2021

Detecting Server-Side Web Applications with Unrestricted File Upload Vulnerabilities

Jin Huang
Wright State University

Detecting Server-Side Web Applications with Unrestricted File Upload Vulnerabilities

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

by

Jin Huang
M.S., Wright State University, 2018
M.S., University of Florida, 2014
B.E., Beijing Technology and Business University, 2010

2021
WRIGHT STATE UNIVERSITY

WRIGHT STATE UNIVERSITY
GRADUATE SCHOOL

July 26, 2021

I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY Jin Huang ENTITLED Detecting Server-Side Web Applications with Unrestricted File Upload Vulnerabilities BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

Junjie Zhang, Ph.D.
Dissertation Director

Yong Pei, Ph.D.
Director, Computer Science and
Engineering Ph.D. Program

Barry Milligan, Ph.D.
Vice Provost for Academic Affairs
Dean of the Graduate School

Committee on
Final Examination

Junjie Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Michelle Andreen Cheatham, Ph.D.

Phu H. Phung, Ph.D.

ABSTRACT

Huang, Jin. Ph.D., Department of Computer Science and Engineering, Wright State University, 2021. Detecting Server-Side Web Applications with Unrestricted File Upload Vulnerabilities

Vulnerable web applications fundamentally undermine website security as they often expose critical infrastructures and sensitive information behind them to potential risks and threats. Web applications with unrestricted file upload vulnerabilities allow attackers to upload a file with malicious code, which can be later executed on the server by attackers to enable various attacks such as information exfiltration, spamming, phishing, and spreading malware. This dissertation presents our research in building two novel frameworks to detect server-side applications vulnerable to unrestricted file uploading attacks. We design the innovative model that holistically characterizes both data and control flows using a graph-based data structure. Such a model makes effortless critical program analysis mechanisms, such as static analysis and constraint modeling. We build the interpreter to model a web program by symbolically interpreting its abstract syntax tree (AST). Our research has led to three complementary systems that can effectively detect unrestricted file uploading vulnerabilities. The first system, namely *UChecker*, leverages satisfiability modulo theory to perform detection, whereas the second system, namely *UFuzzer*, detects such vulnerability by intelligently synthesizing code snippets and performing fuzzing. We also proposed the third system to mitigate the challenge of path explosion that the previous two systems suffered and enable a computationally efficient model generation process for large programs. We have deployed all of our systems, namely *UGraph*, to scan many server-side applications. They have identified 49 vulnerable PHP-based web applications that are previously unknown, including 11 CVEs.

Contents

List of Figures

List of Tables

Acknowledgment

I want to express the deepest gratitude to my advisor Dr. Junjie Zhang for the continuous support of my Ph.D. study and research, for his patience, encouragement, and immense knowledge. He continually and convincingly conveyed his brilliant idea and design regarding research and scholarship and enthusiasm regarding teaching. Without his guidance and persistent advice, this dissertation would not have been possible.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Krishnaprasad Thirunarayan, Dr. Michelle Andreen Cheatham, and Dr. Phu H. Phung, for their encouragement, insightful comments, and outstanding questions.

In addition, a grateful thank you to Dr. Mateen M. Rizki, who introduced me to "Data Structure and Algorithms", and thanks for his patient initiative and guidance. Another grateful thank you to Dr. Jack S. Jean, who introduced me to the "Computer Organization", and whose concise and understandable lectures majority helped me build the fundamentals knowledge. Also, I thank Dr. Yu Li and Jialun Liu for their cooperation and Chuang Li for his assistance with my research.

Last but not least, I would like to thank all my family and friends for their love and support through this challenging period. Without their help, this wouldn't have been possible.

Chapter 1: Introduction

Web-based applications have become the de facto standard for delivering many online services ranging from content sharing (e.g., blogging) to entertainment and financial services. However, cyber-attacks have become of a fundamental concern of these applications, resulting in enormous losses. The vast majority of these attacks, if not all, can be attributed to software vulnerabilities, software flaws that can be exploited by attackers for malicious intents. Detecting vulnerabilities there becomes a central task.

Among various vulnerabilities, those leading to remote code execution (a.k.a, RCE) are widely ranked with the highest risk. Specifically, RCE vulnerabilities render attackers the capability to execute programs remotely at the target system. The unrestricted file upload (UFU) vulnerability is an RCE vulnerability. It allows an attacker to upload an executable file to the target web server and later execute it. Such vulnerabilities are particularly significant for server-side scripts (e.g., those with extensions such as ".php", ".exe", and ".js") that are treated as automatically executable without any file system permissions. Therefore, the UFU vulnerability has been considered a top web vulnerability by OWASP; it has also been recognized as one of the most common vulnerability types for WordPress, a leading PHP-based open-source content management system(CMS). However, it is a challenging task to detect UFU vulnerabilities. First, the vulnerable implementation is usually embedded in server-side web applications whose source code is typically large in size and complicated in logic. Second, developers have extremely high flexibility for software implementation, but the analysts lack an effective, unified program representation to support

effective analysis. Third, it takes significant efforts to operate server-side applications, commonly needing to be distributed running environments supported by various networking and database configurations.

This dissertation focuses on building a framework to detect UFU vulnerabilities by systematically overcoming these challenges. Figure 1.1 presents the architectural overview of the framework. Specifically, it consists of four major innovations

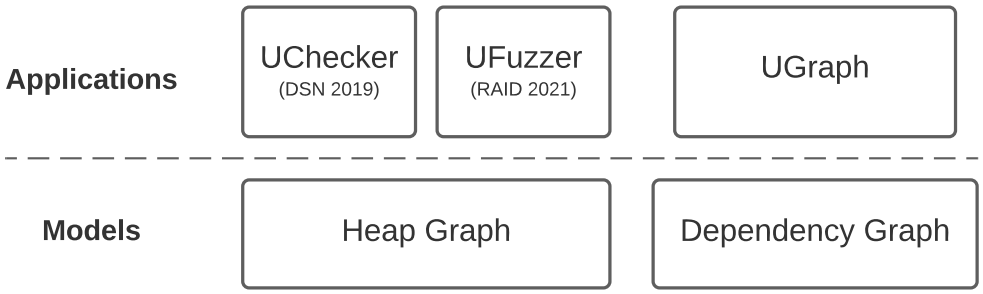

Figure 1.1: The architectural Overview of Framework

- **Model - The Heap Graph [22]**: A novel program representation, namely the heap graph, has been designed to holistically model the control flow and data flow of a web program using a graph-based data structure.

- **Application 1 - *UChecker* [22]**: By traversing the heap graph of a web application, *UChecker* models the condition to exploit a UFU vulnerability using symbolic constraints then leverages SMT solver to evaluate these constraints.

- **Application 2 - *UFuzzer* [23]**: By analyzing the heap graph of a web application, *UFuzzer* generates executable code snippet that models the exploitation of the vulnerability and execute this code snippet in a local, native run-time environment.

Experimental results using a large number of PHP web applications have demonstrated the efficiency and effectiveness of the model and these applications. Specifically,

they have detected 34 vulnerable PHP-based web applications that are previously unknown, including 5 CVEs.

One limitation of the Heap Graph, and consequently its applications, is the limited scalability. Specifically, the model leverages symbolic interpretation, which inherently suffers from the path explosion challenge. Therefore, we mitigate path explosion by designing a new model, possibly at the cost of lowering its precision. We also propose to build a detection system based on the new model.

- **Model - Dependency Graph**: A novel program representation, namely the dependency graph, has been designed to model the immediate data and control dependency among objects. An object refers to the evaluation result of an expression in a program.

- **Application 3 - *UGraph***: By querying the dependency pattern of a web application, *UGraph* loads the dependency model to graph-database and detects the UFU vulnerability leveraging designed queries.

The rest of this dissertation is organized as follows:

- In Chapter 2, we introduce the relative background knowledge and related works.

- In Chapter 3, We start with an introduction to heap graph and illustrate the definition of all elements defined in heap graph and operations of heap graph when performed the AST-based symbolic execution.

- In chapter 4, We built an automatic system to detect PHP-based web programs with unrestricted file upload vulnerabilities named *UChecker* that interprets abstract syntax trees of PHP source code for symbolic execution, whose performance is improved by a novel vulnerability-oriented locality analysis algorithm. We model vulnerabilities using constraints and verify them using a satisfiability modulo theories (SMT)

solver. Experiments have demonstrated that *UChecker* detected 3 vulnerable Wordpress Plugins that have not been publicly reported.

- In Chapter 5, We proposed an automatic detection system for PHP-based web programs that suffered the unrestricted file upload vulnerabilities call it *UFuzzer*. *UFuzzer* models a server-side PHP web application using heap graphs and automatically identifies sub-graphs that are relevant to a vulnerability. Identified sub-graphs are refactored and eventually converted into executable PHP programs for fuzzing. The evaluation results based on real-world PHP applications demonstrated *UFuzzer*'s high detection performance. *UFuzzer* detected 34 unreported vulnerable web applications and got 5 CVE numbers.

- In Chapter 6, We established an automatic detection system for the PHP-based web application with unrestricted file upload vulnerabilities. *UGraph* models a PHP program using dependency graphs and leverages the graph database to load all objects and detects the vulnerability by querying the dependency pattern. *UGraph* significantly improved the scalability of the detecting system than the previous two and detected 14 unreported vulnerable web applications, 6 of which were validated by the CVE Numbering Authority.

- In Chapter 7, we conclude our work.

Chapter 2: Background and Related Work

2.1 Background

Figure 2.1 presents an example of a server script with an unrestricted file upload vulnerability. In this example, the web server responds the client with a webpage for image uploading. Its sample source code is shown in ① and its presentation to the client user is presented in ②. The client (an attacker), instead of uploading an image, uploads a PHP file named `UnrestrictedFileUpload.php` as displayed in ③, whose source code is also manifested. The server side program saves the uploaded file (using `move_uploaded_file(`e_{src}, e_{dst}`)`) to the local directory without validating the extension of the uploaded file (see ④). Later, the attacker accesses the uploaded file as presented in ⑤. Since this uploaded file has "`.php`" as extension, it will be executed by server. Specifically, "PHP executed!" is the execution result of the uploaded script named `UnrestrictedFileUpload.php`. *One root cause of this vulnerability is that it does not check the extension of the file to be permanently saved. Therefore, executables files (i.e., those with "`.php`" extensions) can be uploaded.*

The file uploading function is usually implemented using the "file" input type with a particular name assigned in the script from server to the client (i.e., "userpic" in this case), as shown in ①. When the file is transmitted to the server, the server retrieves client-offered

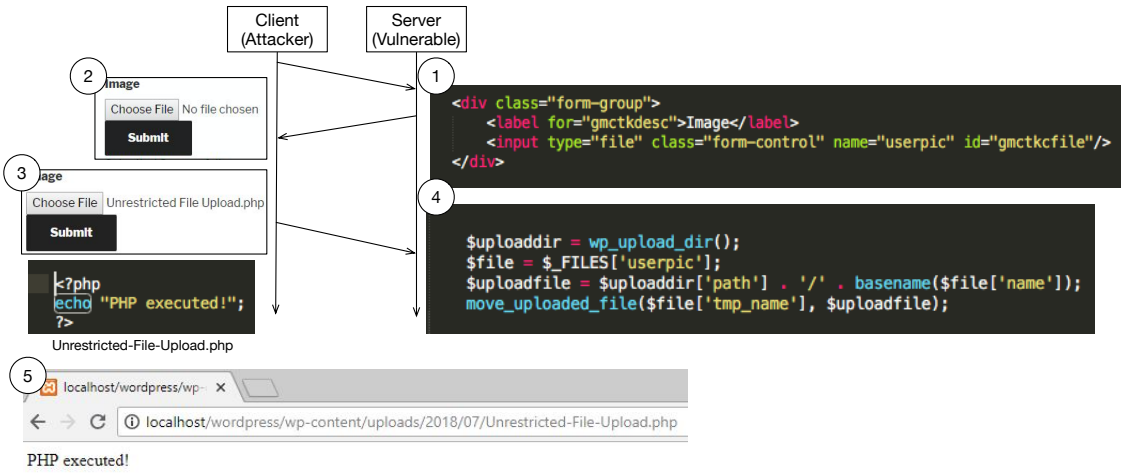

Figure 2.1: A typical scenario of unrestricted file upload vulnerability (Reprinted from [22])

information such as the original file name and the file type. It also identifies the possible transmission error and calculates the file size. The server saves this file in the local file system using a temporal name. Such information is stored in a built-in superglobal variable, namely $_FILES, which is automatically enabled when the "file" input type is used. Specifically, $_FILES can be considered as a two-dimensional array (i.e., $_FILES[i][j]), where both indices are strings. The first index refers to the file name; accessing $_FILES using the first index returns an array with a pre-structured array. For example, $_FILES['userpic'] returns such array for the file submitted through "userpic". The second index refers to properties of this file, such as the original file name, the type information, the temporal filename, the error information, and the size of the file, which are indexed by "name", "type", "tmp_name", "error", or "size", respectively.

As indicated in ④ of Figure 2.1, $file refers to the pre-structured array for the file "userpic". A path is then created to store the file, which is composed of the directory (i.e., $uploaddir['path']) and the original filename (i.e., basename($file['name'])). Specifically, basename($file['name']) returns `Unrestricted-File-Upload.php`. As indicated by the function name, "move_uploaded_file($file['tmp_name'], $uploadfile)" moves the uploaded PHP script to a directory and name it as `Unrestricted-File-Upload.php`.

Since its extension is ".php", `Unrestricted-File-Upload.php` will be executed when it is requested. Other than `move_uploaded_file(`e_{src}, e_{dst}`)`, another function, namely `file_put_content(`e_{dst}, e_{src}`)`, is also commonly used to save an uploaded file.

2.2 Related Work

Detection vulnerability in web applications is a broad area that has fostered a wide range of active research efforts. We explore related work on static program analysis and detecting unrestricted file upload vulnerabilities.

Static Program Analysis is a method of detection done by evaluating an application's source code before deployment. It has been approved and adopted to detect a variety of vulnerabilities by the following efforts [54, 21, 60, 46, 15, 9, 44, 50, 14]. Zheng et. al [54, 21] and Xie et.al [60] leverage static program analysis to detect PHP web applications that are vulnerable to SQL injection and XSS attacks. Son et. al [46] proposed a method to identify PHP web applications with semantic vulnerabilities such as infinite loops and the missing of authorization checks. Dahse et. al [15] designed a system to detect SQL injections and XSS using data flow analysis. Barth et. al [9] designed a system to detect XSS attacks by analyzing the structure of the content submitted to the server. These approaches provide a solid basis for vulnerability detection in web application but do not consider the unrestricted file upload vulnerability, which is also one of the critical web vulnerabilities. Staicu et. al [50] studied Node.js applications vulnerable to injection attacks that exploit `exec` or `eval` APIs. Similar to our framework, this method also interprets the AST of a web application for analysis. However, it uses template matching rather than symbolic execution to detect vulnerabilities. In addition, it targets detecting different vulnerabilities. Dahse et. al [14] proposed novel block and function summaries to detect taint-style vulnerabilities. Unfortunately, taint-analysis alone is insufficient to model unrestricted file uploading vulnerabilities, thereby likely introducing false positives. Samimi

et. al [43] designed a system to automatically repair HTML generation errors in PHP applications. This system also used string constraint solving technique. However, this system addresses a different problem. Nunes et. al [31] conducted benchmark-based assessments to compare capabilities of publicly-available static vulnerability detection tools. A few other methods [55, 8, 33, 19, 26, 7, 53] detect employ fuzzing or penetration testing to reveal vulnerabilities and failures in server-side web applications. Some of them focus on generating fuzzing inputs [55, 19, 7]. A few other tools such as JBroFuzz [47], Wapiti [48], Wfuzz [24], Burp [36], and w3af [52] are also built to fuzzing HTTP request through interception. While these methods have shown great promise in detecting server-side web vulnerabilities, they focus on conventional ones other than unrestricted file upload vulnerabilities.

A few existing projects [51, 6, 40, 15, 11] analyzed unrestricted file upload vulnerabilities without delivering detection capabilities. RIPS [16, 14] and WAP [28] claim their capabilities of detecting unrestricted file upload vulnerabilities. RIPS [16, 14] leverages static taint analysis while WAP[28] combines taint analysis and machine learning. RIPS and WAP are prone to low detection performance of this specific type of vulnerabilities since taint-analysis is over-approximate to model the exploitation. In addition, the learning-based software defect detection commonly suffers from the data sparsity challenge [5], where program samples with the same type of vulnerabilities usually fall short for both quantity and diversity.

FUSE [25] leverages an orthogonal strategy, i.e., black-box fuzzing, to detect such vulnerabilities. It attempts to upload various executable files to a fully operating web service and monitor whether the uploading is successful. While *FUSE* can report concrete inputs for exploitation, it faces significant practical challenges. First, it mandates operating web services, which are labor-intensive for deployment and maintenance. Second, web services commonly offer a large number of access points, whose expected external inputs experience a high diversity in both structures and formats. It is extremely challenging to

extensively address such input diversity without a priori knowledge. In fact, *FUSE* needs a manually pre-specified configuration template file that manifests a variety of parameters. Both challenges fundamentally limit *FUSE*'s applicability in large-scale analysis. Finally, *FUSE* cannot locate statements that cause the vulnerability, offering limited information for mitigation.

Chapter 3: Heap Graph

A PHP-based web application is performed symbolic execution by statically interpreting its AST(s), ultimately generating graph-based data structures, namely *a heap graph* and *environments*. The heap graph compactly profiles dependency among all possible objects produced by all execution paths; each environment maps variables to their corresponding objects in each path and meanwhile keeps track of path constraints.

3.1 Heap Graph Definition

A heap graph is a graph-based IR that models symbolic execution results of a program along all paths towards a given statement (or the end of the program if the given statement is not observed in this path). *A heap graph* has following essential elements:

- Node: A node in a *heap graph* refers to the evaluation result of an expression, which could represent a concrete value, a symbolic value, an operator, or a built-in function (e.g., an API). Since we interpret AST to generate nodes, each node can be precisely mapped back to the program source code.
- Edge: An edge (u, v) represents the operator-operand relationship when u denotes an operator; it represents the function-parameter relationship when u refers to a function.

- Environment: An *environment* is maintained for each execution path. It records the reachability constraint (named as cur) for its corresponding each execution path towards that given statement; it also maps variables to their corresponding objects in each path and meanwhile keeps track of path constraints; if a vulnerability-related API appears in this path, it uses a special variable, namely API, to track the node of that API.

We define the node and edge in a *heap graph* as $G = \{C,\ S,\ FUNC,\ OP,\ L,\ T,\ O_C,\ O_S,\ O_{FUNC},\ O_{OP},\ Edge\}$:

- C is a set of concrete values.
- S is a set of symbolic values.
- $FUNC$ is a set of all PHP built-in functions.
- OP is a set of all operations (e.g., unary and binary operations such as "+" or ".").
- L is a set of labels.
- T is a set of types such as boolean, integer, and etc; T also includes an unknown type $\perp$ (i.e., $\perp \in T$) and an array type (i.e., $array \in T$).
- $O_C \subset C \times T \times L$ is a set of objects (i.e., nodes) for concrete values, where each object in O_C is assigned with a type and a *unique* label.
- $O_S \subset S \times T \times L$ is a set of objects (i.e., nodes) for symbolic values, where each object in O_C is assigned with a type and a *unique* label.
- $O_{FUNC} \subset Func \times T \times L$ is a set of objects (i.e., nodes) for built-in functions, where each node is assigned with a type and a *unique* label. The type indicate the type of the result returned by the function.

- $O_{OP} \subset Op \times T \times L$ is a set of objects (i.e., nodes) for operations, where each node is assigned with a type and a *unique* label. The type indicate the type of the result returned by the operation.

- $Edge \subset \{(l_1, l_2) | (x, t_1, l_1) \in O_{FUNC} \cup O_{OP} \; and \; (y, t_2, l_2) \in O_C \cup O_S \cup O_{FUNC} \cup O_{OP}\}$. Edges are directed and each one connects a node for a built-in function or an operation with another node with an arbitrary type. If the source node of an edge is an object of an operand, its destination node is an operator; if the source node of an edge is for a built-in function, its destination node is a parameter input for this function.

We define the environment for each path $Env = \{Var, Map, cur\}$, which characterizes i) the mapping between a variable name and its object and ii) the reachability constraint for this path. It is worth noting that a program may have multiple paths and each path has its own environment. We therefore define $\mathcal{E} = \{Env_1, \ldots, Env_i, \ldots, Env_n\}$ for n execution paths of a program.

- Var is a set of variable names.

- $Map \subset Var \times L$. It establishes a mapping between a variable name and an object.

- $cur \in \{l | (x, t, l) \in O_C \cup O_S \cup O_{FUNC} \cup O_{OP}\} \cup \{null\}$. cur represents the reachability constraint. It either points to nothing (e.g., $cur = null$) or an object. When $cur \neq null$, the reachability constraint has to be true to enable the execution of this path.

In order to illustrate the heap graph and environments, we use an example presented in Listing 3.2. This program has two variables including `$a` and `$b`. `$a` is initialized with a concrete, integer value. `$b` contains value from an external input, thereby being given a symbolic value. This program has two paths which are governed by the `if` condition and result in different values for `$a`.

```php
<?php
    $a = 55;
    $b =$_GET['number'];
    if($a + $b > 10)
        $a = $b - 22;
    else
        $a = 88;
?>
```

Listing 3.1: Sample code with two paths(Reprinted from [22])

```php
<?php
function is_Vulnerable_0(/*...*/ string $_POST_doadd_symbol)
{
    $exp_reach = ($_POST_doadd_symbol == 'yes' and /*...*/);
    $funCall = explode('.', $sym_file_name . $sym_file_ext);
    $exp_filename = wp_upload_dir(). /*...*/ . end($funCall);
    if ($exp_reach) {
        $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
        if ($ext == 'php') {
            return true;
        }
    }
    return false;
}
?>
```

Listing 3.2: Sample code with two paths(Reprinted from [22])

Figure 3.1 presents the heap graph and path environments that interpreter generates for the example in Listing 3.2. For this specific example, the heap graph G is:

- $C = \{55, 10, 22, 88\}$
- $S = \{s\}$
- $FUNC = \emptyset$
- $OP = \{+, -, >, \mathsf{NOT}\}$
- $L = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

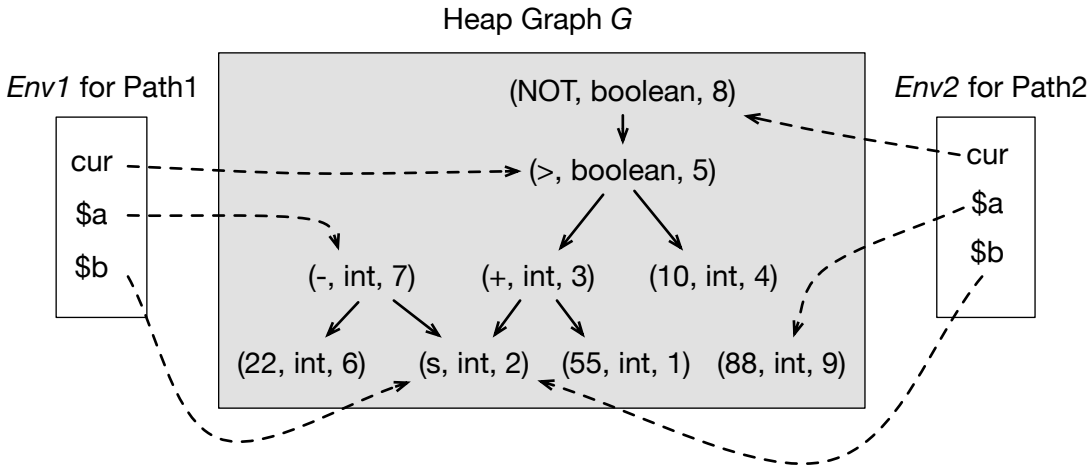

Figure 3.1: The heap graph for the sample code in Listing 3.2 (Reprinted from [22])

- $T = \{\texttt{boolean}, \texttt{int}\}$
- $O_C = \{(55, \texttt{int}, 1), (10, \texttt{int}, 4), (22, \texttt{int}, 6), (88, \texttt{int}, 9)\}$
- $O_S = \{(s, \texttt{int}, 2)\}$
- $O_{FUNC} = \emptyset$
- $O_{OP} = \{(+, \texttt{int}, 3), (>, \texttt{boolean}, 5), (-, \texttt{int}, 7), (\texttt{NOT}, \texttt{boolean}, 8)\}$
- $Edge = \{(7,6), (7,2), (3,2), (3,1), (5,3), (5,4), (8,5)\}$.

To be more specific, we label each object using a distinct integer (i.e., $L = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$). This program has two paths. The completion of two paths will result two environments $\mathcal{E} = \{Env_1, Env_2\}$. For Env_1, $Var = \{a, b\}$, $Map = \{(a, 7).(b, 2)\}$, $cur = 5$; for Env_2, $Var = \{a, b\}$, $Map = \{(a, 9).(b, 2)\}$, $cur = 8$. For example, $(a, 7) \in Map$ of Env_1 means that the value of a for the first path is the result of the object with label 7 (i.e., $(-, \texttt{int}, 7)$). The reachability constraint for the first path is $cur = 5$, pointing to the object of $(>, \texttt{boolean}, 5)$, which has to be satisfied to enable the execution of this path.

As manifested in this example, our design of the heap graph and environments introduces two advantages. First, the tree-like structure of the heap graph enables the s-expression-based representation of an object value using concrete and/or symbolic values. For example, by traversing the heap graph in Figure 6.2, the reachability constraint of path 1 (i.e., the node of $(>, \texttt{boolean}, 5)$) can be expressed using symbolic or constant values in the form of s-expressions, which is specifically $(> \ (+ \ s \ 55) \ 10)$. This facilitates the usage of SMT solvers, such as Z3 [17] and Yices [20], whose rules are expressed in s-expressions. Second, environments keep track of object labels for variables. Therefore, many objects can be shared by different environments, thereby reducing the memory consumption.

3.2 Operations for Heap Graph

We next define a set of operations for G, Env, and $\mathcal{E}$.

$Find(G, l)$ returns an object given its label. If there is no object whose label is l, it will return $null$.

$Create\_Concrete\_Obj(x, t)$ is to create an object of a concrete value, denoted as (x, t, l), given a concrete value of x and its type t; it returns l. This function will assure that the assigned label is unique across all objects in G.

$Create\_Symbol\_Obj(x, t)$, $Create\_FUNC\_Obj(x, t)$, and $Create\_OP\_Obj(x, t)$ are similar to $Create\_Concrete\_Obj(x, t)$. However, they are used to create objects for a symbol value, a built-in function, or an operator, respectively. All these functions return the label of the created object, which is unique across all objects in G.

$Add\_Concrete\_Obj(G, l)$ is to add an object of a concrete value whose label is l, denoted as $o = (x, t, l)$, into heap graph G. Specifically, we will have $C = C \cup \{x\}$, $T = T \cup \{t\}$, $L = L \cup \{l\}$, and $O_C = O_C \cup \{o\}$.

$Add\_Symbol\_Obj(G, l)$, $Add\_FUNC\_Obj(G, l)$, and $Add\_OP\_Object(G, l)$ are also defined. They are similar to $Add\_Concrete\_Obj(G, l)$ but they operate on S and O_S, $FUNC$

and O_{FUNC}, and OP and O_{OP}, respectively.

$\underline{Add\_Edge(G, e)}$ will add an edge into $Edge$ of G. Specifically, $Edge = Edge \cup \{e\}$.

$\underline{Get\_Map(Env, v)}$ where v is a variable name. $Get(Env, v)$ will return the label l of the object associated with v in Map of Env (i.e., $(v, l) \in Map$). If v is not contained in Map, $Get(Env, v)$ will return null.

$\underline{Add\_Var(Env, v)}$ where v is a variable name. This function will add the variable name v into the Var of Env (i.e., $Var = Var \cup v$).

$\underline{Add\_Map(Env, (v, l))}$ where v is a variable name and l is the label of an object. This function adds an association between v and l into the Map of Env (i.e., $Map = Map \cup (v, l)$).

$\underline{ER(G, Env, l)}$ where l is the label of an object denoted as $o = (x, t, l)$ (ER stands for "Extend_Reachability"). If $cur == null$, we will assign l to cur and add o to G (i.e., $cur = l$ and $Add\_FUNC\_Obj(G, l)$ or $Add\_OP\_Object(G, l)$). Otherwise, if $l == null$, this function simply returns cur. If $cur = (y, d, r)$ (i.e., not $null$), we will create a new object $u = Create\_OP\_Obj(\texttt{AND}, \texttt{Boolean})$ (i.e., the created object is $(\texttt{AND}, \texttt{boolean}, u)$). We then create two edges including $e_1 = (u, l)$ and $e_2 = (u, r)$ to represent the dependency between the AND operator (i.e., u) and its operands (i.e., l and r). We next add p and these two edges into G (i.e., $Add\_OP\_Object(G, u)$, $Add\_Edge(G, e_1)$, and $Add\_Edge(G, e_2)$). Finally, we update $cur = u$ so it points to the new AND operator node. This function will return the updated Env.

3.3 AST-Based Interpretation

We next design an interpreter to generate the heap graph (i.e., G) and a set of environments ((i.e., $\mathcal{E} = \{Env_1, \ldots, Env_i, \ldots, Env_n\}$)) by traversing ASTs using operations defined for heap graph and environments. The interpreter processes the root node (i.e., a file or a function) identified by us. It explores all paths towards the execution of a spe-

e **::=**	(EXPRESSION)
$\mid c$	(Constant)
$\mid x$	(Variable)
$\mid op\ e$	(Unary Operation)
$\mid e_1\ op\ e_2$	(Binary Operation)
$\mid x[e]$	(Array Access)
$\mid$ `function`$(x_1, \ldots, x_n)\{S\}$	(Func Define)
$\mid f(e_1, \ldots, e_n)$	(Func Call)
S **::=**	(STATEMENTS)
$\mid S_1; S_2$	(Sequence)
$\mid x := e$	(Assignment)
$\mid$ `if` e `then` S_1 `else` S_2	(Conditional)
$\mid$ `return` e	(Return)

Table 3.1: Core PHP Syntax Interpreted by *UChecker* (Reprinted from [22])

cific API. For example, a file upload build-in function (i.e., `move_uploaded_file()` or `file_put_content()`).

The interpreter recursively evaluates each node in the AST, where the evaluation function is denoted as $eval(node, G, \mathcal{E})$. $node$ refers to an AST node, representing either an expression (e.g., constant, variable, binary operation, and etc) or a statement (e.g., sequence, assignment, conditional, and etc.); G is the heap graph; $\mathcal{E}$ is a set of environments.

The interpreter starts with the initialization of G and $\mathcal{E}$. For heap graph G, $FUNC$ is initialized with built-in functions of PHP languages or specific platforms (such as `WordPress`); T contains primitive data types and the array data type. Other sets of G, including OP, L, O_C, O_S, O_{FUNC}, O_{OP}, and $Edge$ are all assigned as $\emptyset$. $\mathcal{E}$ is initialized with one path $\mathcal{E} = \{Env\}$. For Env, both Var and Map are initialized as $\emptyset$; $cur = null$ (i.e., the reachability constraint is empty).

The interpreter processes core PHP syntax. We use syntax presented in Table 6.1 to illustrate the design of the $eval(node, G, \mathcal{E})$ function. Without the loss of generality, we consider $\mathcal{E}$ has n paths upon evaluating an AST node, which is denoted as $\mathcal{E} = \{Env_1, \ldots, Env_i, \ldots, Env_n\}$. If $node$ is an expression or a `return` statement, $eval()$ will return a vector of labels, denoted as $< l_1, \ldots, l_i, \ldots, l_n >$, where l_i is for the ith environment. For statements other than `return`, $eval()$ modifies G and $\mathcal{E}$ but does not return

anything. For brevity, we describe the evaluation for a few challenging expressions and statements including "Variable", "Binary Operation", "Assignment", and "Conditional".

$\underline{eval(x,\ G,\ \mathcal{E})}$: When interpreter sees a variable x, it queries each Env_i in $\mathcal{E}$ to retrieve the label l_i of the object associated with x (i.e., $l_i = Get\_Map(Env_i, x)$). If $l_i \neq null$, l_i will be returned for Env_i. Otherwise, a symbol object $(s, \perp, l_i)$ will be created and added into G (i.e., $l_i = Create\_Symbol\_Obj(s, \perp)$ and $Add\_Symbol\_Obj(l_i)$); an association between x and this symbol object, (x, l_i), will then be created and inserted into the Map of Env_i (i.e., $Add\_Map(Env_i, (x, l_i))$). Finally, interpreter returns a vector of labels denoted as $< l_1, \ldots, l_i, \ldots, l_n >$, where l_i is for Env_i.

$\underline{eval(e_1\ op\ e_2,\ G,\ \mathcal{E})}$: The interpreter evaluates e_1 and e_2 using G and $\mathcal{E}$. We denote $< l_1,\ \ldots, l_i,\ \ldots,\ l_n >= eval(e_1, G,\ \mathcal{E})$ and $< r_1,\ \ldots,\ r_i,\ \ldots,\ r_n >= eval(e_2,\ G,\ \mathcal{E})$. Then for each path (i.e., Env_i), The interpreter creates a new operator object using $k_i = Create\_OP\_Obj(op, t)$, where t represents the type of the operation result. Two directed edges including $e_{i,l} = (k_i, l_i)$ and $e_{i,r} = (k_i, r_i)$ will be added into G (i.e., $Add\_Edge(e_{i,l})$ and $Add\_Edge(e_{i,r})$). interpreter preserves the order of these two edges for k_i so that it can differentiate between the "left" and "right" operand. Finally, interpreter returns a vector of labels for newly created operator objects, denoted as $< k_1, \ldots, k_i, \ldots, k_n >$, where k_i is for the ith environment Env_i.

$\underline{eval(x[e],\ G,\ \mathcal{E})}$: The interpreter first evaluates x to retrieve the label l_i of the object associated with each path (i.e., $l_i = Get\_Map(Env_i, x)$ for the ith path). If $l_i \neq null$, l_i will be returned for Env_i. Otherwise, if x is a superglobal variable, a symbol object with a specific name will be created and added into G. For example, if x refers to \$_FILES, a symbolic object $(\$\_FILES, array, l_i)$ will be created and added into G. Otherwise, $l_i == null$ and x is not a superglobal variable, a symbolic object with a randomly-generated name, i.e., $(s, \perp, l_i)$, will be created and added into G.

$\underline{eval(x,\ G,\ \mathcal{E})}$: When interpreter sees a variable x, it queries each Env_i in $\mathcal{E}$ to retrieve the label l_i of the object associated with x (i.e., $l_i = Get\_Map(Env_i, x)$). If $l_i \neq null$, l_i

will be returned for Env_i. Otherwise, a symbol object $(s, \perp, l_i)$ will be created and added into G (i.e., $l_i = Create\_Symbol\_Obj(s, \perp)$ and $Add\_Symbol\_Obj(l_i)$); an association between x and this symbol object, (x, l_i), will then be created and inserted into the Map of Env_i (i.e., $Add\_Map(Env_i, (x, l_i))$). Finally, interpreter returns a vector of labels denoted as $< l_1, \ldots, l_i, \ldots, l_n >$, where l_i is for Env_i.

$\underline{eval(e_1\ op\ e_2,\ G,\ \mathcal{E})}$: The interpreter evaluates e_1 and e_2 using G and $\mathcal{E}$. We denote $< l_1,\ \ldots,\ l_i,\ \ldots,\ l_n >= eval(e_1,\ G,\ \mathcal{E})$ and $< r_1,\ \ldots,\ r_i,\ \ldots,\ r_n >= eval(e_2,\ G,\ \mathcal{E})$. Then for each path (i.e., Env_i), interpreter creates a new operator object using $k_i = Create\_OP\_Obj(op, t)$, where t represents the type of the operation result. Two directed edges including $e_{i,l} = (k_i, l_i)$ and $e_{i,r} = (k_i, r_i)$ will be added into G (i.e., $Add\_Edge(e_{i,l})$ and $Add\_Edge(e_{i,r})$). The interpreter preserves the order of these two edges for k_i so that it can differentiate between the "left" and "right" operand. Finally, The interpreter returns a vector of labels for newly created operator objects, denoted as $< k_1, \ldots, k_i, \ldots, k_n >$, where k_i is for the ith environment Env_i.

$\underline{eval(x[e],\ G,\ \mathcal{E})}$: The interpreter first evaluates x to retrieve the label l_i of the object associated with each path (i.e., $l_i = Get\_Map(Env_i, x)$ for the ith path). If $l_i \neq null$, l_i will be returned for Env_i. Otherwise, if x is a superglobal variable, a symbol object with a specific name will be created and added into G. For example, if x refers to \$_FILES, a symbolic object $(\$\_FILES, array, l_i)$ will be created and added into G. Otherwise, $l_i == null$ and x is not a superglobal variable, a symbolic object with a randomly-generated name, i.e., $(s, \perp, l_i)$, will be created and added into G. Next, the interpreter will evaluate e for n paths in $\mathcal{E}$, getting a vector of labels denoted as $< r_1,\ \ldots,\ r_i,\ \ldots,\ r_n >= eval(e,\ G,\ \mathcal{E})$, where r_i is the label of the returned object for the ith path (i.e., Env_i).

The interpreter will then create a special operation node, an array access operation denoted as $array\_access$, for each path (i.e., $k_i = Create\_OP\_Obj(array\_access, \perp)$). The type is unknown (i.e., $\perp$) since it depends on the type of elements in this array. Two directed edges including $e_{i,l} = (k_i, l_i)$ and $e_{i,r} = (k_i, r_i)$ will be added into G (i.e., $Add\_Edge(e_{i,l})$

and $Add\_Edge(e_{i,r})$). Again, The interpreter will preserve the order of these two edges for k_i so that it can differentiate between the object of the array (i.e., "x") and that of the index (i.e., "e"). Finally, The interpreter returns a vector of labels for newly created $array\_access$ objects, denoted as $< k_1, \ldots, k_i, \ldots, k_n >$, where k_i is for the ith environment Env_i.

```
1 <?php
2     $myfile = $_FILES['upload_file'];
3     $name = $myfile['name'];
4     $rnd = $test['123'];
```

Listing 3.3: Array Access Statements (Reprinted from [22])

Figure 3.2 visualizes the heap graph and the environment for a PHP statement in Listing 3.3. `$myfile` refers to `$_FILES['upload_file']`, where `$_FILES` is recognized as a superglobal variable. Therefore, The interpreter creates an object with a special name of `$_FILES`, resulting the object $(\$\_FILES,\ array,\ 1)$ as in Figure 3.2. The object $(\text{“}upload\_file\text{”},\ string,\ 2)$ is created to indicate the index to access an array. Finally, the object $(arrray\_access,\ \perp,\ 3)$ is introduced to combine $(\$\_FILES,\ array,\ 1)$ and $(\text{“}upload\_file\text{”},\ string,\ 2)$, as the name and index of an array, respectively. When `$name = $myfile['name']` is evaluated, $myfile can be retrieved from the environment. Therefore, $(arrray\_access, \perp, 5)$ is introduced to combine $(arrray\_access,\ \perp,\ 3)$ and $(\text{“}name\text{”},\ string,\ 4)$. When `$rnd = $test['123']` is evaluated, the variable `$test` cannot be found in the environment. `$test` is also not recognized as a superglobal variable. Consequently, an object for the symbolic value with the array type, $(s_{\$test},\ array,\ 6)$, is introduced to represent the array. The object for the index, $(\text{“}123\text{”},\ string,\ 7)$, is also created. $(arrray\_access,\ \perp,\ 8)$ is finally created to combine $(s_{\$test},\ array,\ 6)$ and $(\text{“}123\text{”},\ string,\ 7)$ as array name and index, respectively.

$eval(x := e,\ G,\ \mathcal{E})$: The interpreter will first evaluate e and get a vector of returned labels, i.e., $< l_1,\ \ldots,\ l_i,\ \ldots,\ l_n >= eval(e,\ G,\ \mathcal{E})$. Then for each Env_i, The interpreter will

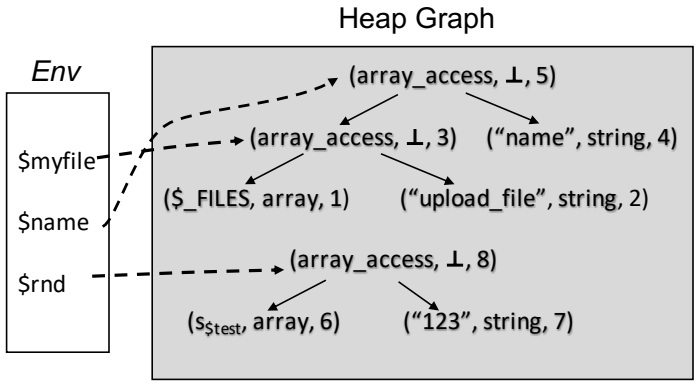

Figure 3.2: The heap graph for array access statements in Listing 3.3 (Reprinted from [22])

add an association between x and l_i into the Map of Env_i (i.e., $Add\_Map(Env_i,\ (x,\ l_i))$).

$eval(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ G,\ \mathcal{E})$: The interpreter will first evaluate e using G and $\mathcal{E}$ for all paths, denoted as $< l_1, \ldots, l_i, \ldots, l_n >= eval(e, G, \mathcal{E})$, where l_i is for Env_i. Then we will make two copies of $\mathcal{E}$, denoted as $\mathcal{E}_T$ and $\mathcal{E}_F$ where $\mathcal{E}_T = \mathcal{E}_F = \mathcal{E} = \{Env_1,\ \ldots,\ Env_n\}$. The interpreter then follows the next three steps.

- We first evaluate the "`true`" branch. For each path (e.g., Env_i) in $\mathcal{E}_T$, we extend its reachability constraint by including this "`true`" branching condition representing by l_i. Specifically, for each Env_i in $\mathcal{E}_T$, The interpreter calls $ER(G,\ Env_i,\ l_i)$. Following this, The interpreter will extend all environments in $\mathcal{E}_T$ with their corresponding '`true`" branching conditions, resulting a set of new environments named $\mathcal{E}'_T = \{ER(Env_1,\ l_1), \ldots, ER(Env_n,\ l_n)\}$. Then, The interpreter will recursively call $eval(S_1,\ G,\ \mathcal{E}'_T)$ and result in a set of new environments denoted as $\mathcal{E}''_T = \{Env_{T,1},\ \ldots,\ Env_{T,u}\}$, where $u \geq n$.

- We next evaluate the "`false`" branch. For each path (e.g., Env_i) in $\mathcal{E}_F$, we extend its reachability constraint by including the negate of the "`true`" branching condition (i.e., l_i). Towards this end, for each Env_i in $\mathcal{E}_F$, we first create a "`NOT`" operator node using $r_i = Create\_OP\_Obj(\texttt{NOT},\ boolean)$ and add r_i into G us-

ing $Add\_OP\_Obj(G, r_i)$, where r_i represents the "`false`" condition. We also create an edge between r_i and l_i, denoted as $e_i = (r_i,\ l_i)$ and add it into G using $Add\_Edge(G, e_i)$. Next, The interpreter calls $ER(G,\ Env_i,\ r_i)$. Following this, The interpreter will extend all environments in $\mathcal{E}_F$ with their corresponding '`false`" branching conditions, resulting a set of new environments named $\mathcal{E}'_F = \{ER(Env_1,\ r_1),\ \ldots,\ ER(Env_n,\ r_n)\}$. Then, The interpreter will recursively call $eval(S_2,\ G,\ \mathcal{E}'_F)$ and produce a set of new environments denoted as $\mathcal{E}''_F = \{Env_{F,1},\ \ldots,\ Env_{F,v}\}$, where $v \geq n$.

- After both branches are evaluated, The interpreter will join their resulted environment using $\mathcal{E} = \mathcal{E}''_T \cup \mathcal{E}''_F$

It is worth noting that the interpreter maintains the mapping between the line number of each expression or statement, which can be derived from AST, and nodes that are created due to the evaluation of this expression or statement.

3.4 Assigning Symblic Values

The interpreter introduces symbolic values to the heap graph through three sources including i) uninitialized variables, ii) built-in functions, and iii) PHP superglobal variables. Some variables are uninitialized since The interpreter conducts symbolic execution on a fraction of PHP programs identified by locality analysis. The interpreter also performs light-weight type inference to assign a type to a symbolic value based on its associated operator or built-in function.

The interpreter handles $_FILES as a special case since its structure is known a priori. Specifically, as discussed in Section 2, $_FILES is a pre-structured array that are indexed by 5 keys including "name", "type", "tmp_name", "error", and "size", which represent the original file name, the type information, the temporal filename, the error information, and

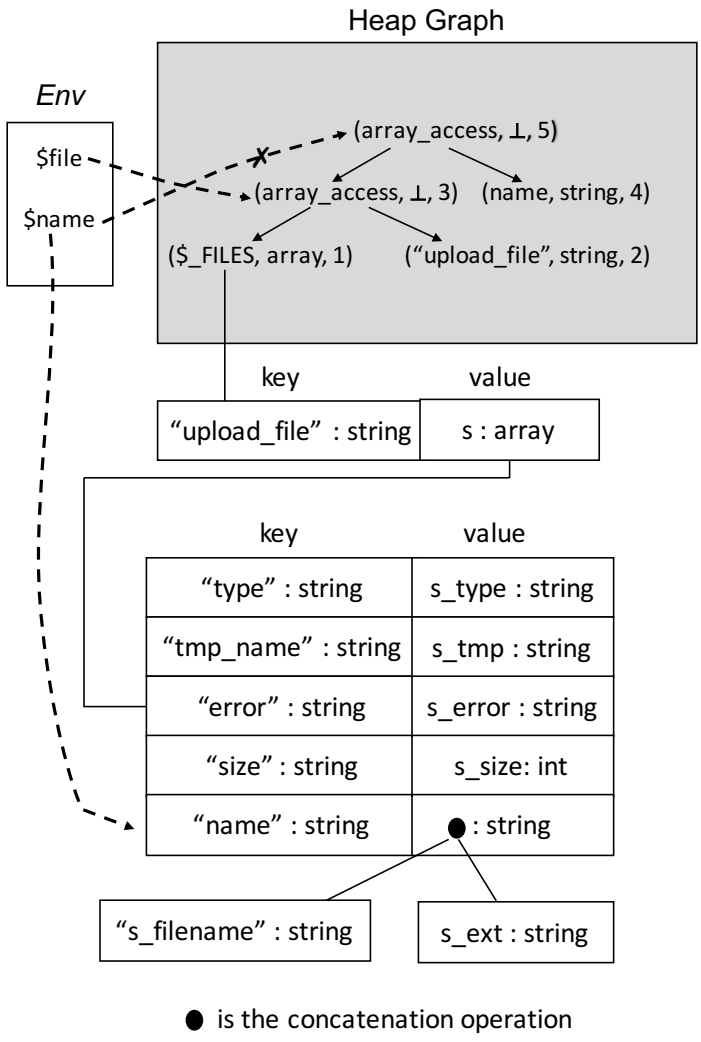

Figure 3.3: An example of pre-structured array built for $_FILES in Listing 3.3. s, s_{type}, s_{tmp}, s_{error}, s_{size}, s_{path}, s_{name}, and s_{ext} are symbolic values. (Reprinted from [22])

the size of the file. We therefore build an array with the pre-defined structure, keys, and symbolic values. In addition, certain values also have pre-defined structures. For example, the value "name" refers to the concatenation of the file name (say $s_{filename}$) and the extension (say s_{ext}), where $s_{filename}$ and s_{ext} are their symbol values. Therefore, the value of "name" can be represented as a structured symbolic value denoted as $(".",\ s_{filename},\ s_{ext})$, where "." is the concatenation operation.

Figure 3.3 presents how $_FILES, i.e., $(\$\_FILES,\ array,\ 1)$ in Figure 3.2, is extended to a pre-structured array. In this case, it is possible to return a specific symbolic value for accessing an element in a pre-structured array. Specifically, `$name`, where `$name = $myfile['name']`, can now directly point to $(".",\ s_{filename},\ s_{ext})$.

Chapter 4: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities – UChecker

4.1 Motivation

Web applications with unrestricted file upload vulnerabilities will allow attackers to upload a file with malicious code, which can be executed on the server. File upload vulnerabilities are taken as top web vulnerabilities by OWASP [13] and has been recognized as one of most common vulnerability types [41] for `WordPress`, a leading PHP-based open-source content management system (CMS) [56]. However, despite active case studies [40, 32], a systematic detection method is still missing. To this end, *we have built a system, namely UChecker, to detect PHP server-side web applications with unrestricted file upload vulnerabilities*. *UChecker* currently focuses on PHP considering its dominating role in implementing server-side web applications. We proposed following design objectives for *UChecker*:

- Automated: *UChecker* will be fully automated, requiring no users' intervention.
- Effective and Efficient: *UChecker* can detect vulnerable web applications to achieve

high accuracy with reasonable consumption of computational resources.

- Source-Code-Focused: *UChecker* can offer developers with precise source-code-level formation of the program such as lines of code that are relevant to the vulnerability.

Building *UChecker*, however, is faced with significant challenges. First, symbolic execution is computationally expensive and known to suffer from the path explosion challenge [10]. Web applications, unfortunately, are usually sizable, implying a large number of execution paths. Second, the PHP programming language has significant semantic gaps compared to languages used by SMT solvers. For example, PHP is dynamic-typing while SMT solvers are static-typing. Finally, abstract syntax trees features complex tree structures and a variety of source-code-level operations, making it impossible to applying symbolic execution methods designed for IRs such as SSA and its variants.

In order to systematically address these challenges, we have made the following contributions:

- We have designed a novel algorithm to drastically reduce the workload of symbolic execution using vulnerability-oriented locality analysis.
- We have designed an interpreter to perform context-sensitive symbolic execution using AST, modeling vulnerabilities using PHP-based operations, functions, and symbolic values in the form of s-expressions.
- We have designed a set of rules to translates PHP-based vulnerability models into SMT constraints by mitigating semantic gaps between PHP and the SMT language.
- We have implemented *UChecker* and evaluated it using 13 vulnerable and 28 non-vulnerable real-world PHP applications. *UChecker* has detected 12 out of 13 vulnerable applications at the cost of introducing 2 false positives out of 28 benign samples.

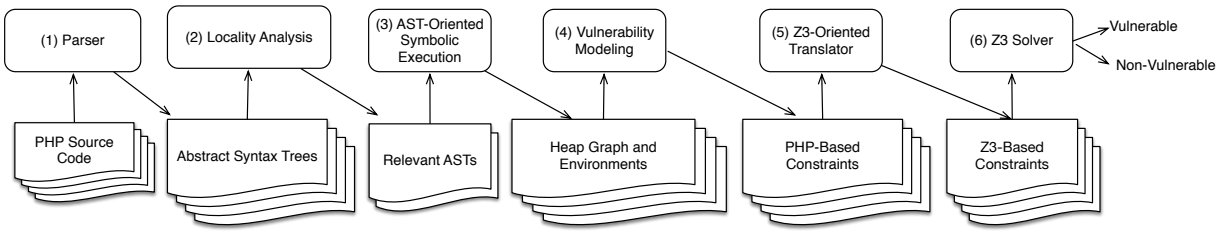

Figure 4.1: *UChecker* Architecture (Reprinted from [22])

- *UChecker* has also detected 3 potentially-exploitable `WordPress` plugins, which, to the best of our knowledge, have not been previously reported.

4.2 System Design

Figure 4.1 presents the architectural overview of *UChecker*, which has 6 major phases.

- **Parsing**: The input of *UChecker* is a set of PHP files for a web application, where *UChecker* parses them to generate AST(s).
- **Vuln-Oriented Locality Analysis**: *UChecker* identifies a small fraction of code, in the form of AST(s), which is relevant to the vulnerability, aiming at reducing the workload of symbolic execution. (see Sec. 4.2.1)
- **AST-Based Symbolic Execution**: *UChecker* next performs symbolic execution on a small fraction of AST using a novel, compact data structure named *heap graph*. (see Chapter. 3)
- **Vulnerability Modeling**: Using heap graph, *UChecker* models vulnerabilities using two constraints including a reachability constraint and an extension constraint. While the first constraint concerns whether a file uploading operation (i.e., `move_uploaded_file()` or `file_put_content()`) is reachable, the second one models the extension of a file to upload (e.g., a "`.php`" file). These constraints are s-expressions using PHP-based operators and functions. (see Sec. 4.2.2)

- **Z3-Oriented Translation**: *UChecker* translates PHP-based constraints into Z3-based constraints [17] guided by a set of novel translation rules. These rules aim to mitigate the semantic gap between PHP and Z3. (see Sec. 4.2.3)

- **SMT-Based Verification**: *UChecker* evaluates the satisfiability of Z3-based constraints using the Z3 SMT solver.

4.2.1 Vulnerability-Oriented Locality Analysis

It is challenging to perform whole-program symbolic execution considering the large number of external inputs and large program sizes, which are typical for web applications. To address this challenge, we propose to identify a fraction of code that is highly likely to be relevant to file upload and conduct symbolic analysis only for it.

Our locality analysis is driven by the observation that file upload is usually one of many functions of a web application. Therefore, the objective of locality analysis is to identify modules, functions, and files that are likely used for file upload. As discussed in Chapter 2, file upload usually retrieves file information from $_FILES and save it to local file system using built-in functions such as `move_uploaded_file()` and `file_put_content()`. Hence, the access to $_FILES and the invocation of `move_uploaded_file()` together imply the boundary of the program relevant to file upload.

Our locality analysis algorithm accordingly has two steps. First, we build a set of call graphs, which slightly extend the original definition of call graphs [30]. Specifically, each node in the graph can represent a function, a PHP file, a read access to $_FILES, the invocation of `move_uploaded_file()` (or `file_put_content()`). A directed edge (say $e = (a, b)$) between two nodes represents one of the following four scenarios.

- Both a and b are PHP files and a refers b using "`include`" or "`require`".

- a is a PHP file, b is function, and a calls b in its body.

- Both a and b are functions, where a calls b.
- a is a PHP file or a function, b is $_FILE, and a accesses b in its body (or its parameter input if a is a function).

It is worth noting that we will not build edges for recursive calls. As a result, each call graph is connected but acyclic, thereby forming a tree.

Second, if a call graph contains both the $_FILES node, say $node_1$, and the `move_uploaded_file()` (or `file_put_content()`), say $node_2$, we will identify the node that serves as *the lowest common ancestor* between these $node_1$ and $node_2$. We will only perform symbolic analysis for the code in the body of this lowest common ancestor, which is either a PHP file or a function.

```
1 <?php
2     function getFileName($file){
3         return $_FILES[$file]['name'];
4     }
5
6     function handle_uploader($file, $savePath){
7         $path_array = wp_upload_dir();
8         $pathAndName = $path_array['path'] . "/" . $savePath;
9         if (!move_uploaded_file($_FILES[$file]['tmp_name'],
10                                 $pathAndName)) {
11             return false;
12         }
13         return true;
14     }
15
16     if (!handle_uploader("upload_file",
17                          getFileName("upload_file"))) {
18        echo "File Uploaded failure!";
19     }
20 ?>
```

Listing 4.1: An example PHP file named "example1.php"(Reprinted from [22])

An example PHP file namely "`example1.php`" is presented in Listing 4.1. It invokes `handle_uploader()`, which next invokes `getFileName($file)` in its parameter. There-

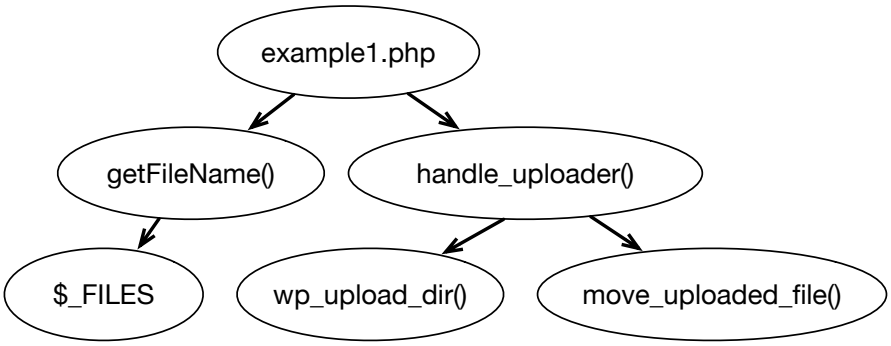

Figure 4.2: The extended call graph generated from Listing 4.1 (Reprinted from [22])

fore, "`example1.php`" calls two functions including `getFileName($file)` and `handle_uploader($file, $savePath)`, where `getFileName()` accesses $_FILES and `handle_uploader()` calls `move_uploaded_file()`. *UChecker* will construct an extended call graph for `example1.php` as illustrated in Figure 4.2. The node "`example1.php`" is the lowest common ancestor for the "$_FILES" node and the "`move_uploaded_file()`" node. Therefore, *UChecker* will perform symbolic analysis for the body of `example1.php` (i.e., starting from line 14 in List 4.1). Other scripts, if they do not contain such lowest common ancestors, will not be analyzed.

4.2.2 Vulnerability Modeling

Once our interpreter encounters a sensitive file writing operation (i.e., `move_uploaded_file` (e_{src}, e_{dst}) or `file_put_content`(e_{dst}, e_{src})), it will generate Z3 constraints to model conditions to exploit a vulnerability. e_{src} and e_{dst} indicate the source and destination files, respectively. We will use `move_uploaded_file`(e_{src}, e_{dst}) to illustrate our design, which is also applicable to `file_put_content($dst, $src)`. Specifically, `move_uploaded_file` (e_{src}, e_{dst}) can be exploited when the following three conditions are simultaneously satisfied for *at least one path*. In other words, if no path can satisfy all these three conditions, this program is free from this vulnerability.

- Constraint-1: The content of the file to be created (i.e., e_{src}) is tainted by $_FILES.
- Constraint-2: The name of the file to be created (i.e., e_{dst}) has "php" or "php5" as the file extension, making it executable. We use "php" in the following sections to simplify the illustration.
- Constraint-3: `move_uploaded_file`(e_{src}, e_{dst}) is reachable (i.e., the reachability constraint of this path can be satisfied).

In order to verify constraint-1, we can first evaluate e_{src} using G and $\mathcal{E}$ to obtain objects for all paths, denoted as $< l_{s,1}, \ldots, l_{s,i}, \ldots, l_{s,n} >= eval(e_{src}, G, \mathcal{E})$, where $l_{s,i}$ is the label of the object for e_{src} of the ith path. Then e_{src} is tainted by $_FILES (i.e., constraint-1 is satisfied) if there exists a path in G from the objected referred by $l_{s,i}$ to $_FILES.

Constraint-2 and constraint-3 will be formally verified using SMT. We evaluate e_{dst} using G and $\mathcal{E}$ to obtain labels for resulted objects, denoted as $< l_{d,1}, \ldots, l_{d,i}, \ldots, l_{d,n} >= eval(e_{dst}, G, \mathcal{E})$, where $l_{d,i}$ is the label of the object for e_{dst} of the ith path. In addition, the reachability constraint has already been represented by cur for each path Env_i (denoted as cur_i). By traversing G starting from $l_{d,i}$ and cur_i, we can generate 2 s-expressions for e_{dst} and the reachability constraint, respectively. For the s-expression of e_{dst}, we will evaluate whether it is possible to find assignments for symbolic values so that it ends up with the string ".php"; for that of the reachability constraint, we will evaluate the feasibility to find assignments for symbolic values to make it as `true`. We leverage Z3 with string extensions [62] to verify the last two constraints.

```
1 <?php
2     $path_array = wp_upload_dir();
3     $filename = $_FILES['upload_file']['name'];
```

```
4     $pathAndName = $path_array['path'] . "/" . $filename;
5     if(strlen($filename) > 5){
6         move_uploaded_file($_FILES["upload_file"]['tmp_name'],
7                            $pathAndName);
8     }
9 ?>
```

Listing 4.2: An example PHP file with unrestricted file upload vulnerability (Reprinted from [22])

Listing 4.2 presents a vulnerable example. `wp_upload_dir()` returns a symbolic value denoted as s_{dir}, to which the variable $path_array maps. Since _FILES is modeled as a pre-structured array (see Section 3.4), $_FILES['upload_file']['tmp_name'] returns (i.e., $(\text{“.”}, s_{name}, s_{ext})$), which is the concatenation of two symbolic values for the name and the extension of a file, respectively. Since $path_array itself maps to an undefined symbolic value s_{dir}, $path_array['path'] returns a symbolic value denoted as s_{path}. $pathAndName maps to the concatenation of $path_array['path'], "/", and _FILES['upload_file']['tmp_name'], which is $(\text{“.”}, s_{path}, (\text{“.”}, \text{“/”}, (\text{“.”}, s_{name}, s_{ext})))$.

For constraint-1, e_{src} is $_FILES['upload_file']['tmp_name'], which is directly tainted by $_FILES. Therefore, constraint-1 is satisfied. For constraint-2 and constraint-3, we can use the heap graph to derive the s-expression of e_{dst}, denoted as se_{dst}, and that for the reachability, denoted as $se_{reachability}$, which are listed as follows:

- $se_{dst} = (\text{“.”}, s_{path}, (\text{“.”}, \text{“/”}, (\text{“.”}, s_{name}, s_{ext})))$, where "." is the concatenation operator in PHP.
- $se_{reachability} = (>, (strlen, (\text{“.”}, s_{name}, s_{ext})), 5)$, where ">" and "strlen" are operators in PHP.

4.2.3 Z3-Oriented Constraint Translation

Despite the fact that both se_{dst} and $se_{reachability}$ are in s-expressions, their semantics, however, are based on PHP rather than Z3, forming a semantic gap. We design a transla-

tion function, namely $trl()$, to recursively translate PHP-based s-expressions into Z3-based ones. With the application of $trl()$, two constraints can be expressed as below, where "str.suffixof" is Z3's operator for suffix checking.

- Constraint-2: `(str.suffixof ".php"` $trl(se_{dst})$`)`
- Constraint-3: $trl(se_{reachability})$

$trl()$ implements a set of translation rules, where the core translation rules are presented in Table 4.1. These translation rules focus on solving four problems including i) different operation names, ii) order of parameters and missing parameters, iii) dynamic typing of PHP, and iv) missing operations in Z3. One example is the logical "And" operator in PHP, which works for different types such as string, integer, and boolean. However, the "and" operator in Z3 can only handle boolean variables. Therefore, we translate the "And" operation in PHP into a set of "and" operations depending on the type of the variable.

It is worth noting that exceptions might be observed when translating s-expressions in PHP. On the one hand, the type matching might not be satisfied. On the other hand, internal structures of certain expected inputs are invisible to *UChecker*. In this case, $trl()$ returns a symbolic value with the expected type of the operator. For example, if "/" in an expression cannot be determined in e, $trl((\text{“}basename\text{”}, e : string))$ will return a new symbol value with the string type.

Using these translation rules, *UChecker* translates constraints-2 and constraint-3 in PHP for the example in Listing 4.2 into those in Z3 as listed below:

- Constraint-2: `(str.suffixof ".php" (str.++` s_{path} `(str.++ ‘‘/" (str.++,` s_{name}`,` s_{ext}`))))`
- Constraint-3: `(> (str.len (str.++` s_{name} s_{ext}`)) 5)`

Operation	PHP	Z3
Constant c	$trl(c : t)$	c
Symbolc s	$trl(s : t)$	s (a symbol value in Z3 with type t)
String concat	$trl((\text{“.”}, e_1 : t_1, e_2 : t_2))$	(str.++ $trl(e_1 : t_1)$ $trl(e_2 : t_2)$) where $t_1 = t_2 = string$
String replace	$trl(\text{“}str\_replace\text{”}, e_1 : t_1, e_2 : t_2, e_3 : t_3))$	(str.replace $trl(e_3 : t_3)$ $trl(e_1 : t_1)$ $trl(e_2 : t_2)$), where $t_1 = t_2 = t_3 = string$
String to int	$trl((\text{”}intval\text{”}, e : t))$	(str.to.int $trl(e : t)$) where $t = string$
Index of string	$trl((\text{“}strpos\text{”}, e_1 : t_1, e_2 : t_2))$	(str.indexof $trl(e_1 : t_1)$ $trl(e_2 : t_2)$) where $t_1 = t_2 = string$
String length	$trl(\text{“}strlen\text{”}, e : t)$	(str.len $trl(e : t)$) where $t = string$
Logical Not	$trl(\text{“!”}, e : t)$	(not $trl(e : t)$) if e:boolean (not (= $trl(e : int)$ 0) if e:int (= (str.len $trl(e : string)$) 0) if e:string
Logical AND	$trl(\text{“}And\text{”}, e_1 : t_1, e_2 : t_2)$	(and $trl(e_1 : t_1)$ $trl(e_2 : t_2)$) if $t_1 = t_2 = boolean$ (and (not (= $trl(e_1 : t_1)$ 0)) $trl(e_2 : t_2)$) if $t_1 = int$ and $t_2 = boolean$ (and (> (str.len $trl(e_1 : t_1)$) 0) $trl(e_2 : t_2)$) if $t_1 = string$ and $t_2 = boolean$ (and (> (str.len $trl(e_1 : t_1)$) 0) (not (= $trl(e_2 : t_2)$ 0))) if $t_1 = string$ and $t_2 = int$
Logical Equal	$trl(\text{“} == \text{”}, e_1 : t_1, e_2 : t_2)$	(= $trl(e_1 : t_1)$ $trl(e_2 : t_2)$) if $t_1 = t_2 = boolean.$ (= $trl(e_1 : t_1)$ $trl(e_2 : t_2)$) if $t_1 = t_2 = integer.$ (= $trl(e_1 : t_1)$ $trl(e_2 : t_2)$) if $t_1 = t_2 = string.$ (= $trl(e_1 : t_1)$ (¿ $trl(e_2 : t_2)$ 0)) if $t_1 = boolean$ and $t_2 = integer.$ (= $trl(e_1 : t_1)$ (> (str.len $trl(e_2 : t_2)$) 0)) if $t_1 = boolean$ and $t_2 = string.$ (= $trl(e_1 : t_1)$ (str.to.int $trl(e_2 : t_2)$)) if $t_1 = integer$ and $t_2 = string.$
Array Check	$trl((\text{“}in\_array\text{”}, needle, haystack : array))$	(or $trl(\text{“} = \text{”}, needle, e_1 : t_1)$, ..., $trl(\text{“} = \text{”}, needle, e_n : t_n)$ if $haystack$ is recognized as $\{e_1 : t_1, \ldots, e_n : t_n\}$; a symbol value in Z3 with the type of string otherwise.
Substring	$trl((\text{“}substr\text{”}, str : t_1, start : t_2))$	(str.substr $trl(str)$, $trl(start)$, (str.len $trl(str)$)) where $t_1 = t_2 = string$
Substring	$trl((\text{“}substr\text{”}, str : t_1, start : t_2, len : t_3))$	(str.substr $trl(str)$, $trl(start)$, $trl(len)$) where $t_1 = t_2 = string$ and $t_3 = int$
Tail Element	$trl((\text{“}end\text{”}, haystack : array))$	$trl(e_n : t_n)$ if $haystack$ is recognized as $\{e_1 : t_1, \ldots, e_n : t_n\}$; a symbol value in Z3 with the type of string otherwise.
File Name	$trl((\text{“}basename\text{”}, e : string))$	the filename if e can be recognized as an absolute file path; a symbol value in Z3 with the type of string otherwise.

Table 4.1: Examples of rules to translate PHP-based constraints into Z3-based constraints (Reprinted from [22])

4.3 Evaluation

We have implemented *UChecker* with approximately 30K LoC in PHP. *UChecker* leverages PHP-Parser [35] to parse source code for AST generation and Z3 [62] as the SMT solver. *UChecker* is deployed on a Windows-10 workstation with Intel i7-5500U CPU and 16GB of memory. PHP 7 is used as the runtime environment.

4.3.1 Ground-Truth-Available Experiments

We have collected totally 13 publicly-reported vulnerable PHP applications. The vulnerable samples include 11 `WordPress` plugins [57], a Joomla extension (i.e., Joomla-Bible-study 9.1.1 [39]), and a Drupal module (i.e., Avatar Uploader 6.x-1.2 [38]). We have manually identified 28 vulnerability-free `WordPress` plugins that supports file upload. We have manually audited or tested the code to assure they are free from unrestricted file upload vulnerabilities. It is practically challenging to collect source code of publicly-available, real-world vulnerable applications; identifying and verifying vulnerability-free applications are also highly labor-intensive considering the diversity of plugins, their highly-customized interfaces, and a large number of functions. This dataset represents our current best-effort practices. The second and third columns of the top 13 rows in Table 6.3 present the names and LoC of all 13 vulnerable applications. For brevity, we only present 2 out of 28 non-vulnerable samples. Again, it is worth noting that all 28 non-vulnerable plugins we used for false-positive evaluation support file uploading.

Performance: *UChecker* starts with locality analysis to reduce lines of code for symbolic execution. The third column of Table 6.3 presents LoC of each application and the fourth column shows the percentage of LoC that are actually symbolically executed. Experiments have shown that the locality analysis drastically reduced the LoC, ranging from 67% (see Avatar Uploader) to 99.7% (see WP Marketplace). For example, “WP Market place” has

	System	LoC	% of LoC Analyzed	Paths	Objects	Objects/Path	Memory(MB)	Time(s)	Detection
Known Vulnerable	Adblock Blocker 0.0.1	484	13.02	7	158	23	4.9	0.50	Yes
	WP Marketplace 2.4.1	10850	0.29	2	55	28	4.7	2.60	Yes
	Foxypress 0.4.1.1-0.4.2.1	15815	0.60	65	1671	26	5.2	2.98	Yes
	Estatik 2.2.5	9913	1.78	12	269	22	5.2	1.72	Yes
	Uploadify 1.0.0	80	35.00	2	35	18	4.7	0.31	Yes
	MailCWP 1.100	2847	0.98	8	161	20	4.7	5.80	Yes
	WooCommerce Catalog Enquiry 3.0.1	3565	3.25	34	373	11	5.1	0.96	Yes
	N-Media Website Contact Form with File Uploader 1.3.4	1099	9.46	126	1679	13	5.2	1.23	Yes
	Simple Ad Manager 2.5.94	4340	7.70	1476	13628	9	9.3	5.35	Yes
	wp-Powerplaygallery 3.3	2757	3.77	1224	16138	13	6.6	2.78	Yes
	Joomla-Bible-study 9.1.1	94659	0.25	16	236	15	5.6	13.72	Yes
	Avatar Uploader 6.x-1.2	458	32.53	9216	62600	7	62.9	52.74	Yes
	Cimy User Extra Fields 2.3.8	9432	2.07	248832	2780067	11			No
False Positives	Event Registration Pro Calendar 1.0.2	16771	0.20	3	79	26	4.8	0.25	Yes
	Tumult Hype Animations 1.7.1	11914	0.19	4	66	16	5	0.236	Yes
New Vuln	File Provider 1.2.3	138	52.17	33	474	14	5.2	0.40	Yes
	WooCommerce Custom Profile Picture 1.0	983	2.65	2	45	23	4.8	0.28	Yes
	WP Demo Buddy 1.0.2	2196	1.32	2	85	42.5	4.83	0.277	Yes

Table 4.2: Detection Results. *UChecker* detected 12 out of 13 known vulnerable scripts at the cost of 2 false positives out of 28 benign samples. It detected 3 unreported vulnerable plugins. (Reprinted from [22])

10,850 LoC while only 32 LoC are symbolically executed.

UChecker then performs AST-based symbolic execution for the selected fraction of code. The fifth column of Table 6.3 shows the number of paths generated by symbolic execution. Despite the small LoC for symbolic execution, certain applications still yield a large number of paths. For example, the analysis of WP-Powerplaygallery, Simple Ad Manager, and Avatar Uploader leads to 1,224, 1,476, and 9,216 paths, respectively. *UChecker* also generates a considerably large number of objects in the heap graph. However, thanks to the design of heap graph that enables the sharing of objects across different environments, the average number of objects for each path is small for the vast majority of evaluated applications. Specifically, except "Cimy User Extra Fields", each path has less than 100 objects on average. Such design also results in the small memory footprint of *UChecker*, where all applications result in less than 65 MB of maximal memory consumption (see the "memory" column in Table 6.3). The analysis of each application is completed within 60 seconds as shown in the "Time" column. All these measures imply that *UChecker* can operate efficiently in a typical runtime environment.

Detection Results: *UChecker* has detected 12 out of 13 vulnerable applications as shown in the last column of Table 6.3. It introduced one false negative (i.e., "Cimy User Extra Fields"). A large number of branches in the "Cimy" plugin resulted in a massive number of paths and objects (i.e., around 248K paths and 278K objects) during the symbolic execution, which eventually exceeded the memory capacity. *UChecker* has caused 2 false positives out of 28 vulnerability-free applications.

False Positive Analysis: Two false positives are `Event Registration Pro Calendar 1.0.2` and `Tumult Hype Animations 1.7.1`. Both plugins indeed allow the upload of PHP scripts. However, accessing both plugins requires admin privilege. Since an administrator has the highest privilege of a system and she can upload arbitrary files anyway, it is acceptable for an admin script to allow the upload of arbitrary files including PHP scripts. We therefore label these two detected plugins as false positives. Listing 4.3 presents the

WordPress built-in function add_action(‘‘admin_menu’’, func_name), which is used by both scripts to make the file-upload function (specified by func_name) only accessible to admins through “admin_menu’.

```
1 <php?
2 add_action( "admin_menu", 'hypeanimations_panel_upload' );
3 //make hypeanimations_panel_upload func accessible through admin_menu
4 function hypeanimations_panel_upload() {
5   //code that allows the upload of arbitrary files
6 }
7 ?>
```

Listing 4.3: add_action("admin_menu", ’func’) makes func only accessible to the administrator in the plugin Event Registration Pro Calendar 1.0.2 (Reprinted from [22])

While *UChecker* successfully models and identifies the upload of PHP scripts, it, unfortunately, does not currently model “add_action()” to consider whether a script is running under admin’s privilege. However, we believe such false positives are acceptable since such alerts are helpful to encourage developers to specify and reassure types of uploaded files (even for the administrator).

4.3.2 Identifying New Vulnerable PHP Applications

We have employed *UChecker* to detect new vulnerable PHP applications by scanning WordPress plugins. WordPress features a large repository of PHP-based, open-source plugins that are contributed from a variety of sources. We have crawled and tested 9,160 WordPress plugins in a reverse chronological order (starting from 4/22/2018) based on their last updated time. We have detected 3 vulnerable plugins including “File Provider 1.2.3”, “WooCommerce Custom Profile Picture 1.0”, and “WP Demo Buddy 1.0.2”. To the best of our knowledge, these 3 vulnerable plugins have not been previously reported. The detection measures for these 3 plugins are presented in the bottom 3 rows in Table 6.3.

WooCommerce is a WordPress eCommerce plugin, which by itself supports third-party plugins to extend its functionality. WooCommerce Custom Profile Picture 1.0 [**?**] is one of such plugins to enable users to upload pictures to their WooCommerce profiles. As indicated by its design objective, WooCommerce Custom Profile Picture should only accept files that are images such as .jpg, .gif, and .png. The locality analysis of *UChecker* identifies that it is only necessary to perform symbolic execution for the function "wc_cus_upload_picture()". It further successfully identified these vulnerability and precisely located it in the source code, which is presented in Listing 4.4. This plugin directly uses the original filename (i.e., $profilepicture['name']) as the name of the destination file; and then it copies the uploaded file (i.e., $profilepicture['tmp_name']) to the destination file. Therefore, any registered user can submit a PHP script through this uploading interface and execute it.

```
1 <?php
2     if($_FILES['profile_pic']){
3         $picture_id = wc_cus_upload_picture($_FILES['profile_pic']);
4     }
5     function wc_cus_upload_picture( $foto ) {
6         $profilepicture = $foto;
7         $wordpress_upload_dir = wp_upload_dir();
8         $new_file_path = $wordpress_upload_dir['path'] . '/' .
9                          $profilepicture['name'];
10        //...
11        if( move_uploaded_file($profilepicture['tmp_name'],
12                               $new_file_path ) ) {
13            //...
14        }
15    }
16 ?>
```

Listing 4.4: Vulnerable Code of WooCommerce 1.0 Custom Profile Picture (Reprinted from [22])

File Provider 1.2.3 [18] is a free WordPress plugin used for website users to upload, search, and share files. Uploaded files are stored in a local folder and the administra-

tor has the option to enable end users to access files in this folder. The locality analysis of *UChecker* effectively identifies the function `upload_file()` for symbolic execution, which accounts for a small percentage (i.e., 2.65%) of all code for `File Provider 1.2.3` *UChecker* has successfully detected this vulnerability and located it in source code as presented in Listing 4.5. Specifically, $nome_final, which is actually the original filename "$nome_final=$_FILES['userFile']['name']", is used as the destination filename without sanity check. Since this plugin does not validate the type of an uploaded file, a user can upload a PHP script and then trigger its execution by accessing it.

```
1 <?php
2     function upload_file(){
3         $folderId = sanitize_text_field($_POST['folderId']);
4         $folderPath = get_file_path($folderId); // User declared ...
              method: get the upload path
5         $nome_final = $_FILES['userFile']['name'];
6         if (!move_uploaded_file($_FILES['userFile']['tmp_name'],
7             $folderPath . '/' . $nome_final)) {
8             echo '<div class="error">...</div>';
9         }
10    }
```

Listing 4.5: Vulnerable Code of `File Provider 1.2.3` (Reprinted from [22])

WP Demo Buddy 1.0.2 [1] is used to create demo instances, whose relevant source code is displayed in Listing 4.6. Although it rejects any uploaded file whose extension is not "zip", it deliberately adds ".php" priori to writing this ".zip" file into server. Therefore, an attacker can simply upload a PHP script with ".zip" extension (e.g.,, "exploit.zip"), which will be eventually written to the server as "exploit.zip.php".

```
1 <?php
2     function file_Upload($type)
3     {
4         global $wpdb;
5         $upload_dir = get_option('wp_demo_buddy_upload_dir');
6         $ext = pathinfo($_FILES[$type]['name'], PATHINFO_EXTENSION);
```

```
7         if ($ext !== 'zip') return;
8         $info = pathinfo($_FILES[$type]['name']);
9         $newname = time() . rand() . '_' . $info['basename'] . '.php';
10        $target = $upload_dir . $newname;
11        move_uploaded_file($_FILES[$type]['tmp_name'], $target);
12        $ret = array($newname, $info['basename']);
13        return $ret;
14    }
```

Listing 4.6: Vulnerable Code of `WP Demo Buddy 1.0.2` (Reprinted from [22])

4.3.3 Comparison With Other Detection Solutions

We have experimented with two publicly available PHP vulnerability scanners including RIPS [16, 14] and WAP [28], where both of them offer options to detect unrestricted file uploading vulnerabilities. Specifically, RIPS detects sensitive sinks as potential vulnerable functions if they are tainted by untrusted inputs. While taint analysis concerns the source of the uploaded file, it does not model the name or the extension of this file, thereby being likely to introduce false positives. WAP integrates taint analysis and machine learning for detection without particularly modeling the uploaded file. By scanning 28 vulnerability-free samples, 13 known vulnerable scripts, and 3 newly detected vulnerable plugins, RIPS detected 15 out of 16 vulnerable samples (missing "the WooCommerce Custom Profile") with a high false positive rate of 27/28. WAP led to a detection rate of 4/16 with a false positive rate of 1/28. We acknowledge that *UChecker* currently only focuses on unrestricted file uploading vulnerability while RIPS and WAP offer options to cover more types. Nevertheless, these systems can complement each other in practical usage.

4.4 Discussion

The current implementation of *UChecker* has a few limitations. First, *UChecker* now focuses on vulnerabilities that allow the uploading of PHP files (i.e., those with ".php" and

".php5"). However, variant vulnerabilities may allow files with other potential harmful extensions such as ".asa" and ".swf". *UChecker* can easily cover these variants by verifying more extensions. Second, as demonstrated in Section 4.3.1, false positives are mainly attributed to the fact that *UChecker* does not model `WordPress`'s built-in function namely `add_action(''admin_menu'', func_name)`. However, this does not represent a design flaw. In fact, modeling a platform-dependent function is constrained by the awareness of this function. Nevertheless, this indeed might be a practical challenge if the variety and dynamics of built-in functions are high. Third, *UChecker*'s interpreter does not cover all language features of PHP. For example, *UChecker* does not precisely model loops, which may lead to false negatives and false positives. Nevertheless, performing effective symbolic execution for loops is an intrinsic challenge of static program analysis, which is not a specific flaw of our design. In addition, PHP is a dynamic language. Although *UChecker*'s translation rules partially address challenges introduced by PHP's dynamic types, it does not tackle executable content that is generated by a PHP script at runtime. As a consequence, scripts analyzed by *UChecker* might be incomplete, leading to detection inaccuracy. A potential solution is to integrate dynamic analysis to access all executables produced at runtime. Finally, *UChecker* does not model PHP regular expression matching operations considering their high complexity. A potential solution is to leverage Z3's built-in regular-expression-enabled solver. Unfortunately, such solver may not sufficiently cover all cases that can be expressed by PHP regular expressions. Another potential solution is to integrate dynamic analysis into the interpretation process, assigning concrete values to certain symbols.

4.5 Summary

We have built *UChecker* to automatically detect PHP-based web programs with unrestricted file upload vulnerabilities. *UChecker* interprets abstract syntax trees of PHP source code

for symbolic execution, whose performance is improved by a novel vulnerability-oriented locality analysis algorithm. We model vulnerabilities using constraints and verify them using an SMT solver. Experiments have demonstrated that *UChecker* detected 3 vulnerable web applications that have not been publicly reported.

Chapter 5: Lightweight Detection of PHP-Based Unrestricted File Upload Vulnerabilities Via Static-Fuzzing Co-Analysis – UFuzzer

5.1 Motivation

In chapter 4, we have demonstrated the promising detection performance of *UChecker*, but its applicability heavily depends on the PHP-to-solver constraint translation. Such translation is guided by manually-engineered rules, which faces significant practical challenges. On the one hand, the variety, complexity, and flexibility of PHP APIs are overwhelmingly larger than those of solver APIs. On the other hand, PHP is a dynamic typing language whereas solver languages are usually static. One salient example is that the regex solver is incomplete [27], which limits *UChecker*'s capabilities to model and solve sophisticated PHP-based regex operations.

In this work, we proposed a novel, fully-automated vulnerability detection system, namely *UFuzzer*, with following design objectives:

- Effective and Efficient: *UFuzzer* should achieve high detection accuracies with low performance overhead.
- Minimal Dependency: *UFuzzer* can be sufficiently supported by a local, native PHP runtime environment.
- Operating-Free: *UFuzzer* does not need an operating web service to perform detection.
- Traceable: *UFuzzer* can precisely identify statements that introduce such vulnerabilities at the source-code level.

UFuzzer with these design objectives, once built, can systematically address the challenges faced by existing detection systems. Particularly, it avoids the semantic gaps between PHP and solver languages that are inherent to *UChecker*. It also eliminates *FUSE*'s dependency on operating web services.

We build *UFuzzer* by integrating static program analysis and fuzzing. Specifically, *UFuzzer* leverages static program analysis to identify those statements that are relevant to the unrestricted file upload vulnerability. Then it refactors these identified statements to make them independent to operating web services. *UFuzzer* next generates executable code templates from these selected, refactored statements. It finally "fuzzes" each template to perform detection. Our work makes the following contributions.

- We have designed a novel method to detect server-side scripts with unrestricted file upload vulnerabilities through static-fuzzing co-analysis.
- We have built a system, namely *UFuzzer*[1], to implement this method for PHP-based server-side web programs.
- We have evaluated *UFuzzer* using real-world, ground-truth-available data. The evaluation results have demonstrated *UFuzzer* outperforms existing methods in either

[1]We plan to release the code of *UFuzzer* once the paper is accepted by this conference.

detection accuracies, or system performance, or both.

- We have employed *UFuzzer* to detect 31 new vulnerable PHP applications by scanning a large corpus of real-world server-side PHP applications.

Listing 5.1 shows a code snippet that introduces the unrestricted file upload vulnerability, and we will use it to illustrate our system. In this snippet, the server receives a file from a remote client through the `$_FILES` superglobal variable, which is actually a two-dimension array (i.e., $_FILES[i][j]). The first index refers to the label of the uploaded file (i.e., `$_FILES['newfile']`). Accessing $_FILES using the first index returns a pre-defined one-dimensional array, which are indexed by "name", "type", "tmp_name", "error", and "size". Figure 5.2 presents the heap graph for the vulnerable code snippet in Listing 5.1 towards the end of this program.

```
1 <?php
2     $dir = "../wp-content/plugins/upload/";
3     if(isset($_POST['action'])){
4     $localDir = $dir . time();
5         $fName = preg_replace("/\s/", "",
6                               $_FILES['newfile']['name']);
7         if(is_writable(localDir)){
8         $fName = $localDir . "_" . $fName;
9         $tmpFile = $_FILES['newfile']['tmp_name'];
10        move_uploaded_file($tmpFile, $fName);
11
12        }
13    }
14    //...
15 ?>
```

Listing 5.1: A code snippet for unrestricted file upload vulnerability (Reprinted from [23])

The remaining of this chapter is organized as follows. We presents the system design in Section 5.2 . Section 5.3.1 illustrates evaluation results and Section 5.3.2 presents the detection of new vulnerable programs. Section 5.4 elaborates the limitation and potential

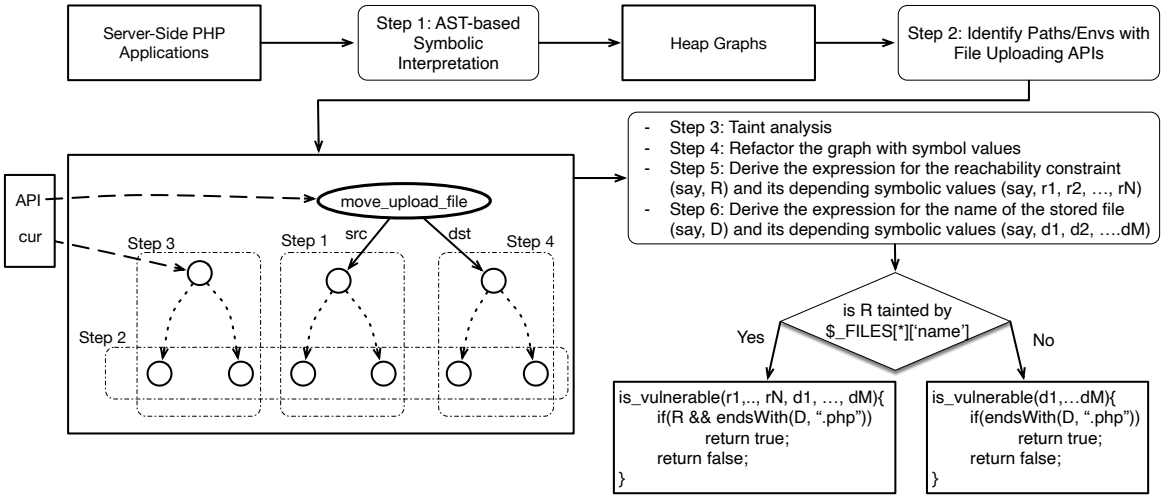

Figure 5.1: The architectural overview of *UFuzzer*(Reprinted from [23])

solutions of the current design. Section 5.5 concludes.

5.2 System Design

Figure 5.1 presents the architectural overview of *UFuzzer*. *UFuzzer* first scans each program of a PHP server-side web application and identifies whether it contains any file uploading API (i.e., `move_uploaded_file()` and `file_put_content()`). If so, *UFuzzer* will leverage the inter-procedural, context-aware symbolic interpreter in [22] to generate a heap graph for this program towards each identified file uploading API or the end of the program (i.e., Step 1 in Figure 5.1). Next, for each environment in a heap graph, we will preserve it if its $API \neq null$ (i.e., an execution path that contains a file uploading API), which is illustrated as Step 2 in Figure 5.1.

For each preserved environment, *UFuzzer* will first evaluate whether the source file of a file uploading API is derived from an untrustworthy source via taint analysis. It will next refactor the graph with symbolic values, which are used to model superglobal variables, initialized variables, and certain built-in APIs. *UFuzzer* will then traverse the heap graph to yield executable expressions that characterize the exploit conditions, including i) the

reachability constraint and ii) the name of the file to be permanently stored. These activities are illustrated as Step 3, 4, 5, and 6 in Figure 5.1.

UFuzzer will next generate executable code templates for fuzzing. Towards this end, *UFuzzer* will evaluate whether the reachability constraint is tainted by the name of the uploaded file (i.e., `$_FILES[*]['name']`) to reduce the space of variables for fuzzing. Finally, *UFuzzer* will execute each template in a local PHP environment after binding its free variables with mutated values.

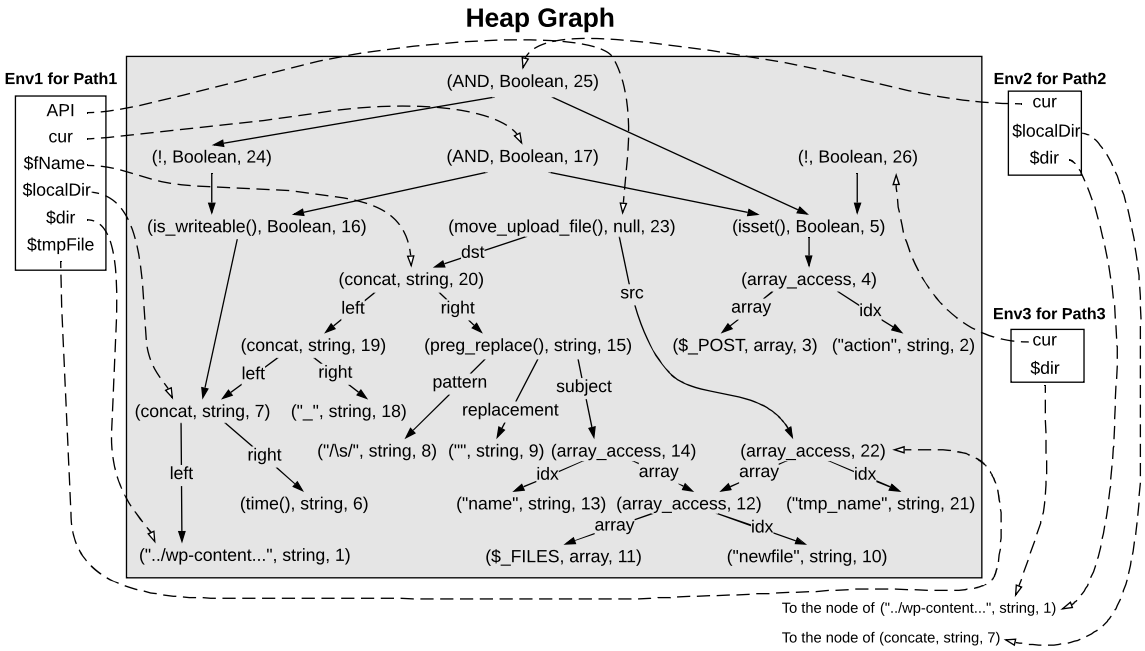

Figure 5.2: The heap graph for the sample code in Listing 5.1 (Reprinted from [23])

5.2.1 Taint Analysis

Each edge in a heap graph represents an immediate data dependency between two objects in this graph. Therefore, all edges collectively characterize global data flows among all objects along an execution path. Taint analysis can therefore be performed using heap graphs: given two objects (say α and β) in a heap graph (say G), α is tainted by β (i.e., there exists an explicit data flow from β to α) if and only if β is reachable from α in G.

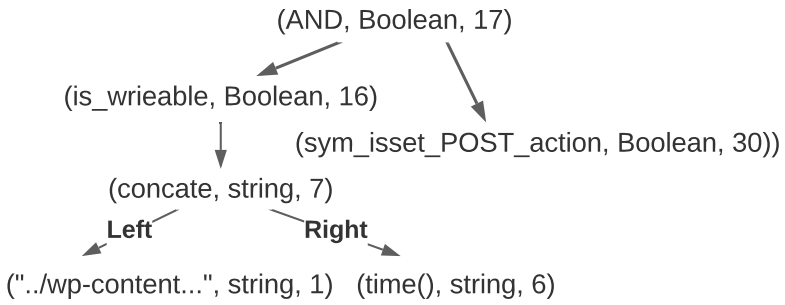

Figure 5.3: The sub-tree for the reachability constraint derived from Figure 5.2 (Reprinted from [23])

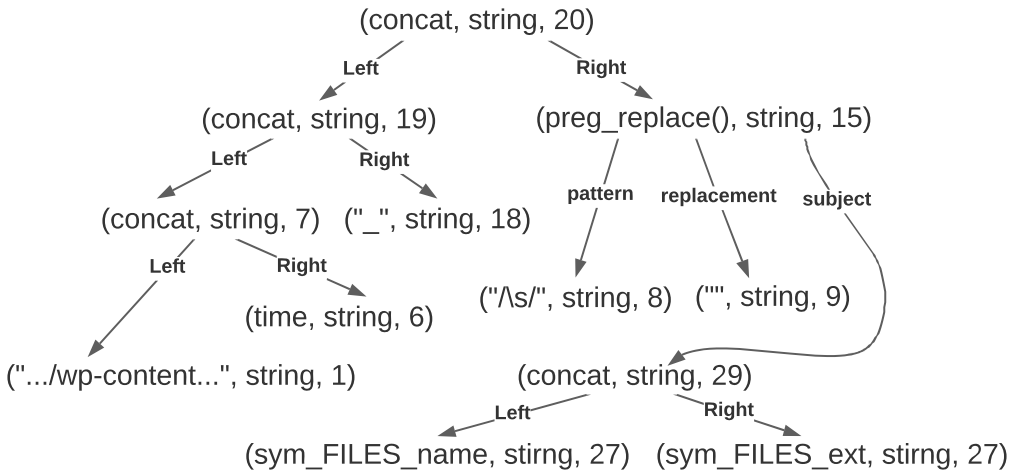

Figure 5.4: The sub-tree for the filename derived from Figure 5.2 (Reprinted from [23])

The "src" edge originated from a `move_uploaded_file()` node points to the source of the file to be permanently saved. Such file is untrustworthy if it is from external inputs. Currently, external inputs are mainly modeled as global variables in *UFuzzer*. Therefore, our objective is to verify if the `move_uploaded_file()` node is tainted by a node of a global variable *through its "src" edge*. This could be effectively fulfilled by *UFuzzer*. For example, the node of the FILES global variable (i.e., the node $(\$\_FILES, array, 11)$) is reachable from the `move_uploaded_file()` node (i.e., node 19) in Figure 5.2 through its "src" edge, indicating that the source file of the `move_uploaded_file()` API is tainted by external inputs.

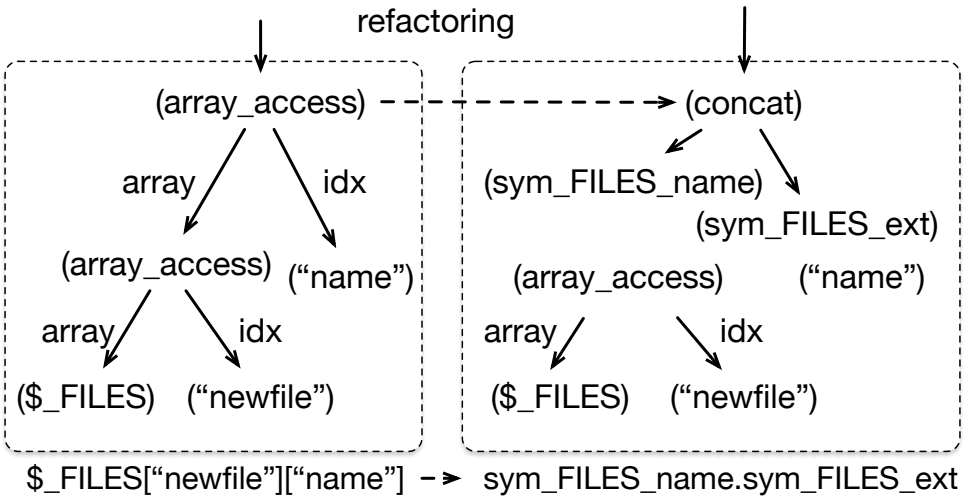

Figure 5.5: An example of refactoring an array_access node associated with the $_FILES superglobal variable. The sub-tree rooted in the top $array\_access$ denotes $_FILES["newfile"]["name"]. This $array\_access$ node will be replaced by a node that concatenates two symbolic values of the filename and the extension, respectively. (Reprinted from [23])

5.2.2 Graph Refactoring With Symbolic Values

We fuzz three data sources including i) uninitialized variables, ii) superglobal variables, and iii) certain built-in APIs. *UFuzzer* will refactor heap graphs by replacing their corresponding nodes with nodes of symbolic values.

Unintialized Variables: Our symbolic interpreter performs vulnerability-oriented locality analysis [22] to identify a sub-AST for symbolic interpretation, aiming at mitigating the path explosion challenge. Therefore, it is possible to encounter uninitialized variables in the sub-AST. For an uninitialized variable, we first create a node of a symbolic value and next establish an association between this variable and this node.

Superglobal Variables: Superglobal variables are used by external users to offer information to a web service. Therefore, we create a node of symbolic value when interpreting a superglobal variable, whose type is considered as "string".

It is worth noting that *UFuzzer* handles $_FILES as a special case since its structure is

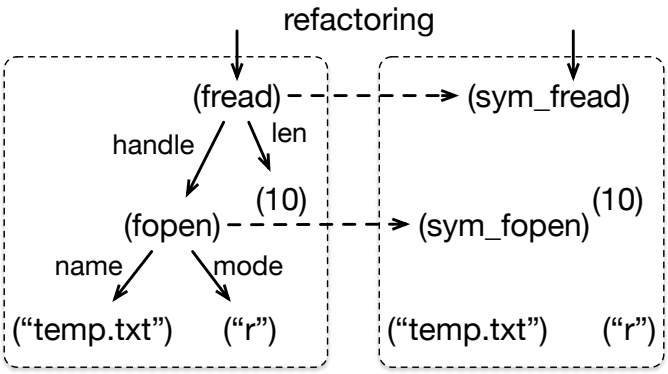

Figure 5.6: An example of refactoring a `fread` node with a node of symbolic value (i.e., sym_fread) (Reprinted from [23])

known a priori. Specifically, $_FILES is a pre-structured array that are indexed by 5 keys including "name", "type", "tmp_name", "error", and "size", which represent the original file name, the type information, the temporal filename, the error information, and the size of the file. We therefore traverse a heap graph and identify all $array\_access$ nodes where each such node satisfies two conditions: i) its "array" edge points to another $array\_access$ whose "array" edge connects a node of the $\$\_FILES$ global variable; ii) its "index" edge points to one of keys (with the string type) including "name", "type", "tmp_name", "error", and "size". The first condition indicates this is the access to the 2nd dimension of $_FILES superglobal variable and the second condition illustrates the specific key used for accessing second dimension of $_FILES. We will replace this $array\_access$ using a symbolic-value-based node in the heap graph. If this symbolic-value-based node corresponds to the "size" or "error" index, its type will be "int"; otherwise, it will be "string".

Figure 5.5 presents an example, where the type and label for each node is omitted for briefness. Specifically, the node ($array\_access$) satisfies both conditions, indicating that this node and its underlying sub-tree together represents $_FILES["newfile"]["name"]. *UFuzzer* therefore replaces this node using the concatenation of two symbolic nodes, namely sym_FILES_name and sym_FILES_ext. Here, sym_FILES_name represents the file name and sym_FILES_ext refers to the extension.

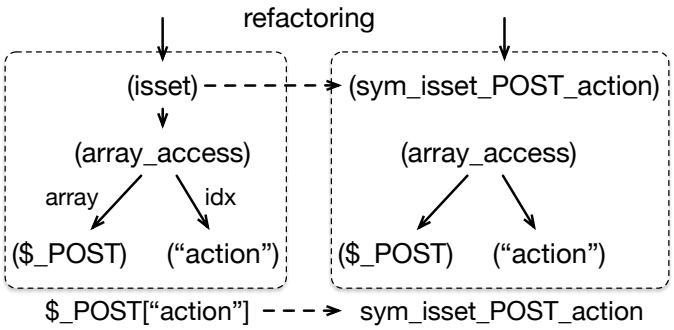

Figure 5.7: An example of refactoring a `isset` node with a node of symbolic value (i.e., $sym\_isset\_POST\_action$). (Reprinted from [23])

Operation and Validation APIs: Some APIs in the server-side web program can only function in a properly-configured run-time environment, making automatically analysis extremely challenging. These APIs are often used for operations of networking, databases, and file accessing, which can only function in a properly-configured run-time environment. We name such APIs as operation APIs. *UFuzzer* will traverse a heap graph and identify every node if it corresponds to any API that is used for networking, databases, and file operations. For an identified node, *UFuzzer* replaces this node using a node of the symbolic value, whose type will be simultaneously derived based on the API. Figure 5.6 shows an example, where the node of `fopen` and that of `fread` have been replaced using two symbolic nodes including `sym_fopen` and `sym_fread`, respectively. We respectively assign the `pointer` type and the `string` type to these two nodes based on the definition of these APIs.

We also symbolize PHP validation functions (e.g., those with "is_" as prefixes) to improve the efficiency of the fuzzing. Examples of such functions include `isset()`, `is_writable()`, and `is_string()`. For example, these functions are widely used by PHP programs, which only outputs `TRUE` or `FALSE` but have infinite input spaces. We therefore symbolize these functions regardless of their arguments. For example, any node of `isset()` in the heap graph will be replaced using a symbol node with boolean type. Figure 5.7 presents an example. The node of `isset()`, which is corresponding to the expression of

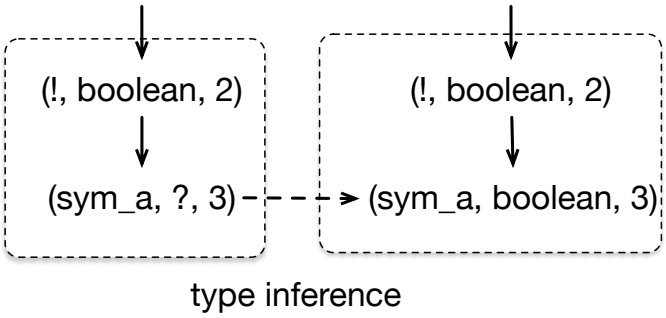

Figure 5.8: An example of type inference. Inferring the type of the symbolic node using its immediate operator node. (Reprinted from [23])

`isset($_POST['action'])`, will be replaced by a symbol node of $sym\_isset\_POST\_action$.

Lightweight Type Inference: The type information is assigned when a symbolic node for either a superglobal variable or a selected APIs is created. However, it is uncertain for those nodes created for uninitialized variables. To address this challenge, we perform lightweight type inference. Specifically, we identify the operator node or an API node that immediately depends on this node (i.e., has an edge to this node). We next use the expected operand/argument types to infer the type of this node. Figure 5.8 presents an example, where we assign "boolean" to a symbolic node of an uninitialized variable since it serves as the operand for the "`negate`" operator.

5.2.3 Deriving Executable Expressions for The Reachability Constraint and The File Name

For each preserved environment after heap graph refactoring, *UFuzzer* will generate expressions of constraints for both the reachability and the filename, and next integrate them into a function with fuzzing variables as function arguments.

The cur variable in an environment is bounded to the node of that represents the reachability constraint and we name this node as v_{reach}. The API variable in an environment is bounded to the node of a file uploading API. The "dst" edge from this API node points

to the node that represents the name of the file to be permanently saved and we name this node as $v_{filename}$. For each preserved environment, we can traverse the heap graph from v_{reach} and $v_{filename}$ to generate sub-trees of all relevant nodes. Figures 5.3 and 5.4 present sub-trees for v_{reach} and $v_{filename}$ derived from Figure 5.2, respectively.

The core function is to generate executable expressions by traversing these two sub-trees. As illustrated in Algorithm 1, we have designed an algorithm to evaluate each sub-tree (starting from its root node) and leverage an existing parser (i.e., PHP-Parser [35]) to build an AST, which will be finally converted into PHP code through the pretty printing function of this parser.

Algorithm 1 Generating AST from A Sub-Tree in Heap Graph (Reprinted from [23])

```
1:  function eval(v)
2:      switch v.getType() do
3:          case scalar
4:              val ← v.getValue()
5:              type ← v.getType()
6:              return new _Scalar_(val, type)
7:          case symbol
8:              val ← v.getValue()
9:              type ← v.getType()
10:             return new _var_(val, type)
11:         ...
12:         case binaryOP
13:             e_left ← eval(v.getLeft())
14:             e_right ← eval(v.getRight())
15:             op ← v.getOperator()
16:             return new _BinaryOp(op, e_left, e_right)
17:         case func
18:             name ← v.getName()
19:             < arg_1, ..., arg_n > ← v.getArgs()
20:             arg_list ← []                          ▷ [] : empty list
21:             for i = 1...n do
                    arg_list.add(eval(args[i]))
22:             end for
23:             return new _FuncCall_(name, arg_list)
24:         ...
25: end function
```

It is worth noting that our algorithm needs to recursively interpret all types of nodes in a heap graph. But for briefness, Algorithm 1 only presents the interpretation of nodes for constants, symbolic values, binary operators, and function calls. The interpretation will return an AST built through PHP-parser's APIs (i.e., "$new\ \_Object\_(...)$").

$\underline{scalar}$: If $eval()$ sees a scalar node it will return a scalar AST node, i.e., "$new\ \_Scalar\_(val, type)$, where val and $type$ represent its value and type respectively.

$\underline{symbol}$: When $eval()$ sees a symbol node sym, it returns a variable AST node, i.e., "$(new\ \_var\_(val,\ type))$", where val and $type$ represent the value and the type of this symbolic node, respectively.

$\underline{binaryOP}$: Upon visiting a node for a binary operator, $eval()$ will first recursively interpret its left and right child nodes and derive two AST nodes, denoted as e_{left} and e_{right}, respectively. Each of these two AST nodes could be the root of another AST tree. Finally,$eval()$ will return an AST node, i.e., "$new\ \_BinaryOp(op,\ e_{left},\ e_{right})$".

$\underline{func}$: When processing a node for function all, $eval()$ will first derive the name of the function call and all nodes of its arguments. This algorithm will then retrieve the AST node for each argument through recursive evaluation (i.e., $eval(args[i])$). Finally, it will return a function call AST node, i.e., $new\ \_FuncCall\_(name,\ arg\_list)$.

5.2.4 Generate Fuzzing Templates

By leveraging the expression generation algorithm in Algorithm 1, we can generate executable code templates, namely *fuzzing templates*, as presented in Algorithm 2. Arguments $r_1, \ldots, r_N$ represent all symbolic nodes in the sub-tree rooted in v_{reach} (i.e., the sub-tree for the reachability constraint); arguments $d_1, \ldots, d_M$ refer to all symbolic nodes in the sub-tree rooted in $v_{filename}$. The $prettyprint()$ function outputs a decompiled version of the AST in a format that is a legal PHP program for execution in a standard PHP running environment.

Algorithm 2 Fuzzing Template With Reachability (Reprinted from [23])

```
1: function IS_VULNERABLE(r_1, ..., r_N, d_1, ..., d_M)
2:     exp_reach ← prettyprint(eval(v_reach))
3:     exp_filename ← prettyprint(eval(v_filename))
4:     if exp_reach then
5:         ext ← get_extension(exp_filename)
6:         if ext == "php" then
7:             return TRUE
8:         end if
9:     end if
10:    return FALSE
11: end function
```

Algorithm 3 Fuzzing Template Without Reachability (Reprinted from [23])

```
1: function IS_VULNERABLE(d_1, ..., d_M)
2:     exp_filename ← prettyprint(eval(v_filename))
3:     ext ← get_extension(exp_filename)
4:     if ext == "php" then
5:         return TRUE
6:     end if
7:     return FALSE
8: end function
```

It will drastically increase fuzzing efficiency if we can reduce the number of variables to be mutated. Towards this end, we develop following rules:

We assess whether the reachability constraint is tainted by $_FILES[*]['name'], the name of the uploaded file, where "*" here refers to an arbitrary string. If not, it indicates that the reachability constraint does not verify the name of the uploaded file, implying no sanitization checks are enforced for the name of the uploaded file. Therefore, we only use fuzzing to evaluate the name of the file to be saved, thereby using the fuzzing template generated by Algorithm 3. if the reachability constraint is indeed tainted by $_FILES[*]['name'], we will perform fuzzing to jointly evaluate the reachability constraint and the name of the file to be saved, thereby using the template generated by Algorithm 2.

Listing 5.2 and Listing 5.3 present two fuzzing templates for Listing 5.1, which are generated by Algorithm 2 and Algorithm 3, respectively. In these two fuzzing templates, $sym_file_name, $sym_file_ext, $sym_isset_POST_action, and $sym_is_writable are vari-

ables to be mutated, where the first two have the type of strings and the last two are boolean. The `pathinfo` function used in fuzzing templates is a PHP built-in API for returning the extension of a file name with parameter `PATHINFO_EXTENSION`. The "$in\_array(\$ext, array('php'))$" is to check whether the extension is 'php', which can be easily extended to include additional sensitive extensions (e.g., 'jsp').

```
1 <?php
2     function is_vulnerable($sym_file_name,
3                            $sym_file_ext,
4                            $sym_isset_POST_action,
5                            $sym_is_writable)
6     {
7         $exp_reach = $sym_isset_POST_action and
8                      $sym_is_writable;
9         $exp_fileName = "../wp-content/plugins/upload/" .
10                        time() . "_" .
11                        preg_replace("/\s/", "",
12                                     $sym_file_name .
13                                     $sym_file_ext);
14        if($exp_reach){
15            $ext = pathinfo($exp_fileName,
16                            PATHINFO_EXTENSION);
17            if(in_array($ext, array('php'))){
18                return TRUE;
19            }
20        }
21        return FALSE;
22    }
23 ?>
```

Listing 5.2: The Fuzzing Template With Reachability Evaluated for Listing 5.1 (Reprinted from [23])

```
1 <?php
2     function is_vulnerable($sym_file_name,
3                            $sym_file_ext)
4     {
5         $exp_fileName = "../wp-content/plugins/upload/" .
6                         time() . "_" .
```

```
7                           preg_replace("/\s/", "",
8                                        $sym_file_name .
9                                        $sym_file_ext);
10
11          $ext = pathinfo($exp_fileName,PATHINFO_EXTENSION);
12          if(in_array($ext, array('php'))){
13              return TRUE;
14          }
15          return FALSE;
16      }
17 ?>
```

Listing 5.3: The Fuzzing Template Without Reachability Evaluated for Listing 5.1 (Reprinted from [23])

Since the reachability constraint of Listing 5.1 is not tainted by the name of the uploaded file (i.e., $_FILES[*][‘name’]), the fuzzing template in Listing 5.3 will be used for fuzzing. As shown by this example, it drastically reduces the fuzzing space by eliminating two free variables to use the fuzzing template in in Listing 5.3 compared to that in Listing 5.2 without undermining the detection accuracy.

5.2.5 Executing a Fuzzing Template

We then execute each fuzzing template to assess whether its corresponding PHP script is vulnerable. It starts with assigning values to arguments in the fuzzing template (i.e., in the `is_vulnerable()` function), following rules below:

- `$sym_file_ext`: if this argument refers to the extension of the name for the original uploaded file, we build a set of sensitive extensions such as “.php”, “.gif.php”, “.mp3.php”, “.zip.php”, “pdf.php”, and “.jpg.php”.

- `$sym_file_name` or an argument with the string type: if this argument represents the extension-removed name of the uploaded file or its type is string, we leverage a PHP Fuzzer [34] as a drop-in fuzzer to mutate string values for `$sym_file_name`. We use different mutators in PHP Fuzzer [34] such as `EraseBytes`, `InsertByte`,

`ChangeByte`, `ChangeBit`, and `ChangeASCIIInt`. For each mutator, we iterate for 50 times with a length between 5 and 70.

- An argument with the boolean type (e.g., `$sym_is_*`): We enumerate both `True` and `False` values to this argument if it is a boolean type.

- An argument with the number type: We enumerate values in $[-20, 20]$ for an argument if it has the number type (e.g., integer, float, and double).

We execute each fuzzing template by iterating over arguments' different values. If any execution returns `True`, we will cease the iteration and report that the PHP script is vulnerable. If no template returns `True` after all mutated values are exhausted, *UFuzzer* will stop and report this web application as non-vulnerable. We admit that the selection of these fuzzing parameters in *UFuzzer*, similar to that in other fuzzers, are empirical rather than provable. Nevertheless, they are highly configurable to support practical deployment.

5.2.6 Executing a Fuzzing Template

We then execute each fuzzing template to assess whether its corresponding PHP script is vulnerable. It starts with assigning values to arguments in the fuzzing template (i.e., in the `is_vulnerable()` function), following rules below:

- `$sym_file_ext`: if this argument refers to the extension of the name for the original uploaded file, we build a set of sensitive extensions such as ".php", ".gif.php", ".mp3.php", ".zip.php", "pdf.php", and ".jpg.php".

- `$sym_file_name` or an argument with the string type: if this argument represents the extension-removed name of the uploaded file or its type is string, we leverage a PHP Fuzzer [34] as a drop-in fuzzer to mutate string values for `$sym_file_name`. We use different mutators in PHP Fuzzer [34] such as `EraseBytes`, `InsertByte`,

	System	LoCs	Fuzzing Templates	Detected by UFuzzer	Detected UChecker	Detected by RIPS	Detected by WAP	Detected by FUSE
Known Vulnerable	Adblock Blocker 0.0.1	369	2	✔ (0.26s)	✔ (0.50s)	✔ (0.01s)	✖ (0.58s)	✖ (1.53h)
	Audio Record 1.0	342	1	✔ (0.29s)	✔ (0.53s)	✔ (0.01s)	✖ (0.39s)	✖ (1.28h)
	Baggagefreight Shipping 0.1.0	5581	1	✔ (0.78s)	✔ (1.12s)	✔ (0.10s)	✔ (1.00s)	✔ (1.38h)
	Estatik 2.2.5	9823	3	✔ (0.89s)	✔ (1.72s)	✔ (0.31s)	✖ (1.00s)	✖ (2.17h)
	File Provider 1.2.3	983	1	✔ (0.24s)	✔ (0.40s)	✔ (0.02s)	✖ (0.65s)	✖ (1.85h)
	Finale - WooCommerce ... Timer 2.8.0	28643	6	✔ (4.91s)	✔ (5.01s)	✔ (0.42s)	✖ (1.00s)	✔ (0.33h)
	N-Media Website ... Uploader 1.3.4	1857	14	✔ (0.26s)	✔ (1.23s)	✔ (0.06s)	✖ (0.624s)	✔ (1.38h)
	Image Gallery with Slideshow 1.5.2	569	2	✔ (0.34s)	✔ (0.35s)	✔ (0.04s)	✔ (0.94s)	✖ (4.35h)
	Open Flash Chart Core 0.4	2337	2	✔ (0.35s)	✔ (0.70s)	✔ (0.10s)	✔(0.94s)	✖ (1.80h)
	PDW Media File Browser 1.1	20664	1	✔ (3.59s)	✖ (4.01s)	✔ (3.68s)	✔ (1.00s)	✖ (3.03h)
	Ip Blocker Lite 10.2	5574	1	✔ (1.46s)	✔ (0.99s)	✔ (0.05s)	✖ (0.73s)	✖ (1.53h)
	Uploadify 1.0.0	285	1	✔ (0.20s)	✔ (0.31s)	✔ (0.01s)	✔ (0.53s)	✖ (1.43h)
	WooCommerce Custom Profile Picture 1.0	138	16	✔ (0.15s)	✔ (0.28s)	✖ (0.01s)	✖ (0.44s)	✖ (1.55h)
	WooCommerce Catalog Enquiry 3.0.1	3560	8	✔ (0.67s)	✔ (1.21s)	✔ (0.09s)	✖ (1.00s)	✔ (0.33h)
	WooCommerce Checkout Manager 4.2.5	14942	8	✔ (1.70s)	✔ (0.96s)	✔ (0.70s)	✖ (1.00s)	✔ (0.33h)
	WP Marketplace 2.4.1	13956	1	✔ (2.6s)	✔ (2.60s)	✔ (0.32s)	✖ (1.00s)	✔ (0.33h)
	wp-Powerplaygallery 3.3	2752	416	✔ (1.33s)	✔ (2.78s)	✔ (0.07s)	✖ (0.86s)	✔ (2.21h)
	WP Seo Spy 3.1	3431	2	✔ (0.50s)	✔ (0.57s)	✔ (0.07s)	✔ (1.00s)	✔ (0.81h)
	WP Demo Buddy 1.0.2	2208	8	✔ (0.34s)	✔ (0.28s)	✔ (0.06s)	✖ (1.00s)	✖ (2.95h)
	Avatar Uploader 6.x-1.2	495	1	✔ (0.22s)	✔ (52.74s)	✔ (0.01s)	✖ (0.54s)	N/A
	Foxypress 0.4.1.1-0.4.2.1	13358	64	✔ (1.79s)	✔ (2.98s)	✔ (0.30s)	✔ (2.00s)	N/A
	Asset Manager 0.2	3784	1	✔ (0.22s)	✔ (0.81s)	✔ (0.04s)	✖ (0.87s)	N/A
	Simple Ad Manager 2.5.94	1937	4	✔ (1.24s)	✔ (5.35s)	✔ (0.33s)	✔ (1.00s)	N/A
	SpamTask 1.3.6	3434	2	✔ (0.61s)	✔ (0.61s)	✔ (0.15s)	✔ (1.00s)	N/A
	MailCWP 1.100	43191	1	✔ (5.01s)	✔ (5.80s)	✔ (1.26s)	✔ (2.00s)	N/A
	Joomla-Bible-study 9.1.1	87626	16	✔ (13.70s)	✔ (13.72s)	✔ (1.31s)	✔ (1.00s)	N/A
	Cimy User Extra Fields 2.3.8		100000+	✖ (N/A)	✖ (N/A)	✔ (0.97s)	✖ (1.00s)	✔ (5.50h)
Non-Vulnerable	Fullscreen background slider 1.1	8324	2	✖ (7.09s)	✖ (0.91s)	✔ (0.01s)	✖ (0.62s)	✖ (1.80h)
	TinyPNG for WordPress 0.2	256	8	✖ (6.17s)	✖ (0.33s)	✖ (0.01s)	✖ (0.64s)	✖ (1.80h)
	Mobile AppWidget 1.2	2873	3	✖ (9.95s)	✖ (0.91s)	✔ (0.07s)	✖ (1.00s)	✖ (1.81h)
	BackupGuard 1.1.46	10509	6	✖ (14.07s)	✖ (3.01s)	✔ (7.79s)	✖ (1.00s)	✖ (4.28h)
	WooCommerce Catalog Enquiry 3.1.0	3545	2	✖ (6.88s)	✖ (1.09s)	✔ (0.09s)	✖ (1.00s)	✖ (1.43h)
	Telegram-chat 3.0.4	2665	4	✖ (5.76s)	✖ (0.67s)	✔ (0.06s)	✖ (0.88s)	✖ (1.83h)
	Just a simple popup 2.0.1	948	4	✖ (8.36s)	✖ (0.38s)	✔ (0.02s)	✖ (0.48s)	✖ (1.61h)
	Booster for WooCommerce 2.8.2	47689	40	✖ (190.81s)	✖ (25.81s)	✔ (39.46s)	✖ (14.00s)	✖ (1.41h)
	Morbits SMS 1.0	71787	1	✖ (175.31s)	✖ (27.00s)	✔ (11.84s)	✖ (2.00s)	✖ (1.81h)
	Eventer 0.1.0	377	2	✖ (1.70s)	✖ (2.01s)	✔ (0.02s)	✔ (0.70s)	✖ (1.68h)
	Customize Random Avatar 1.0.0	1254	2	✖ (7.73s)	✖ (0.94s)	✔ (0.02s)	✔ (0.70s)	✖ (1.73h)
	IntelliWidget Custom Post Types 1.1.1	903	2	✖ (5.79s)	✖ (0.40s)	✔ (0.02s)	✖ (0.47s)	✖ (1.61h)
	PHP Event Calendar 1.5	10730	1	✖ (11.95s)	✖ (1.41s)	✔ (1.34s)	✖ (1.00s)	N/A
Results	Detection Rate			26/27	25/27	26/27	10/27	9/27
	False Positive Rate			0/32	0/32	12/32	2/32	0/32

Table 5.1: Evaluation Results Using Ground-Truth-Available Data (✔ and ✖ refer to vulnerable and non-vulnerable, respectively). *UFuzzer* detects 26 out of 27 known vulnerable scripts with no false positives; it outperforms *UChecker*, *RIPS*, and *WAP*. (Reprinted from [23])

`ChangeByte`, `ChangeBit`, and `ChangeASCIIInt`. For each mutator, we iterate for 50 times with a length between 5 and 70.

- An argument with the boolean type (e.g., `$sym_is_*`): We enumerate both `True` and `False` values to this argument if it is a boolean type.
- An argument with the number type: We enumerate values in $[-20, 20]$ for an argument if it has the number type (e.g., integer, float, and double).

We execute each fuzzing template by iterating over arguments' different values. If any execution returns `True`, we will cease the iteration and report that the PHP script is vulnerable. If no template returns `True` after all mutated values are exhausted, *UFuzzer* will stop and report this web application as non-vulnerable. We admit that the selection of these fuzzing parameters in *UFuzzer*, similar to that in other fuzzers, are empirical rather than provable. Nevertheless, they are highly configurable to support practical deployment.

5.3 Evaluation

5.3.1 Ground-Truth-Available Evaluation

We have implemented *UFuzzer* with approximately 28K LoC, which reuses the AST-based symbolic interpreter in [22] with minor improvements. We leveraged PHP-Parser [35] for AST construction and pretty printing. *UFuzzer* is deployed on an Ubuntu 18.04 LTS 64-bits operating system with AMD Fx-8350 CPU, 16 GB of memory, and PHP 7.4. As our data contains a large set of real-world, open-source plugins collected from `WordPress`, we install `WordPress` libraries in our running environment.

Data: We have collected totally 27 publicly-reported vulnerable PHP applications. This 27 samples consists of 13 known samples used in [22], 3 new vulnerable applications detected by UChecker [22], and 11 more vulnerable samples we have recently collected.

We have identified totally 32 vulnerability-free server-side PHP applications that *support file upload capabilities*. It is worth noting that it is a labor-intensive process to collect publicly-available, real-world samples and verify whether they are vulnerable. Such challenge is mainly attributed to the diversity of server-side applications, their highly-customized interfaces, and the high complexity.

Tools for Comparison: We have compared *UFuzzer* with *UChecker* [22], *RIPS* [16, 14], *WAP* [28] and *FUSE* [25]. We have deployed them in the same running environment of *UFuzzer*.

Evaluation Results: Table 6.3 presents the detection results and the running time for *UFuzzer*, *UChecker*, *RIPS*, *WAP*, and *FUSE*. The second and third columns in Table 6.3 present the names and lines of code (LoC) for each sample, respectively. The fourth column presents the number of fuzzing templates generated by *UFuzzer*. The remaining columns demonstrate the detection result and the running time for each program, where ✔ and ✖ stand for "detected" and "undetected", respectively. The last two rows of Table 6.3 summarize the detection rates and the false positive rates.

UFuzzer is both effective and efficient on the ground-truth available dataset. It detects 26 vulnerable samples out of 27 without incurring any false positive. The running time is mostly within one second for vulnerable cases and within one minute for non-vulnerable ones. *UFuzzer* fails to detect *Cimy User Extra Fields 2.3.8* since its underlying symbolic interpreter crashes due to path explosion.

Compared with *UChecker*, *UFuzzer* accomplishes a comparable detection rate (i.e., 26/27 of *UFuzzer* v.s. 25/27 of *UChecker*, with 0 false positive for both). Although *UFuzzer* outperforms *UChecker* by detecting only one more vulnerable sample (i.e., `PDW Media File Browser 1.1`), this single sample alone is significant to demonstrate how *UFuzzer* addresses the intrinsic limitation faced by *UChecker*. Specifically, this vulnerable application employs a regular expression operation as presented in Listing 5.4, which is challenging to be effectively modeled and solved by satisfiability solvers (and hence

UChecker). In contrast, *UFuzzer* can easily execute such operation in a native PHP runtime environment.

```
1 <php?
2 $valid_chars_regex = '.A-Z0-9_ !@#$%^&()+={}\[\]\',¬`-';
3 $file_name = preg_replace('/[^'.$valid_chars_regex.']|\.+$/i', "", ...
    basename($_FILES[$upload_name]['name']));
4 /*...*/
5 if (!@move_uploaded_file(
6           $_FILES[$upload_name]["tmp_name"],
7           $save_path . $file_name)) {
8    /*...*/}
9 ?>
```

Listing 5.4: *UChecker* fails to correctly model the regex operation in the PDW Media File Browser plugin (Reprinted from [23])

Although both *RIPS* and *WAP* accomplish comparable efficiency with *UFuzzer*, they have lower detection performances. Specifically, *RIPS* suffers from a high false positive rate of 12/32. *WAP* demonstrates a low detection rate of 10/27.

FUSE has accomplished a lower detection rate of 9/27 and the same false positive rate of 0/32. In addition, *FUSE* requires significantly longer time for detection, typically around a few hours. It is also worthy noting that it has taken an excessive amount of manual efforts to make each PHP application fully operable, which is unfortunately required by *FUSE*. When an application fails to function, *FUSE* will miss the opportunity to perform detection. Samples annotated with "N/A" in Table 6.3 represent such cases. Despite our best efforts, these samples failed to operate in our evaluation environment, mainly because of missed files or unknown configuration problems. Such evaluation results imply that *FUSE* has limited applicability for large-scale vulnerability scanning.

No.	Application	D. by UFuzzer	VulnSrcFile : Line No.	D. by UChecker	Verification Method	Root Cause	Adm Req?
1	Basic-Laravel-CMS - PHP Framework For Web Artisans	✔	uploader.php:31	✔	Code Review	LS	No
2	BloggerCMS - Easiest Static Blog Generator	✔	Image.php:77	✖	Code Review	SInS	No
3	Lapin_CMS - Slim 3 RAD Skeleton	✔	upload.php:36	✖	Code Review	SInS	No
4	Learningphp-CMS	✔	upload.php:41	✔	Code Review	LS	No
5	Mini_CMS - PHP Based Mini Blog	✔	zamiesc-post.php:40	✔	Exploiting	SInS	No
6	laravelCMS - PHP Framework For Web Artisans	✔	ProfileController.php:29	✔	Code Review	SInS	No
7	WikiDocs - Databaseless Markdown Wiki Engine	✔	submit.php:264	✖	Code Review	SInS	No
8	Buffalo-Webpage-CMS	✔	actionProductoCtrl.php:81	✖	Code Review	SInS	No
9	LCMS - College Website with CMS	✔	student_avatar.php:13	✔	Code Review	LS	No
10	Palette - PHP Based Site Builder	✔	upload.php:27	✖	Code Review	LS	No
11	Progress_Business - CMS for Company Profile Web	✔	adding_news.php:12	✔	Exploiting	LS	No
12	publisher.mod - FlatCore CMS Module	✔	upload.php:29	✖	Code Review	SInS	No
13	User-Management-PHP-MYSQL	✔	edit-user.php:32	✔	Exploiting	SC	No
14	MicroCMS1 - CMS Based On Model-View-Controller	✔	uploads.php:31	✖	Code Review	LS	No
15	BlogStop - Simple Content Management System	✔	admin_edit_post.php:22	✖	Exploiting	LS	Yes
16	CMS-Blogging-System - Blog Made with PHP and MySQL	✔	add_post.php:15	✔	Code Review	LS	Yes
17	Cmsphp - Simple PHP based CMS System	✔	add_post.php:21	✔	Code Review	LS	Yes
18	CMSPortfolio - PHP based Portfolio Template	✔	func.php:464	✖	Code Review	SInS	Yes
19	CMSProjectPHP	✔	add_post.php:16	✔	Code Review	LS	Yes
20	CMSsite - Simple CMS Site	✔	profile.php:27	✖	Exploiting	LS	Yes
21	CmsV1 - CMS Based on PHP	✔	add_user.php:21	✔	Code Review	LS	Yes
22	N5 Upload Form 1.0	✔	n5uploadform.php:156	✖	Exploiting	LS	No
23	Testimonials King Light 0.1	✔	testimonial-king-form.php:38	✖	Exploiting	MisAPI	No
24	WP-Curriculo Vitae Free 6.1	✔	enviarCadastro.php:86	✖	Exploiting	LS	No
25	Easy Form Builder 1.0	✔	newForm.php:49	✔	Exploiting	LS	No
26	imagements 1.2.5	✔	imagements.php:127	✔	Exploiting	SInS	No
27	Event Banner 1.3	✔	admin_events.php:29	✖	Exploiting	LS	Yes
28	Quick Image Transform 1.0.1	✔	file-upload.php:79	✖	Exploiting	SC	Yes
29	College Publisher Import 0.1	✔	college-publisher-import.php:144	✖	Exploiting	LS	Yes
30	BSK Files Manager 1.0.0	✔	bsk-files-manager.php:269	✖	Exploiting	MisAPI	Yes
31	Banner Cycler 1.4	✔	admin.php:167	✖	Exploiting	LS	Yes
32	Gallerio 1.0.1	✖	gallerio.php:610	✔	Exploiting	LS	Yes

Table 5.2: Detecting New Vulnerable Applications. *UFuzzer* detected 30 vulnerable PHP applications that have not been previously reported, where 1-21 are from GitHub and 22-32 are `WordPress` plugins. Each vulnerability is verified through either exploiting or thorough code review. The root cause of each vulnerable sample has also been labeled, where LS for "lacking sanitization", MisAPI for "misusing sanitization APIs", SInS for "sanitizing incorrect sources", and SC for "sanitizing at the client". (Reprinted from [23])

5.3.2 Detecting New Vulnerable PHP Applications

We have used *UFuzzer* to detect PHP applications with unrestricted file upload vulnerabilities. We leverage two repositories, including `WordPress` plugins and GitHub, which both offer a large number of PHP-based, open-source applications. We collected 9,157 `WordPress` plugins in a reverse chronological order (starting from 4/22/2018) based on their last updated time. We also retrieved source code of top 900 highly rated (i.e., "start"-ed) PHP content management systems (CMS) from GitHub (on 07/01/2020). Since *UChecker* achieved comparable detection performance on the ground-truth-available data, we also use it to scan all these applications.

Table 5.2 presents the detection results. *UFuzzer* and *UChecker* together detect totally 32 vulnerable applications. The first 21 are from GitHub and the remaining 11 are `WordPress` plugins. We have confirmed all of them allow the uploading of PHP files through i) actual exploiting or ii) code review. The "verification method" column in the table presents how each application is verified. Specifically, 16 out of 32 applications can be installed and we have successfully exploited their file uploading vulnerabilities. The remaining 16 applications fail to operate mainly due to the lack of configuration files (e.g., required database configuration files are missing). We therefore manually review their source code thoroughly. All of these 32 vulnerable applications have not been previously reported to the best of our knowledge.

Among these 32 vulnerable applications, *UFuzzer* detects 31 whereas *UChecker* only detects 15. *UFuzzer* also effectively identify the source of each vulnerable application, i.e., the file name and the line no. of the vulnerable statement (see the 4th column of Table 5.2). Manual analysis of *UChecker*'s false negatives reveals that some of their APIs are not currently covered in its PHP-to-Z3 translation rules. Comparatively, *UFuzzer* executes these PHP APIs in native PHP runtime environment. *UFuzzer* misses one vulnerable sample since *UFuzzer* was not successful in mutating inputs that satisfy their reachability

conditions. For example, the fuzzing template of `Gallerio 1.0.1` has the reachability condition of `$reach_reach = ($sym_Isset_POST_doadd and $_POST_doadd_symbol == 'yes' and $sym_file_name . $sym_file_ext != '')` and the string mutator fails to generate `'yes'` for the free variable `$_POST_doadd_symbol`.

Among these 32 vulnerable applications, our manual investigation reveals that 13 samples need administration privilege for successful exploitation. While these 13 applications expose less risks since they require attackers to gain higher privileges, they may still lead to unintended behaviors that could be potentially misused. We agree with *FUSE* [25] that mature web services should limit the uploading capability even for administrators through web interfaces while use other channels such as SFTP or SCP for script uploading. Therefore, using the same criteria in [25], we identify these 12 samples as potentially exploitable.

We attribute root causes of these 32 new vulnerable applications into four categories including *lacking sanitization* (LS), *misusing sanitization APIs* (MisAPI), *sanitizing incorrect sources* (SInS), and *sanitizing at the client* (SC). To further illustrate each root cause, we present a few representative cases below.

Lacking Sanitization: `Basic Laravel CMS` is a content management system (CMS) based on the Laravel framework. Its vulnerable code is presented in Listing 5.5, which does not check the extension of the uploaded file. Although the developer attempts to randomize the name of the saved file, the random number is derived from a very narrow range and therefore highly predictable. Listing 5.6 shows the fuzzing template that successfully detects this vulnerability. This template evaluates both the reachability condition and the file extension.

```
1 <?php
2 if ($_FILES['file']['name']) {
3     if (!$_FILES['file']['error']) {
4         $name = md5(rand(100, 200));
5         $ext = explode('.', $_FILES['file']['name']);
6         $filename = $name . '.' . $ext[1];
```

```
7         $destination = '/public/images/' . $filename;
8         $location = $_FILES["file"]["tmp_name"];
9         move_uploaded_file($location, $destination);
10        //...
11    } else{
12      //...
13    }
14 }
15 ?>
```

Listing 5.5: Vulnerable Code of `Basic Laravel CMS` (Reprinted from [23])

```
1 function is_Vulnerable_0(string $sym_file_name,
2                           string $sym_file_ext,
3                           int $_FILES_file_error_symbol){
4     $exp_reach = ($sym_file_name . $sym_file_ext and
5                   !$_FILES_file_error_symbol);
6     $funCall = explode('.', $sym_file_name . $sym_file_ext);
7     $exp_filename = '/public/images/' . (md5(rand(100, 200)) .
8                     '.' . $funCall[1]);
9     if ($exp_reach) {
10        $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
11        if (in_array($ext, array('php'))) {
12            return true;
13        }
14    }
15 }
```

Listing 5.6: A Fuzzing Template for `Basic Laravel CMS` (Reprinted from [23])

Misusing Sanitization APIs: `Testimonial King Light 0.1` [18] is a `WordPress` plugin. This plugin intends to support the upload of images such as ".jpg", ".png", and ".gif". Listing 5.7 presents the vulnerable code snippet. We suspect this vulnerability is rooted in the mis-interpretation of the "`sanitize_file_name()`", a WordPress built-in API. This function is for removing special illegal characters rather than guaranteing to return a filename that is allowed to be uploaded [59]. Therefore, a file with the executable ".php" extension can still be uploaded. The fuzzing template that successfully revealed this vulnerability was presented in Listing 5.8; it only concerns the name of the file to be saved since the

reachability constraint is not tainted by the name of the uploaded file.

```
1 function gmctk_testimonial_form(){
2     //...
3     $hasError = false;
4     if (isset($_POST['title']) && trim($_POST['title']) === '')
5         $hasError = true;
6     if (isset($_POST['description']) && trim($_POST['description']) ...
        === '')
7         $hasError = true;
8     if (!$hasError) {
9         if (isset($_FILES['userpic'])) {
10            $uploadfile = wp_upload_dir() . sanitize_file_name(
11                $_FILES['userpic']['name']) );
12            move_uploaded_file(
13                $_FILES['userpic']['tmp_name'], $uploadfile);//...
14        }
15    }//...
16 }
```

Listing 5.7: Vulnerable Code in `Testimonial King Light 0.1` (Reprinted from [23])

```
1 function is_vulnerable($sym_file_name, $sym_file_ext)
2 {
3     $funCall_0 = wp_upload_dir();
4     $exp_fileName = $funCall_0['path'] . '/' .
5                 basename(sanitize_file_name($sym_file_name
6                     . $sym_file_ext));
7     $ext = pathinfo($exp_fileName, PATHINFO_EXTENSION);
8     if (in_array($ext, array('php'))) {
9         return TRUE;
10    }
11    return FALSE;
12 }
```

Listing 5.8: A Fuzzing Template for `Testimonial King Light 0.1` (Reprinted from [23])

Sanitizing Incorrect Sources: `Imagements 1.2.5` [58] is a `WordPress` plugin that supports a visitor to leave an image-based comment in users' blogs. Listing 5.9 presents the vulnerable code snippet. The function "add_action()" is a `WordPress` built-in API to

bind a function (“imagments_formverwerking()” in this case) with an action(i.e. “comment_post” in this case). The “imagments_formverwerking()” function intends to permanently store an uploaded file in the server. The developer seems aware of this type of vulnerabilities and use a filter (see the “add_filter()” function) to abort any uploading action if it submits a non-image file. Unfortunately, the added filter is flawed. It investigates $_FILES[‘image’][‘type’], which is the type of the file derived from client’s request. Since an attacker have full control of client-side software (e.g., the browser), she can upload a PHP executable script and meanwhile instrument the browser to change $_FILES[‘image’][‘type’] to “image/png”, successfully bypassing this filter. The fuzzing template that successfully reveals this vulnerability was illustrated in Listing 5.10.

```
1 add_action('comment_post', 'imagements_formverwerking');
2 add_filter('preprocess_comment', 'imagements_verify_post_data');
3 function imagements_formverwerking(){
4     if(isset($_POST['checkbox'])){
5         $name = $_FILES['image']['name'];
6         //...
7         move_uploaded_file($_FILES["image"]["tmp_name"],
8                            PATH . '/images/' . $name);
9     }
10 }
11 function imagements_verify_post_data($commentdata){
12     if(isset($_POST['checkbox'])) {
13         if($_FILES['image']['name'] != null) {
14             if($_FILES["file"]["error"] > 0) {
15                 //...
16             } else {
17                 if(!($_FILES['image']['type'] == 'image/x-png'||
18                      $_FILES['image']['type'] == 'image/pjpeg'||
19                      $_FILES['image']['type'] == 'image/jpeg' ||
20                      $_FILES['image']['type'] == 'image/jpg'  ||
21                      $_FILES['image']['type'] == 'image/png')) {
22                     wp_die('this file is no image...');
23                 }
24             }
25         }
26         //...
```

```
27     }
28 }
```

Listing 5.9: Vulnerable Code in `Imagements` 1.2.5 (Reprinted from [23])

```
1 function is_Vulnerable(string $sym_const_PATH,
2                         string $sym_file_name,
3                         string $sym_file_ext){
4     $exp_filename = $sym_const_PATH . '/images/' .
5                     ($sym_file_name . $sym_file_ext);
6     $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
7     if (in_array($ext, array('php'))) {
8         return true;
9     }
10    return false;
11 }
```

Listing 5.10: A Fuzzing Template for `Imagements` 1.2.5 (Reprinted from [23])

Sanitizing at the Client: `User-Management-PHP-MYSQL` [37] is an open-source web-application collected from GitHub. Listing 5.11 presents the client-side HTML page and the server-side PHP script to process the file submission request. The developer seems aware of this vulnerability and implemented the validation function at the client using JavaScript (i.e., by only allowing files with "jpg" or "jpeg" extensions). Unfortunately, since an attacker has full control of her browser, she can either disable this validation function or manipulate the file name carried by the POST request. Listing 5.12 presents the fuzzing template that reveals this vulnerability.

```
1 <?php
2 if(isset($_POST['submit'])){
3     $file = $_FILES['image']['name'];
4     $final_file = str_replace(' ', '-', strtolower($file));
5     if (move_uploaded_file($_FILES['image']['tmp_name'],
6                            "images/" . $final_file)) {
7         //...
8     }
9 }
```

```
10 ?>
11 <!doctype html>
12 <html lang="en" class="no-js">
13 <!--...-->
14 function validate() {
15     var extensions = new Array("jpg", "jpeg");
16     var final_ext = // Get the file extensions by JavaScript
17     // return true if the final_ext is in extensions
18     // return false otherwise
19 }
20 <form method="post" <!--...--> onSubmit="return validate();">
21     <!--...-->
22     <button <!--...--> type="submit">Register</button>
23 </form>
24 <!--...-->
25 </html>
```

Listing 5.11: Vulnerable Code in `User-Management-PHP-MYSQL` (Reprinted from [23])

```
1 function is_Vulnerable_0(string $sym_file_name, string $sym_file_ext){
2     $exp_filename = '../images/' . str_replace(' ', '-', ...
          strtolower($sym_file_name . $sym_file_ext));
3     $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
4     if (in_array($ext, array('php'))) {
5         return true;
6     }
7     return false;
8 }
```

Listing 5.12: A Fuzzing Template for `User-Management-PHP-MYSQL` (Reprinted from [23])

5.4 Discussion

Incomplete Runtime Modeling: UFuzzer currently does not model system configuration information. For example, it ignores `.htaccess` or `php.ini`, which may have already disabled file uploading for the entire system. Nevertheless, we believe the detected vulnerability in the web application, which could be disabled at runtime through system configu-

ration, is highly informative.

Incomplete Interpretation of OOP and Loops: The current implementation of *UFuzzer* does not interpret all PHP grammars. Although it interprets essential OOP grammars including class declaration, object initialization, and the invocation of member functions, *UFuzzer* does not handle other OOP features such as inheritance, function overriding, and deserialization. Since *UFuzzer* directly interprets AST, it takes little efforts to *represent* a loop statement in a heap graph. However, significant challenges arise when one intends to *execute* a loop-included fuzzing template with mutated inputs. Specifically, it would be difficult to identify reasonable variable values to exercise a generated loop effectively and efficiently. Therefore, we skip the process of loops in the current implementation.

The practical impact of such incomplete interpretation is however alleviated by the "locality analysis" of *UFuzzer*'s interpreter, which inherits from *UChecker*. Specifically, we first use "locality analysis" to identify statements that are likely to be relevant to the vulnerability and then only symbolically interprets these identified statements. These identified statements, which usually represent a very small portion of the entire program, rarely contain advanced OOP features and loops in our evaluated and scanned cases. This suggests a limited impact of our current implementation.

Nevertheless, extending *UFuzzer*'s interpretation capability could enhance its detection capabilities. For example, we can track the relationship of classes when they are declared to interpret OOP polymorphism features. We can execute a loop-included fuzzing template through massive parallelization. Such solutions fall into our future work, and our plan to open *UFuzzer*'s code will also facilitate community's efforts towards this direction.

Specific to PHP: UFuzzer is specific to the ".php" extension. We are aware that files with other extensions such as ".asa" and ".swf" are also potential harmful. Nevertheless, *UFuzzer* can be easily extended to cover additional file extensions. Meanwhile, *UFuzzer*'s design is applicable to web services written in other languages.

Focusing on the File Extension: UFuzzer investigates whether an uploaded file could have

the ".php" extension, which represent an immediate, arguably the most significant and common exploitation of the studied vulnerability. It might also be risky if one can submit a file of executable content without executable extension to a web system. However, it requires additional exploitations to actually execute the file (e.g., altering file names via other interfaces). Nevertheless, we acknowledge that it will enhance *UFuzzer* by analyzing the executable content in an uploaded file.

Reusable Inputs: UFuzzer "fuzzes" a fuzzing template that approximates the original code rather than original code itself. Therefore, an input that successfully "exploits" the fuzzing template is unlikely to be directly reused to reproduce the exploitation in the operating web system corresponding to this fuzzing template. This is indeed different from conventional fuzzers. Nevertheless, such difference represents *UFuzzer*'s design that trades precision for efficiency rather than a flaw.

False Negatives: UFuzzer introduces one false negative in detecting new vulnerable applications. This is mainly because the mutation process is unguided and therefore it cannot guarantee the generation of values that provably satisfy certain conditions in the fuzzing template. One potential solution is to integrate *UFuzzer* and *UChecker*, where we can infer values for applicable variables using a solver and mutate values for the remaining. This falls into our future work.

5.5 Summary

We have built *UFuzzer* to automatically detect PHP-based web programs with unrestricted file upload vulnerabilities.

UFuzzer models a server-side PHP web application using heap graphs and automatically identifies sub-graphs that are relevant to a vulnerability. Identified sub-graphs are refactored and eventually converted into executable PHP programs for fuzzing. The evaluation results based on real-world PHP applications demonstrated *UFuzzer*'s high detection

performance. It also detects 31 new vulnerable services that have not been publicly reported.

Chapter 6: Mining Vulnerabilities in PHP-Based Web Programs Using Graph Models – UGraph

6.1 Motivation

Vulnerable web servers fundamentally undermine Internet security as they often expose critical infrastructures and sensitive information behind them to potential risks. Unfortunately, the development of vulnerability detection systems significantly lags behind the reveal of web vulnerabilities. Such gap can be attributed to facts. First, modern server-side web programs commonly experience a high level of syntactical variances. Second, typical program representations such as abstract syntax tree (AST) and the intermediate representation (IR) are compiler-oriented, lacking sufficient information to support essential security-oriented analysis such as information flow analysis and static taint analysis. As a result, vulnerability detection systems are usually forced to address these challenges as part of their design, introducing huge but unwanted efforts.

This chapter presents our solution towards enabling the rapid development of server-side vulnerability detection systems by designing a new framework. The core of this framework is a novel program model, namely *dependency graphs*, with the following design

objectives:

- Security-Oriented: it characterizes information flow (i.e., both explicit and implicit data flows) in a studied program that is essential for security analysis of a program.
- Intuitive: It facilitates efficient and intuitive implementation of security applications.
- Resilient to Syntax Variances: it focuses on program semantics and abstracts away program syntax.
- Scalable: it enables a computationally-efficient model generation process for large programs.

We therefore design our model in the form of a graph to characterize *immediate* data and control dependency among all objects generated by the program under analysis. As a consequence, information flow between an arbitrary pair of objects can be verified by analyzing the graph. Since graph-based operations are natively supported by of-the-shelf graph databases, the information flow analysis can then be translated into efficient and intuitive graph queries. A dependency graph is generated by symbolically interpreting a server-side web program with flow- and context-sensitivity, where the path sensitivity is traded for scalability to avoid path explosion.

In order to illustrate dependency graph and demonstrate its usage in security analysis of server-side web programs, we have built a system, namely *UGraph* to detect the PHP-based program vulnerability. *UGraph* generates dependency graphs by directly interpreting the ASTs of web programs, which makes it possible to precisely map the model back to the source code, greatly facilitating further exploration, debugging, and patching if needed. *UGraph* implements one security applications to detect unrestricted file upload vulnerabilities. *UGraph* currently focuses on web programs written in PHP, one of the most popular programming languages for server-side web development; it also leverages the `neo4j` [29] graph database to support the storage and analysis of generated models. By integrating

heap graphs and off-the-shelf graph databases, *UGraph* illustrates security applications for server-side web programs can be effectively, efficiently, and intuitively developed.

We have applied *UGraph* to scan 15,154 real-world PHP-based web applications. *UGraph* has successfully detected totally 14 vulnerable applications that have not been previously reported. We have also received 6 CVE confirmations. To summarize, we have made the following contributions:

- We have designed a novel, graph-based program model, namely *dependency graph*, to enable efficient and intuitive information flow analysis for server-side web programs.
- We have designed an interpreter to generate a dependency graph of a PHP-based web program by performing flow- and context-sensitive symbolic interpretation directly using the AST of this program.
- We have built a security applications to detect the vulnerability of unrestricted file upload in web application
- We have employed our solutions to detect 16 vulnerable services with 3 CVE confirmed.

The remains of this chapter will present the design, generation, and applications of this novel model in the context of the *UGraph* system. Specifically, it presents the system overview in Section 6.2, the definition of the dependency graph in Section 6.3, its generation using AST-based symbolic interpretation in Section 6.4, security applications (in Section 6.5), and the detection results in Section 6.6. We also discussed the related work in Section 6.7 as will as possible limitations and solutions in Section 6.8. Section 6.9 summarize the system.

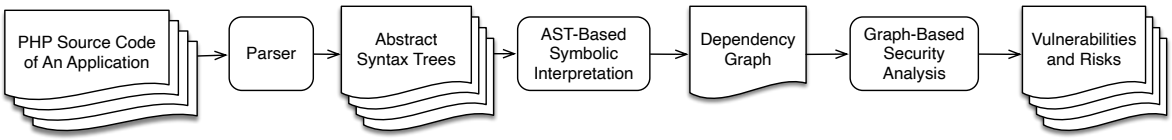

Figure 6.1: The architectural overview of the *UGraph*

6.2 System Design

Figure 6.1 presents the architectural overview of *UGraph*, which has 3 major phases.

- **Parsing**: The input of *UGraph* is a set of PHP files for a web application and our *UGraph* parses them to generate AST(s).
- **AST-Based Symbolic Interpretation**: *UGraph* next symbolically interprets AST(s) to produce the dependency graph.
- **Security Analysis**: *UGraph* stores the dependency graph into a graph database and leverages its interfaces to build security applications.

We leverage PHP-Parser [35], a popular PHP parser, to parse all source code files of a server-side application and generate a collection of abstract syntax trees. Our symbolic interpreter needs to address the path explosion challenge that is inherent to symbolic execution. We employed employs `neo4j` [29], a graph database, to store generated dependency graphs. Our security applications can then be implemented as `neo4j` declarative queries.

6.3 Dependency Graph

6.3.1 Definition

A dependency graph aims at modeling *immediate* data and control dependency among objects, where an object refers to the evaluation result of an expression in a program. Specifi-

cally, we define a *dependency graph* as $G = \{C, S, N_{Var}, N_{Func}, FUNC, OP, AUX, L, O_C, O_S, O_{FUNC}, O_{OP}, O_{AUX}, Edge, Env_{Var}, Env_{Func}\}$:

- C is a set of concrete values.
- S is a set of symbolic values.
- N_{Var} is a set of variable names.
- N_{Func} is a set of names of PHP built-in functions augmented by indexes.
- $FUNC$ is a set of all PHP built-in functions.
- OP is a set of all PHP operators (e.g., unary and binary operators such as "+", "-", and ".").
- AUX is a set of operators introduced for graph building. $AUX = \{comb, array\_access\}$, where $comb$ aggregates values through different paths and $array\_access$ indicates the access of an array.
- I is a set of IDs for nodes in this graph.
- L is a set of labels for edged in this graph, where L contains "$data$", "$ctrl$", and annotations of arguments and operands.
- $O_C \subset I \times C$ is a set of objects (i.e., nodes) for concrete values, where each object in O_C is assigned with a *unique* id.
- $O_S \subset I \times S$ is a set of objects (i.e., nodes) for symbolic values, where each object in O_C is assigned with a *unique* id.
- $O_{FUNC} \subset I \times Func$ is a set of objects (i.e., nodes) for built-in functions, where each node is assigned with a *unique* id.

- $O_{OP} \subset I \times OP$ is a set of objects (i.e., nodes) for operations, where each node is assigned a *unique* id.
- $O_{AUX} \subset I \times AUX$ is a set of objects (i.e., nodes) for the newly introduced operators, where each node is assigned with a *unique* id.
- $Edge \subset \{(id_1, id_2, l) | (id_1, x) \in O_{FUNC} \cup O_{OP} \cup O_{AUX}\ and\ (id_2, y) \in O_C \cup O_S \cup O_{FUNC} \cup O_{OP} \cup O_{AUX}\ and\ l \in L\}$. Edges are directed and each one connects a node for a PHP built-in function, a PHP operator, and a AUX node with another node with an arbitrary type. If the source node of an edge is an object of an operator, its destination node is an operand; if the source node of an edge is for a built-in function, its destination node is an argument for this function.
- $Env_{Var} \subset N_{Var} \times I$. It establishes a mapping between a variable name and an object.
- $Env_{Func} \subset N_{Func} \times I$. It establishes a mapping between an augmented function name and an object.

6.3.2 An Example Dependency Graph

In order to illustrate the heap graph and environments, we use an example presented in Listing 6.1. This program has four variables including `$a`, `$b`, `$c`, and `$d`. Both `$a` and `$b` are initialized with values from external inputs. The variable `$c` is bound to one of three possible values, depending on the evaluation result of two expressions including "`(abs($a)) > 222`" and "`$b > 333`". The variable `$d` is derived from `$c`. The "`abs($a)`" function invocation will always be reachable. Comparatively, the reachability of "`pow($b, 2)`" also depends on the evaluation of these two expressions including "`abs($a) > 22`" and "`$b > 333`". Similarly, the reachability of "`round(99)`" depends on the evaluation of the expression of "`abs($a)> 22`".

```php
<?php
    $a = $_GET['A'];
    $b = $_GET['B'];
    if(abs($a) > 222) {
        if($b > 333) {
            $c = pow($b, 2);
        }
        else {
        $c = 555;
        }
    }
    else {
    $c = round(99)
    }
    $d = $c + 666;
    echo $d;
?>
```

Listing 6.1: Sample code with data and control dependency

Figure 6.2 presents the dependency graph for the example in Listing 6.1, which is shown as follows with $Edge$ omitted for brevity:

- $C = \{2, 99, 222, 333, 555, 666, \text{"A"}, \text{"B"}\}$
- $S = \{sym\_GET\}$
- $N_{Var} = \{\$a, \$b, \$c, \$d\}$
- $N_{Func} = \{abs_1, pow_1, round_1, echo_1\}$
- $FUNC = \{abs, pow, round, echo\}$
- $OP = \{+, -, >\}$
- $AUX = \{comb, array\_access\}$
- $I = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23\}$
- $L = \{array,\ index,\ ctrl,\ data,\ arg,\ left,\ right\}$

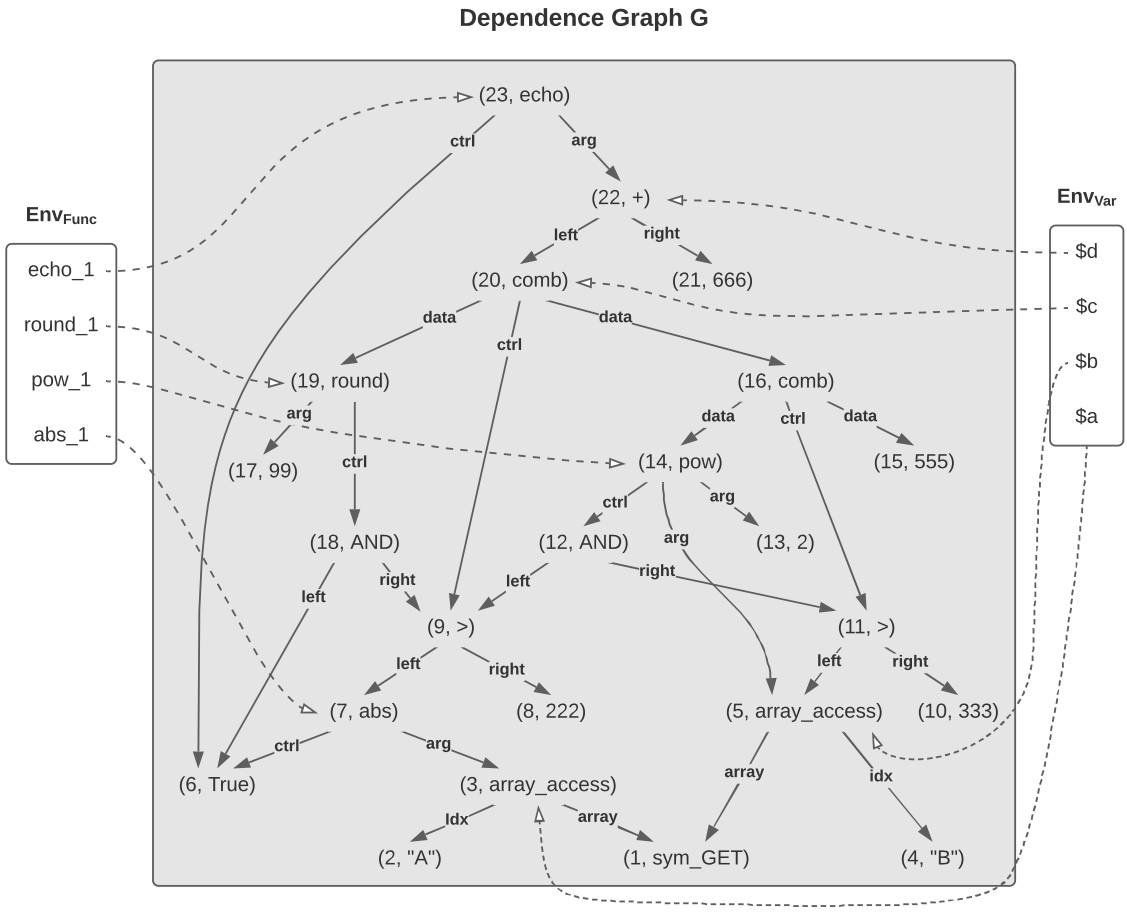

Figure 6.2: The dependency graph for the sample code in Listing 6.1

- $O_C = \{(2, \text{"A"}), (4, \text{"B"}), (8, 222), (10, 333), (14, 2), (15, 555), (17, 99), (21, 666)\}$
- $O_S = \{(1, sym\_GET)\}$
- $O_{FUNC} = \{(7, abs), (13, pow), (19, round), (23, echo)\}$
- $O_{OP} = \{(9, >), (11, >), (22, +)\}$
- $O_{AUX} = \{(3, array\_access), (5, array\_access), (16, comb), (20, comb)\}$
- $Env_{Var} = \{(\$\_GET, 1), (\$a, 3), (\$b, 5), (\$c, 20), (\$d, 22)\}$
- $Env_{Func} = \{(abs\_1, 7), (pow\_1, 13), (round\_1, 19), (echo\_1, 23)\}$

In this dependency graph, we use distinct integers as node IDs (i.e., integers in the set of I). Each superglobal variable is pre-assigned with a symbolic value. In this particular example, a node of $(1, sym\_GET)$ is created to represent the symbolic value of `$_GET`.

`$_GET['A']` is to access an array named "`$_GET`" using the index named 'A'. We therefore create an "array_accsss" node, i.e., (3, array_access), to indicate the array access. The edge (3, 1, array) means that this array access using the node with ID of "1" as the array; The edge (3, 2, index) means that this array access using the node with ID of "2" as the index. Since `$_GET['A']` is then assigned to the variable `$a`, we create an association of ($a, 3) in Env_{Var} to indicate that `$a` is bound to the object with the ID of "3". Following the same procedure, we create two nodes with IDs of "4", "5", two edges of (5,1, array) and (5,4, index), and an association of ($b, 5).

The value of a variable could depend on the execution path of this program. We therefore introduce a node, namely "comb", to summarize all possible values of a variable (i.e., to characterize its data flows) and all conditions that decide the selection of these values (i.e., to characterize its control flows). Each "comb" node has two outgoing edges: the "$data$" and "$ctrl$", which indicate the *immediate* data and control dependencies, respectively.

The variable $c is such a variable. Depending on the evaluation results of two conditions, i.e., `abs($a) > 222` and `$b > 333`, the value of $c could be `pow($b, 2)`, `555`, or `round(99)`. Therefore, we introduce a "comb" node, i.e., (20, comb) to model $c's control and data dependencies in outer if-else. Specifically, the "$ctrl$" edge from (20, comb) indicates the control dependency; it points to the node (i.e., (9, ¿)) whose derived expression governs the outer if-else. There are two outgoing "$data$" edges from the (20, comb) node, indicating $c's value could come from two branches. . The first edge (i.e., (20, 19, data)) refers to $c's possible value from the "false" branch (i.e., from line 12 to 14), which is `round(99)`. The second edge (i.e., (20, 16, data)) refers to $c's possible value from the "true" branch, which is another "comb" node (i.e., (16, comb)). The node of (16, comb)

aggregates both the condition and $c's possible values from the inner if-else. Specifically, the "$ctrl$" edge (i.e., (16, 11, ctrl)) indicates that the condition for the inner if-else could be derived from the node $(11, >)$. The first "$data$" edge (i.e., (16, 14, data)) points to $c's value derived from the "true" branch of the inner loop (i.e., `pow($b, 2)`); the second "$data$" edge (i.e., (16, 15, data)) points to $c's value derived from the "true" branch of the inner loop (i.e., `555`).

The variable `$d` is bound with the evaluation result of `$c + 666`. We therefore create the node `(22, +)` for this "+" operator. Since `$c` is bound to the "comb" node of (20, comb), we introduce an edge of `(22, 20, left)` to indicate the left operand of (22, +); similarly, we create a constant node of `(21, 666)` and an edge `(22, 21, right)` to indicate the its right operand.

In addition, we have i) created an environment, namely Env_{Func} track the invocation of each built-in API and ii) augmented the node for an API invocation to track its reachability condition(s). Once the invocation of an API is evaluated, a new variable, whose name is a combination of the API name and a unique integer, will be added into Env_{Func}. An association will then be created between this newly created variable and the node corresponding to the invocation of this API. There are four API invocations in Listing 6.1, i.e., `abs()`, `pow()`, `round()`, and `echo`. Our dependency graph will hence have four variables in Env_{Func}, i.e., `abs_1`, `pow_1`, `round_1`, and `echo_1`. Each of these variables points to the evaluation result of its corresponding API. For example, `pow_1` points to the node of (14, pow), which represents the evaluation result of the expression, `pow($b, 2)` in line 6. Since `pow($b, 2)` has two arguments including `$b` and `2`, (14, pow) has two outgoing "arg" edges points to `$b`'s evaluation result (i.e., (5, array_access)) and a node of a constant (i.e., (13, 2)), respectively. It is worth noting that `pow($b, 2)`'s reachability is governed by two conditions including `abs($a) > 222` and `$b > 333`. Therefore, the node of (14, pow) is augmented by a "ctrl" edge; this edge points to the "AND"-based joint of expressions that govern its reachability. Specifically, the node of `(12, AND)` joins these two conditions, i.e.,

`(9, >)` and `(11, >)`, respectively.

6.3.3 Graph-Driven Information Flow Analysis

Information flow analysis [42], including both explicit and implicit flow analysis, is essential for language-based security analysis. Explicit flows generally refer to data flows (e.g., as a results of assignments and data copy) whereas implicit flows represent flows due to conditional statements. Flow graphs profoundly facilitates information flow analysis by translating it into intuitive graph analysis, which can further takes advantage of graph-oriented parallelized platforms for both interactivity and scalability. In this section, we will describe how explicit and implicit data flow analysis could be performed using a flow graph.

We consider two arbitrary nodes, namely p and q, in a flow graph G.

- There is an *information flow* from p to q if there exists a path from q to p.
- There is an *explicit data flow* from p to q if there exists at least one path from q to p that does not have any "$ctrl$" edge.
- There is an *implicit data flow* from p to q if there exists at least one path from q to p that has at least one "$ctrl$" edge.

For example, there is one explicit data flow from $(1, sym\_GET)$ to the node $(22, +)$ (i.e., $\$d$'s value) since there exists a path $22 \rightarrow 20 \rightarrow 16 \rightarrow 14 \rightarrow 5 \rightarrow 1$ where none of edges is labeled as "$ctrl$". This path essentially models that $\$d$'s value can be originally coming from the superglobal variable of `$_GET` (see line 3, 6, and 15 in Listing 6.1).

On the other hand, there are two implicit data flows from $(1, sym\_GET)$ to $(22, +)$ since there exist two paths that involve at least one "$ctrl$" edge. The first path is $22 \rightarrow 20 \rightarrow 9 \rightarrow 7 \rightarrow 3 \rightarrow 1$, where the edge $(20, 9, ctrl)$ is labeled as "$ctrl$"; the second path is $22 \rightarrow 20 \rightarrow 16 \rightarrow 11 \rightarrow 5 \rightarrow 1$, where the edge $(16, 11, ctrl)$ is labeled as "$ctrl$".

Following the same graph-based analysis, we can easily conclude that the invocation of the pow API in line 6 has an explicit data flow from the invocation of the echo API in line 16 since there exists a path from (23, echo) to (14, pow), i.e., $23 \rightarrow 22 \rightarrow 16 \rightarrow 14$, which does not involve any "$ctrl$" edge. Comparatively, there is an implicit data flow from the invocation of the abs API in line 4 to that of the echo API in line 16 since there exists a path from (23, echo) to (7, abs), which has an "$ctrl$" edge (i.e., (20, 9 ctrl)).

6.4 AST-Base Symbolic Interpretation

We next design an interpreter to generate a dependency graph (i.e., G) by traversing AST of an server-side PHP application. We first define a set of operations for a dependency graph. Next, we will design an interpreter to build a dependency graph using defined graph operations.

We define a set of graph operations for a dependency graph G.

$Init(G)$ will initialize a dependency graph G. G will be preloaded with all superglobal variables in its Var. For each superglobal variable, we create a symbolic object and create an association between this superglobal variable and this created object.

$Find(G, id)$ returns an object given its id. If there is no object whose label is id, it will return $null$.

$Create\_Concrete\_Obj(G, x)$ is to create an object of a given concrete value x, where the created object is denoted as (id, x); it returns the id of this object (i.e., id). This function will assure that the assigned label is unique across all objects in G. It will meanwhile add this object into G. Specifically, we will have $C = C \cup \{x\}$, $I = I \cup \{id\}$, and $O_C = O_C \cup \{o\}$.

$Create\_Symbol\_Obj(G, x)$, $Create\_FUNC\_Obj(G,\ x)$, $Create\_OP\_Obj(G,\ x)$, and $Create\_AUX\_Obj(G,\ x)$ are similar to $Create\_Concrete\_Obj(G, x)$. They are used to create objects for a symbol value, a built-in function, an operator, and an AUX operator, respectively. Each of these functions returns the id of the created object, which is unique

e **::=**	(EXPRESSION)
$\mid c$	(Constant)
$\mid x$	(Variable)
$\mid x.f$	(Obj Attribute Access)
$\mid op\ e$	(Unary Operation)
$\mid e_1\ op\ e_2$	(Binary Operation)
$\mid x[e]$	(Array Access)
$\mid \texttt{function}(x_1, \ldots, x_n)\{S\}$	(Func Define)
$\mid f(e_1, \ldots, e_n)$	(Func Call)
$\mid x.f(e_1, \ldots, e_n)$	(Obj Func Call)
$\mid new\ f(e_1, \ldots, e_n)$	(NEW)
S **::=**	(STATEMENTS)
$\mid S_1; S_2$	(Sequence)
$\mid x := e$	(Assignment)
$\mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$	(Conditional)
$\mid \texttt{while}\ e\ \texttt{do}\ S$	(While)
$\mid \texttt{return}\ e$	(Return)

Table 6.1: Core PHP Syntax Interpreted by *UGraph*

across all objects in G.

$\underline{Add\_Edge(G, e)}$ will add an edge into $Edge$ of G. Specifically, $Edge = Edge \cup \{e\}$.

$\underline{Add\_Var(G, (v, id))}$ where v is a variable name and id is the id of an object. This function adds an association between v and id into Env (i.e., $Env = Env \cup (v, id)$). It will meanwhile add v into Var, i.e., $Var = Var \cup v$.

$\underline{Get\_Var(G, v)}$, where v is a variable name, will return id if $(v, id) \in Env$; it will return $null$ otherwise.

$\underline{Get\_OR\_Add\_Var(G, v)}$, where v is a variable name, will return id if $(v, id) \in Env$. If v is not contained in Env, a symbol object (id, s) will be created using $id = Create\_Symbol\_Obj(s)$; an association between x and this symbol object, (x, id) will be added into Env using $Add\_Var(G, (x, id))$; finally, it returns the id id.

$\underline{Enum\_Var(G)}$, returns all variables in the Env of the graph G.

6.4.1 Interpreter

The core of our interpreter is an evaluation function denoted as "$eval(node, G)$", where "$node$" refers to an AST node and G is the dependency graph. An AST node represents either an expression (e.g., a constant, a variable, a binary operation, and a function invoca-

tion) or a statement (e.g., a sequence, an assignment, and a conditional branch). Starting from the root node of an AST, *UGraph* recursively interprets each AST node and build the dependency graph accordingly.

UGraph processes core PHP syntax. We use syntax presented in Table 6.1 to illustrate the design of the $eval(node, G)$ function. For brevity, we describe the evaluation for a few challenging expressions and statements including "Variable", "Binary Operation", "Assignment", "Conditional", and "Function Invocation".

$eval(x,\ G)$: When *UGraph* sees a variable x, it queries the Env_{Var}, attempting to retrieve the id of the object associated with x in Env_{Var}, using $id = Get\_OR\_Add\_Var(G, x)$. This will introduce a symbolic object if x cannot be found in Env_{Var}.

$eval(e_l\ op\ e_r,\ G)$: *UGraph* evaluates e_l and e_r using G individually. Specifically, we denote $id_l = eval(e_l, G)$ and $id_r = eval(e_r, G)$. Next, *UGraph* creates a new operator object using $id_{op} = Create\_OP\_Obj(G, op)$. Two directed edges including $e_l = (id_{op},\ id_l,\ left)$ and $e_r = (id_{op},\ id_r,\ right)$ will be added into G (i.e., $Add\_Edge(G, e_l)$ and $Add\_Edge(G, e_r)$). Edge types of "$left$" and "$right$" indicate e_l and e_r point to this operator's left and right operands, respectively. Finally, *UGraph* returns the id of the operator node, i.e., id_{op}.

$eval(x[e],\ G)$: *UGraph* first evaluates x to retrieve the id id_{arr} of the object associated with x using $id_{arr} = Get\_OR\_Add\_Var(G, x)$. Next, *UGraph* will evaluate e to derive the id id_{idx} of the evaluation result using $id_{idx} = eval(e, G)$. *UGraph* will then create an "$array\_access$" node using $id_{arr\_acc} = Create\_AUX\_Obj(G, array\_access)$. Two directed edges including $e_l = (id_{arr\_acc},\ id_{arr},\ array)$ and $e_r = (id_{arr\_acc},\ id_{idx},\ index)$ will be added into G using $Add\_Edge(G, e_l)$ and $Add\_Edge(G, e_r)$. The type of $array$ for e_l indicates that this edge points to the node for the array; the type of $index$ for e_r means this edge points to the node used as the index.

$eval(x := e,\ G)$: *UGraph* will first evaluate e and get an id denoted as id using $id = eval(e, G)$. Then it will add an association between x and id into the Env_{Var} using

$Add\_Var(G, (x, id))$.

$\underline{eval(f(e_1, \ldots, e_j, \ldots, e_m),\ G}$: When a function is called, ***UGraph*** will first evaluate its arguments (i.e, $e_1, \ldots, e_j, \ldots, e_m$) in G and retrieve their corresponding ids (i.e., $id_1 = eval(e_1, G)$, $\ldots$, $id_j = eval(e_j, G)$, $\ldots$, $id_m = eval(e_m, G)$) for each path. Specifically, for each input e_j, ***UGraph*** will evaluate it to obtain $< l_j^1, \ldots, l_j^i, \ldots, l_j^n >= eval(e_j, G, \mathcal{E})$, where l_j^i is the evaluation result of e_j for Env_i.

$\underline{eval(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ G)}$: ***UGraph*** will first evaluate e using G, obtaining the id of the evaluation result as $c = eval(e, G)$. Then we will make two shallow copies of G, denoted as G_T and G_F where $G_T = G_F = G$. the system then follows three steps.

First, ***UGraph*** will evaluate the "`true`" branch using G_T, denoted as $eval(S_1, G_T)$. Since such evaluation will introduce changes to G_T, we name the updated graph after this evaluation as G'_T. Meanwhile, we evaluate the "`false`" branch using $eval(S_2, G_F)$ to derive a new graph named G'_F.

Second, ***UGraph*** will integrate G'_T and G'_F. It will create a new graph G', which inherit all objects, edges among objects, superglobal variables in their environments, and associations between super-global variables and their associated objects. For each variable v in $V = Enum\_Var(G'_T) \cup Enum\_Var(G'_F)$ (i.e., $v \in V$), we retrieve its IDs in G'_T and G'_F, respectively. Specifically, we denote $id_{v,T} = Get\_Var(G'_T, v)$ and $id_{v,F} = Get\_Var(G'_F, v)$.

- If $id_{v,T} = id_{v,F}$, which means neither S_1 nor S_2 changes v's value, we will add an association between v and $id_{v,T}$ into G' using $Add\_Var(G', (v, id_{v,T}))$.

- If $id_{v,T} \neq id_{v,F}$, which means either or both of S_1 and S_2 changes v's value, we will integrate all possible values for v and their dependency on the condition e. Specifically, we create a new node using $id_{comb} = Create\_AUX\_Obj(G', comb)$. We next create three edges including $e_T = (id_{comb},\ id_{v,T},\ data)$, $e_F = (id_{comb},\ id_{v,F},\ data)$, and $e_{ctrl} = (id_{comb},\ c,\ ctrl)$ and add them into G', where $c = eval(e, G)$. We will then add an association between v and id_{comb} into G' using $Add\_Var(G',\ (v,\ id_{comb}))$.

Third, it will use the new graph G' for the subsequent evaluation (i.e., $G = G'$) after all variables in $V = Enum\_Var(G'_T) \cup Enum\_Var(G'_F)$ is investigated by following the second step.

6.5 Vulnerability Detection

Once a dependency graph is built, we can analyze this graph for vulnerability detection. Each dependency graph, in fact, is a property graph [29], where each node can be represented by an ID and its attribute (i.e., $(id, attr)$) and each edge can be represented by a pair of node IDS and an attribute (i.e., $(id_i, id_j, attr)$). This makes it possible to leverage graph-enabled databases such as for algorithm design and implementation, boosting the efficiency and extensibility.

In our current implementation, we use `Cypher`, the graph query language of `Neo4j` [29], to create deductive database to store dependency graphs and devise detection algorithms as declarative queries.

6.5.1 Representing and Analyzing Dependency Graphs Using Cypher Query Language

`Cypher` is the graph query language of `Neo4j`. The property graph model of `Cypher` is composed of nodes and edges. Table 6.2 presents the elements we have defined to represent the model of dependency graph. The node of dependency graph depicted in `Cypher` a surround with parentheses $()$. The node can be given a variable like (var) and also be reference the node with relative properties as $(var\{attr : P_x\})$ by using curly braces that attribute the node with the property of P_x. `Cypher` represents the edge of the dependency graph using an arrow $-- >$ or $< --$ between two nodes, and any properties relating to the edge can be placed in a pair of square brackets to describe the relationship of these two

$Node$	$(), (var), (var\{attr : P_x\})$
$Edge$	$() - - > (), (var_1) < - - (var_2), (var_1) - [\{attr : P_y\}] - > (var_2)$

Table 6.2: `Cypher` models nodes and edges in a dependency graph

nodes such as $(var_1) - [\{attr : P_y\}] - > (var_2)$ as presented in Table 6.2.

We design detection algorithms using the patterns of the `Cypher`. A pattern is comprised of node and relationship elements and can express complex or straightforward traversals and paths.

$$(M\{attr : P_1\}) - [\{attr : P_e\}] - > (Root) < - [\{attr : P_e\}] - (N\{attr : P_2\})$$

$(M\{attr : P_1\})$, $(N\{attr : P_2\})$, and $(Root)$ are three nodes in the dependency graph; the arrow $(M\{attr : P_1\}) - [\{attr : P_e\}] - > (Root)$ describes an edge that connects node (M) and node $(Root)$ with the property P_e. The above pattern means that node root is the sink of two nodes (M) and node (N) with the edge attribute P_e

6.5.2 Detection Rules

By leveraging the `Cypher` query, we enable to detect the vulnerability of unrestricted file upload in the graph database, `Neo4j`. Figure 6.3 illustrates the detecting rules for the vulnerability of unrestricted file upload.

- **Rule 1**: The control flow of the $move\_uploaded\_file()$ was tainted by $\$\_FILES[`*`][`name`]$.
- **Rule 2**: The date flow of the $move\_uploaded\_file()$ was tainted by $\$\_FILES[`*`][`name`]$.
- **Rule 3**: Checking the extension of the uploaded file by sanitize APIs.

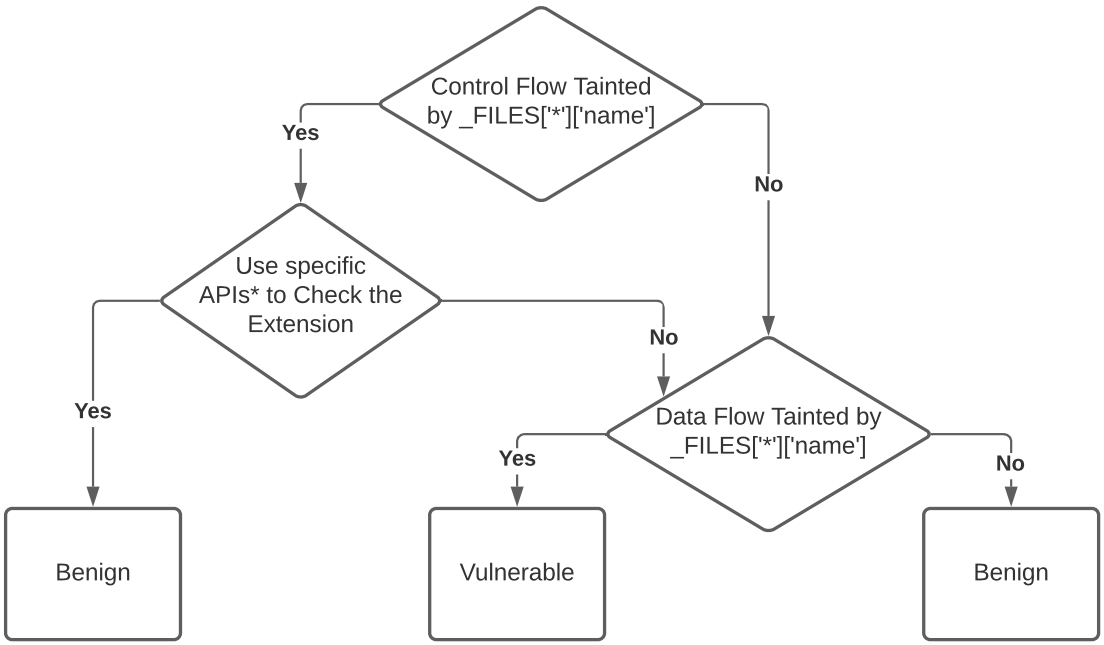

Figure 6.3: The Rules for detecting the unrestricted file upload

When the Rule 1 is satisfied, we need to check Rule 3 next. But if Rule 1 does not meet, we will direct to Rule 2. Once the Rule 2 is satisfied, it means the file upload API (e.g. $move\_uploaded\_file()$) uploads a file without check the extension of the file. It is a vulnerable operation.

Listing 6.2 presents the query for searching the superglobal expression $\$\_FILEs[`*`][`name`]$ in `Neo4j`. Because the expression is a array access expression, we use the query language to depict the structure of the array dimension fetch in PHP language. Listing 6.3 declares the query for searching the $\$\_FILEs[`*`][`name`]$ in the control flow of the function $move\_uploaded\_file()$. First, we get the control flow branch of the function $move\_uploaded\_file()$ in line 1, and find the root of the $\$\_FILES[`*`][`name`]$. After that, we query these two node in the build-in API $gds.alpha.dfs.stream()$ that defined by `Cypher`. It will preforms the deep first search in the graph. After the DFS searching, we build the path and return all nodes by the queries from line 7 to 9.

```
1 MATCH (q{name:'_FILES'})<--(p{name:`array'})<--(o{name:'array_access'})
```

```
2 <--(root{name:'array_access'})-->(target{name:'name'})
3 RETURN root.id
```

Listing 6.2: Query for superglobal variable FIELS

Listing 6.4 introduces the query for searching the $\$\_FILEs[`*`'][`name`']$ in the data flow of the function $move\_uploaded\_file()$. Different from the searching in control flow, the query change the type of the edge or relationship of the function node from 'ctrl' to 'arg' that means searching only perform on the argument branch of the function node.

```
1 MATCH ({name:'move_uploaded_file'})-[{type:'ctrl'}]->(root)
2 MATCH ({name:'_FILES'})<--({name:'array'})<--({name:'array_access'})
3       <-[*..2]-(t{name:'array_access'})-->({name:'name'})
4 WITH id(root) AS start, [id(t)] AS targets, t AS taint_node
5 CALL gds.alpha.dfs.stream('myGraph',
6                           {startNode:start, targetNodes:targets})
7 YIELD path
8 UNWIND [n IN nodes(path) WHERE n.id = taint_node.id | n ] as res
9 RETURN res
```

Listing 6.3: Searching the superglobal variable _FIELS in contral flow of move_uploaed_file()

```
1 MATCH ({name:'move_uploaded_file'})-[{type:'arg'}]->(root)
2 MATCH ({name:'_FILES'})<--({name:'array'})<--({name:'array_access'})
3       <-[*..2]-(t{name:'array_access'})-->({name:'name'})
4 WITH id(root) AS start, [id(t)] AS targets, t AS taint_node
5 CALL gds.alpha.dfs.stream('myGraph',
6                           {startNode:start, targetNodes:targets})
7 YIELD path
8 UNWIND [n IN nodes(path) WHERE n.id = taint_node.id | n ] as res
9 RETURN res
```

Listing 6.4: Searching globalvariable _FIELS in data flow of move_uploaed_file()

Listing 6.5 presents the query for searching the sanitizaiton APIs in the control flow of the function $move\_uploaded\_file()$.

```
MATCH ({name:'move_uploaded_file'})-[{type:'ctrl'}]->(root)
MATCH ({name:'_FILES'})<--({name:'array'})<--({name:'array_access'})
      <-[*..2]-(t{name:'array_access'})-->({name:'name'})
WITH id(root) AS start, [id(t)] AS targets, t AS taint_node
CALL gds.alpha.dfs.stream('myGraph',
                          {startNode:start, targetNodes:targets})
YIELD path
UNWIND [n IN nodes(path) WHERE n.name IN ['pathinfo', 'explode'] AND
        (n)-[*]->(target) | n ] as sanitizationAPIs
RETURN sanitizationAPIs
```

Listing 6.5: Searching sanitization APIs in control flow of move_uploaed_file()

6.6 Evaluation

We have employed *UGraph* to detect vulnerable PHP applications by scanning `WordPress` plugins. `WordPress` features a large repository of PHP-based, open-source plugins that are contributed from a variety of sources. We have crawled and tested 15,154 `WordPress` plugins in a reverse chronological order (starting from 4/22/2018) based on their last updated time.

The detection results are summarized in Table 6.3. We manually verified each application that is identified as vulnerable by *UGraph* and searched it in the vulnerablity database of `WordPress` plugins [45]. The column of detection Table 6.3 indicates whether a detection result is a vulnerability that has been reported (i.e., "known"), or a vulnerability that has not been previously reported to the best of our knowledge (i.e., "new"). As shown in the detection results, The system has successfully detected a large number of vulnerable real-world web applications, including 14 new ones.

Compared with *UFuzzer*, *UGraph* addresses the intrinsic restriction faced by *UFuzzer*. *UFuzzer* was not successful in mutating the free variable input of the vulnerable application Gallerio 1.0, which needed the string 'yes' to satisfy their reachability conditions. In contrast, our novel system performs the tainted analysis for the control flow restriction that

bypasses the free variable matching successfully.

We have experimented with two publicly available PHP vulnerability scanners including RIPS [16, 14] and WAP [28], where both of them offer options to detect unrestricted file uploading vulnerabilities. We used both of them to scan 86 WordPress plugins. We have manually verified the detection results. Specifically, RIPS reported totally 43 vulnerable samples while 12 of them are false positives; WAP reported totally 13 vulnerable samples while 2 of them are false positives.

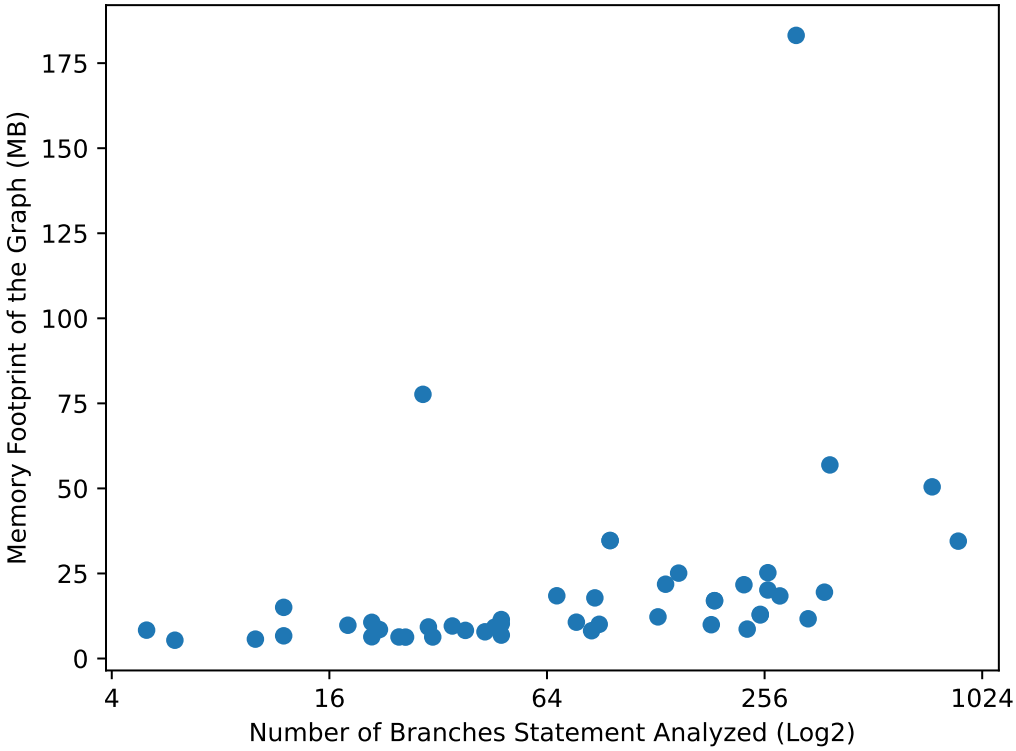

Figure 6.4: Distribution of the Memory Footprint

System	Loading(s)	Query(s)	New System	UFuzzer	UChecker	RIPS	WAP	Detection
ads-ez	25.4	0.11	✔	✖	✖	✖	✖	New
classyfrieds	39.8	0.57	✔(CVE-2021-24252)	✖	✖	✔	✔	New
college-publisher-import	3.88	0.33	✔(CVE-2021-24253)	✔	✖	✔	✖	New
email-artillery	15.3	0.1	✔(CVE-2021-24490)	✔	✔	✔	✖	New
fileviewer	1.5	0.08	✔(CVE-2021-24491).	✔	✖	✔	✖	New
formi-form-builder	3.66	0.3	✔	✔	✔	✔	✔	New
google-analytics-client	1.72	0.09	✔	✔	✖	✔	✔	New
image-twinning	1.7	0.1	✔	✔	✖	✔	✖	New
advert-manager-plugin	0.31	0.06	✔	✔	✔	✔	✔	New
rad-dropbox-uploader	1.56	0.34	✔	✔	✖	✔	✖	New
rad-text-highlighter	1.5	0.09	✔	✔	✖	✔	✖	New
scroll-baner	0.87	0.09	✔(CVE-2021-24642)	✖	✖	✔	✖	New
simple-schools-staff-directory	42.75	0.13	✔(CVE-2021-24663)	✔	✖	✔	✖	New
daily-different-corner-band	0.67	0.3	✔	✔	✖	✔	✖	New
Testimonials King Light 0.1	6.55	0.39	✔	✔	✖	✖	✖	Known
WP-Curriculo Vitae Free 6.1	19.63	0.3	✔	✔(CVE-2021-24222)	✖	✖	✖	Known
Easy Form Builder 1.0	1.68	0.25	✔	✔(CVE-2021-24224)	✔	✖	✖	Known
imagements 1.2.5	0.59	0.08	✔	✔(CVE-2021-24236)	✔	✖	✖	Known
Event Banner 1.3	2.3	0.16	✔	✔(CVE-2021-24251)	✖	✔	✖	Known
Quick Image Transform 1.0.1	0.67	0.09	✔	✔	✖	✖	✖	Known
BSK Files Manager 1.0.0	0.8	0.33	✔	✔	✖	✖	✖	Known
Gallerio 1.0.1	14.67	2.28	✔	✖	✔	✖	✖	Known
Banner Cycler 1.4	1.91	0.09	✔	✔	✖	✔	✖	Known
N5 Upload Form 1.0			✖	✔(CVS-2021-24223)	✖	✖	✖	Known

Table 6.3: Detection Results. Our system detected 26 out of 27 known vulnerable scripts. It detected 14 unreported vulnerable plugins.

6.6.1 Performance

Figure 6.4 presents the distribution of the peak memory consumption for each application. These measures demonstrated *UGraph*'s great performance. Specifically, although the generated dependency graphs are considerably large, the vast majority of analyses (i.e., more than 89%) are done within 30 second and all applications resulted in less than 100 MB of maximal memory consumption as presented in Table 6.3.

6.6.2 New Vulnerable Examples

Classyfrieds is a `WordPress` plugin that provides a classifieds system for its user and allows users having admin privilege account to upload image file such as .gif, .jpeg, .jpg, or .png. As presented in listing 6.6 The developer seems aware of the file type sanitization and add a filter to abort any uploading action if it submits a non-image file. However, the added filter to only investigates the type of the uploaded file, $\$\_FILES['foto']['type']$, which is the type of the file derived from the client's request. Because an attacker have complete control the browser, she can upload a executable PHP script and instruct the browser to change the $\$\_FILES['foto']['type']$ to 'image/gif', successfully bypassing the filter. *UGraph* successfully queries the control flow of the $move_uploaded_file()$ without being tainted by $\$\_FILES['*']['name']$, and the $\$\_FILES['*']['name']$ tainted the data flow.

```
1 <?php
2 if ($_FILES['foto']['error'] == "0") {
3    if (($_FILES['foto']['size'] < 2000000) &&
4        ($_FILES['foto']['type'] == 'image/gif' ||
5            $_FILES['foto']['type'] == 'image/jpeg' ||
6            $_FILES['foto']['type'] == 'image/jpg' ||
7            $_FILES['foto']['type'] == 'image/png')
8    ) {
9        move_uploaded_file($_FILES["foto"]["tmp_name"],
```

```
10              "/classyfrieds/" . /*...*/ .
11              str_replace(" ", "-", $_FILES["foto"]["name"]));
12
13      } else {
14          $error .= $cfl[nopic];
15      }
16  }
```

Listing 6.6: Vulnerable Code in `Classyfrieds`

Ads EZ Lite is a plugin in `WordPress`. It provides a personal and powerful Ad server for the user and supports the simplest possible way to serve the ads to multiple web pages. The plugin intends to support an interface to receive the uploading images and tries to validate the uploading files but a misused API ”getimagesize()” rooted the vulnerability of arbitrary file upload as presented in Listing 6.7. This API have cautioned by the PHP official manual [2] that it only can be used for obtaining the information of an image file rather than checking whether a given file is a valid image or not. a ”.php” file can be uploaded without any restriction. *UGraph* detected this vulnerable program successfully.

```
1 <?php
2 $ds = DIRECTORY_SEPARATOR;
3 $targetPath = dirname(dirname(__DIR__)) . $ds . "banners" . $ds;
4 if (!empty($_FILES)) {
5     $tempFile = $_FILES['file']['tmp_name'];
6     if (getimagesize($tempFile) === false) {
7         http_response_code(400);
8         $error = "{$_FILES['file']['name']}: Not allowed.";
9         die($error);
10    }
11    $targetFile = $targetPath . $_FILES['file']['name'];
12    if (!@move_uploaded_file($tempFile, $targetFile)) {
13        http_response_code(400);
14        die(/*...*/);
15    }
16 }
```

Listing 6.7: Vulnerable Code in `Classyfrieds`

6.7 Related Work

Static program analysis has been widely used to detect variety of vulnerabilities in server-side web programs [54, 21, 60, 46, 15, 9, 44, 50, 14]. These methods uses various strategies such as signature matching, symbolic execution, and taint analysis.

For example, Staicu et. al [50] leverages pre-defined templates to detect Node.js applications vulnerable to injection attacks that exploit `exec` or `eval` APIs. However, such method requires templates to be predefined at the AST level, thereby limiting its adaption to variant and new vulnerabilities.

A few methods [44] including ours [22] take advantage of symbolic execution, where they first model conditions to exploit vulnerabilities as symbolic constraints and evaluate these constraints using automated solvers [17]. These methods tend to be sound. However, they cannot always use SMT solvers to model web programming languages since web programming languages are usually dynamic typing and SMT solver languages are static. Therefore, certain programs cannot be completely modeled, introducing false negatives. In addition, symbolic execution is computationally expensive, inherently suffering from path explosion.

Other systems mainly leverage taint analysis, same to our system, detecting vulnerabilities through tracing how untrusted data influences sensitive APIs. For example, Dahse et. al [15] designed a system to detect SQL injections and XSS using data flow analysis.

Barth et. al [9] designed a system to detect XSS attacks by analyzing the structure of the content submitted to the server. Unfortunately, none of these methods intends to deliver an intuitive, reusable language model that can be efficiently used to build new applications.

Dahse et. al [14] proposed block and function summaries to detect taint-style vulnerabilities. This work is closest to ours. However, their work drastically differs from *UGraph* in multiple perspectives. First, their method processes SSA-based IR, thereby causing information loss fo the source code. In contrast, *UGraph* performs analysis di-

rectly on program AST, enabling one-to-one mapping between the source code and the graph model. Second, their method leverages compositional strategy to perform whole program analysis, where function summaries are individually generated and later integrated for taint analysis. Comparatively, *UGraph* directly performs context-sensitive, whole program analysis, which is more precise. Finally, our model is graph-based, enabling more intuitive and efficient analyses by translating them into graph operations.

6.8 Discussion

Although *UGraph* is currently implemented to analyze server-side web programs written in PHP, it can be easily extended to support other programming languages. One will only need to extend the interpreter to process AST with new syntax. The `Neo4j` query can be readily reused by changing names of APIs and super global variables.

The current implementation of *UGraph* has a few limitations. First, *UGraph* leverages a misuse-based detection paradigm and it needs rules to perform detection. Nevertheless, it generates intuitive graph-based program models that are further represented by graph database models. As a consequence, detection rules could be implemented as Cypher queries with extremely low cost. In addition, our proposed graph models show great promise to support statistical analysis, which falls into our future works. Second, *UGraph* 's interpreter does not cover all language features of PHP. For example, it does not precisely model loops. As a consequence, scripts analyzed by our system might be incomplete, leading to detection inaccuracy. A potential solution is to integrate dynamic analysis to access all executables produced at runtime.

Finally, loading the dependency graph to `Neo4j` account for the vast majority of the analysis time. Its performance can be further improved using potential solutions such as `Neo4j` causal cluster with Apache Spark.

6.9 Summary

We have built a novel system to automatically detect vulnerable PHP-based web programs. it generates graph models by interpreting abstract syntax trees of PHP source code. These graph models are further represented in the graph database, `Neo4j`. Vulnerability detection algorithms have been developed as declarative rules that lead to provable accuracy. Experiments have demonstrated that our system detected 14 vulnerable web applications that have not been publicly reported and received 6 CVEs.

Chapter 7: Conclusion

In this dissertation, we present two graph-based models for PHP programming language and implement three systems, i)UChecker, ii)UFuzzer, and iii) a novel system discussed in chapter 6, to detect unrestricted file upload vulnerability in PHP-based web applications by using three complementary methods. Our first graph-based model is *heap graph* that is a intermediate representation (IR) of PHP programs that models symbolic execution results of a program along all paths towards a given statement. On the basis of the *heap graph*, we implement our first detection system, namely *UChecker*, to automatically detect PHP-based web programs with unrestricted file upload vulnerabilities. By leveraging the SMT solver, UChecker verifies the model of vulnerability constraints and detects three vulnerable web applications that didn't have been reported. But the gap between the PHP-based statement and SMT solver is the barrier for the batch analysis. To bridge this gap, we integrate the static analysis with fuzzing as a static-fuzzing co-analysis in our second system, namely *UFuzzer*, to detect the unrestricted file upload vulnerabilities. *UFuzzer* detects 31 new vulnerable services that have not been publicly reported and contributes 5 CVEs. Although we have demonstrated the effectiveness of the *heap graph*, the path explosion is the most significant challenge that limited the scalability of our detection system. Our second graph-based model mitigates this challenge by using a novel model. We name it *dependency graph*. Taking advantage of this model, we established the third vulnerability detection system. In this system, we use an industry-level graph database, `Neo4j`, to represent the objects of the dependency graph and query the PHP program's pattern by using

Cypher query language. The system detected 14 vulnerable unreported web applications and contributed 6 CVEs. In summary, each of these systems has its pros and cons and leverages altered methods to complement the detection of the unrestricted file uploaded vulnerabilities on PHP-based server-side web applications.

Bibliography

[1] Wp demo buddy 1.0.2. https://wordpress.org/plugins/wp-demo-buddy. [Online; accessed 05-Dec-2018].

[2] getimagesize. https://www.php.net/manual/en/function.getimagesize.php, accessed on July 17, 2021.

[3] DSN 2019. The 49th ieee/ifip international conference on dependable systems and networks, 2019.

[4] RAID 2021. The 24th international symposium on research in attacks, intrusions and defenses, 2021.

[5] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018.

[6] Oxana Andreeva, Sergey Gordeychik, Gleb Gritsai, Olga Kochetova, Evgeniya Potseluevskaya, Sergey I Sidorov, and Alexander A Timorin. Industrial control systems vulnerabilities statistics. *Kaspersky Lab, Report*, 2016.

[7] I. Andrianto, M. M. I. Liem, and Y. D. W. Asnar. Web application fuzz testing. In *2017 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2017.

[8] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272. ACM, 2008.

[9] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *2009 30th IEEE Symposium on Security and Privacy*, pages 360–371, May 2009.

[10] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[11] Davide Canali and Davide Balzarotti. Behind the scenes of online attacks: an analysis of exploitation behaviors on the web. In *20th Annual Network & Distributed System Security Symposium (NDSS 2013)*, pages n–a, 2013.

[12] CISOMAG. 76% security professionals face cybersecurity skills shortage: Reporte.

[13] Wikipedia contributors. Unrestricted file upload, 2018. [Online; accessed 22-July-2018].

[14] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.

[15] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security Symposium*, pages 989–1003, 2014.

[16] Johannes Dahse and Jörg Schwenk. Rips-a static source code analyser for vulnerabilities in php scripts. In *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*, 2010.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] dimdavid. File provider 1.2.3, 2017. [Online; accessed 30-July-2018].

[19] F. Duchene, R. Groz, S. Rawat, and J. Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 815–817, 2012.

[20] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[21] Wassermann Gary and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 171–180. IEEE, 2008.

[22] J. Huang, Y. Li, J. Zhang, and R. Dai. Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 581–592, 2019.

[23] J. Huang, J. Zhang, J. Liu, C. Li, and R. Dai. Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis. In *2021 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[24] 2014) Kali.org.(February 18. Wfuzz package description. http://tools.kali.org/web-applications/wfuzz.

[25] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. Fuse: Finding file upload bugs via penetration testing. In *2020 Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium, 2020.

[26] L. Li, Q. Dong, D. Liu, and L. Zhu. The application of fuzzing in web software security vulnerabilities test. In *2013 International Conference on Information Technology and Applications*, pages 130–133, 2013.

[27] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll (t) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification*, pages 646–662. Springer, 2014.

[28] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2016.

[29] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, 2013.

[30] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):158–191, 1998.

[31] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[32] Patrick. Unrestricted file upload (rce), 2018. [Online; accessed 22-July-2018].

[33] Andrey Petukhov and Dmitry Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[34] Nikita Popov. Php fuzzer. *URl: https://github.com/nikic/PHP-Fuzzer (visited on 2020-12-09)*.

[35] Nikita Popov. Php parser. *URl: https://github. com/nikic/PHP-Parser (visited on 2014-03-28)*, 2014.

[36] PortSwigger.(n.d). Burp suite. http://portswigger.net/burp/, accessed on April 15, 2020.

[37] Ajay Randhawa. User-management-php-mysql, 2018.

[38] CVE Report. Unrestricted file upload vulnerability in the avatar uploader module before 6.x-1.3, 2018. [Online; accessed 30-July-2018].

[39] CVE Report. Unrestricted file upload vulnerability in the joomla content editor, 2018. [Online; accessed 30-July-2018].

[40] Imam Riadi and Eddy Irawan Aristianto. An analysis of vulnerability web against attack unrestricted image file upload. *Computer Engineering and Applications Journal*, 5(1):19–28, 2016.

[41] Darius S. How to protect site from malware upload by file upload form. https://blog.threatpress.com/protect-site-malware-upload/, 2018.

[42] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[43] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 277–287. IEEE, 2012.

[44] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[45] WordPress Security Scanner. Wordpress vulnerability database.

[46] Sooel Son and Vitaly Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, pages 8:1–8:13, New York, NY, USA, 2011. ACM.

[47] Sourceforge.(n.d). Jbrofuzz. https://sourceforge.net/projects/jbrofuzz/, accessed on April 15, 2020.

[48] Sourceforge.(n.d). Wapiti. https://wapiti.sourceforge.io/, accessed on April 15, 2020.

[49] Beth Stackpole. Covid-19 attacks continue and new threats on the rise.

[50] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node. js. In *Network and Distributed System Security (NDSS)*, 2018.

[51] Nasir Uddin and Mohammad Jabr. File upload security and validation in context of software as a service cloud model. In *IT Convergence and Security (ICITCS), 2016 6th International Conference on*, pages 1–5. IEEE, 2016.

[52] w3af. (n.d.). w3af - open source web application security scanner. http://w3af.org/, accessed on April 15, 2020.

[53] Liu Qiang Wang Chunlei, Liu Li. Automatic fuzz testing of web service vulnerability. *IET Conference Proceedings*, pages 1.035–1.035(1), January 2014.

[54] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

[55] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260. ACM, 2008.

[56] Brad Williams, David Damstra, and Hal Stern. *Professional WordPress: design and development*. John Wiley & Sons, 2015.

[57] Brad Williams, Ozh Richard, and Justin Tadlock. *Professional WordPress Plugin Development*. Wrox Press Ltd., 2011.

[58] williewonka. Imagements, 2012.

[59] WordPress.org. sanitize_file_name. URl:https://developer.wordpress.org/reference/functions/, accessed on April 15, 2020.

[60] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pages 179–192, 2006.

[61] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 652–661. IEEE, 2013.

[62] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124. ACM, 2013.