.

[49] ZHANG, H., CHEN, W., HAO, Y., LI, G., ZHAI, Y., ZOU, X., AND QIAN, Z. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021), CCS '21, Association for Computing Machinery, p. 811–824.

A Appendix

In table 6, we show the colors associated with each application.

Application - CVEID	Color		Application - CVEID	Color
SeoPanel02			FantasticBlog12	
Collabtive98			FantasticBlog31	
CE Phoenix58			FantaticBlog24	
Clansphere10			Engineers Online Portal83	
Fruit Bazar78			Blood Bank System27	
Covid19tms03			Dairy Farm93	
CodeAstro33			Loan Management90	
Visitor Management83			Visitor Management60	
Engineers Online Portal76			Automated Enrollment26	
Hospital Management11			Tailor60	
Advocate Office28			Tailor73	
Automated Enrollment94			Fruit Bazar89	
Blood Bank System04			Covid19tms04	
CodeAstro68			CodeAstro72	
Daily Expense06			Daily Expense04	
Visitor Management61			Keerti00	

Table 6: Color Representations For The Applications Shown In The Graphs