2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

UChecker: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities

Jin Huang*, Yu Li*, Junjie Zhang*, and Rui Dai†
*Wright State University, †University of Cincinnati
{huang.70, li.137, junjie.zhang}@wright.edu, rui.dai@uc.edu

***Abstract*—Unrestricted file upload vulnerabilities enable attackers to upload and execute malicious scripts in web servers. We have built a system, namely *UChecker*, to effectively and automatically detect such vulnerabilities in PHP server-side web applications. Towards this end, *UChecker* first interprets abstract syntax trees (AST) of program source code to perform symbolic execution. It then models vulnerabilities using SMT constraints and further leverages an SMT solver to verify the satisfiability of these constraints. *UChecker* features a novel vulnerability-oriented locality analysis algorithm to reduce the workload of symbolic execution, an AST-driven symbolic execution engine with compact data structures, and rules to translate PHP-based constraints into SMT-based constraints by mitigating their semantic gaps. Experiments based on real-world examples have demonstrated that *UChecker* has accomplished a high detection accuracy. In addition, it detected three vulnerable PHP scripts that are previously unknown.**

Keywords*—*web security, vulnerability, detection, symbolic execution, program analysis

I. Introduction

Web applications with unrestricted file upload vulnerabilities will allow attackers to upload a file with malicious code, which can be executed on the server. The uploaded file, once executed, could be used to launch a variety of attacks such as installing web shells [1], contaminating web applications, spreading malware, and phishing. Such vulnerabilities are particularly significant for server side scripts (e.g., those with extensions such as "`.php`", "`.asp`", and "`.js`") that are treated as automatically executable, requiring no file system permissions for execution. File upload vulnerabilities are taken as top web vulnerabilities by OWASP [2]; they have also been recognized as one of most common vulnerability types [3] for `WordPress`, a leading PHP-based open-source content management system (CMS) [4].

It is therefore of urgent importance to detect unrestricted file upload vulnerabilities. Unfortunately, despite active case studies [5], [6], 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.

In order to accomplish these objectives, *UChecker* performs automated symbolic execution to model conditions to exploit a vulnerability. These conditions are expressed using SMT constraints and verified using an SMT solver with provable accuracy [7], [8]. Different from existing symbolic execution methods [9], [10] that use intermediate representation (IR) such as static single assignment (i.e., SSA), *UChecker* performs symbolic execution directly using abstract syntax trees (AST) derived from web applications. Compared to after-compilation IR, AST offers unique advantages since it enables the one-to-one mapping between AST nodes and lines of source code. Such mapping makes possible the identification of source code that are directly relevant to the vulnerability. This not only offers developers with immediate source-code-level feedbacks, but also facilitates various vulnerability-mitigation applications such as annotation and visualization.

Building *UChecker*, however, is faced with significant challenges. First, symbolic execution is computationally expensive and known to suffer from the path explosion challenge [11]. 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.

978-1-7281-0057-9/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN.2019.00064

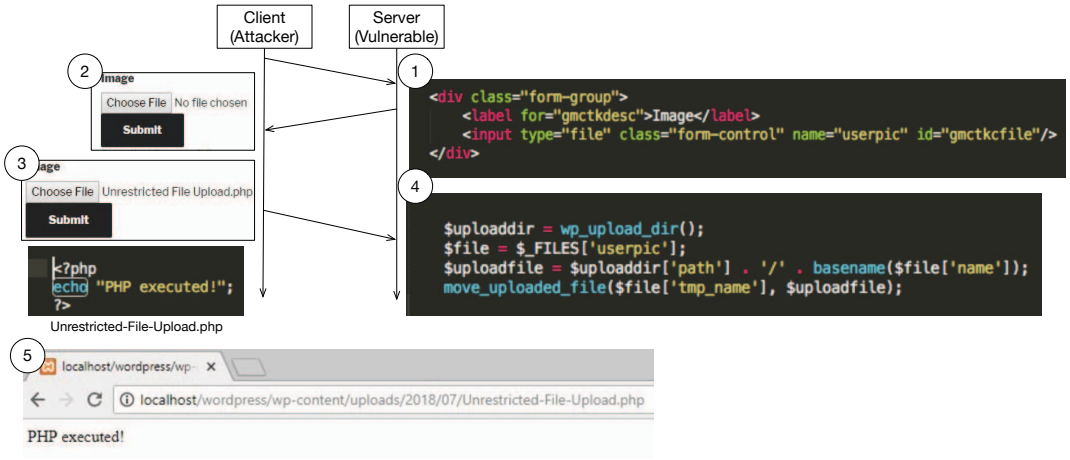

Fig. 1: A typical scenario of unrestricted file upload vulnerability

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

II. Background

Figure 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 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 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.

III. System Design

Figure 2 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. III-A)
- **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 Sec. III-B)
- **Vulnerability Modeling**: Using heap graph, *UChecker* models vulnerabilities using two constraints

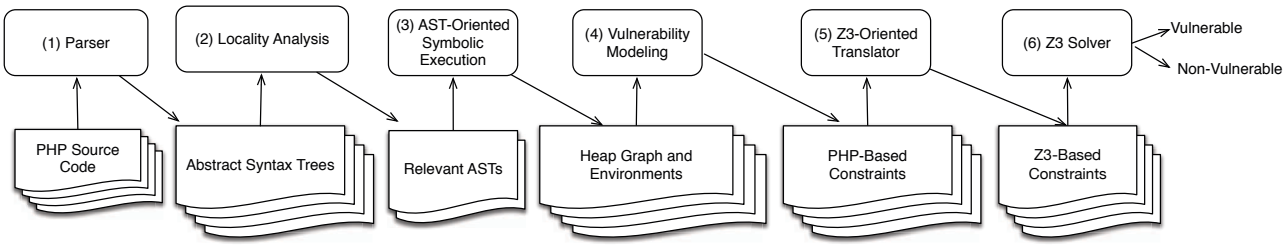

Fig. 2: *UChecker* architecture

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. III-C)

- **Z3-Oriented Translation**: *UChecker* translates PHP-based constraints into Z3-based constraints [8] guided by a set of novel translation rules. These rules aim to mitigate the semantic gap between PHP and Z3. (see Sec. III-D)
- **SMT-Based Verification**: *UChecker* evaluates the satisfiability of Z3-based constraints using the Z3 SMT solver.

A. 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 Section II, 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()` (or `file_put_content()`) 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 [12]. 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 function getFileName($file){
2     return $_FILES[$file]['name'];
3 }
4
5 function handle_uploader($file, $savePath){
6     $path_array = wp_upload_dir();
7     $pathAndName = $path_array['path'] . "/" . $savePath;
8     if (!move_uploaded_file($_FILES[$file]['tmp_name'],
          $pathAndName)) {
9         return false;
10    }
11    return true;
12 }
13
14 if (!handle_uploader("upload_file", getFileName("
       upload_file"))) {
15        echo "File Uploaded failure!";
16 }
```

Listing 1: An example PHP file named "`example1.php`"

An example PHP file namely "`example1.php`" is presented in Listing 1. "`example1.php`" invokes `handle_uploader()`, which next invokes `getFileName($file)` in its parameter. Therefore, "`example1.php`" calls two functions including `getFileName($file)` and `handle_uploader($file, $savePath)`, where `getFileName()` accesses $_FILES and `handle_uploader()` calls `move_uploaded_file()`. *UChecker* will construct an extended call graph for `example1.php` as illustrated in Figure 3. The node "`example1.php`" is the lowest common ancestor for the "$_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 1). Other scripts, if they do not contain such lowest common ancestors, will not be analyzed.

B. AST-Based Symbolic Execution

For the PHP file or the function identified by the locality analysis, *UChecker* performs 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

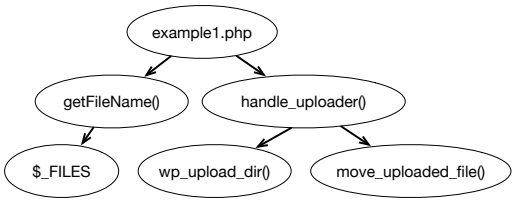

Fig. 3: The extended call graph generated from Listing 1

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. By traversing the heap graph, *UChecker* can symbolically model the reachability constraint for each path and the name of the uploaded file.

1) Heap Graph and Environments

We first define 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 "+", "-", and ".").
- 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 next 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.

- 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.

It is worth noting that a program may have multiple paths and each path has its own environment. We therefore define $\mathscr{E} = \{Env_1, \ldots, Env_i, \ldots, Env_n\}$ for n execution paths of a program.

In order to illustrate the heap graph and environments, we use an example presented in Listing 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.

```
1    <?php
2        $a = 55;
3        $b =$_GET['number'];
4        if($a + $b > 10){
5                $a = $b - 22;
6        }
7        else
8                $a = 88;
9    ?>
```

Listing 2: Sample code with two paths

Figure 4 presents the heap graph and path environments that *UChecker* generates for the example in Listing 2. For this specific example, the heap graph G is:

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

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 $\mathscr{E} = \{Env_1, Env_2\}$. 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 4, 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,

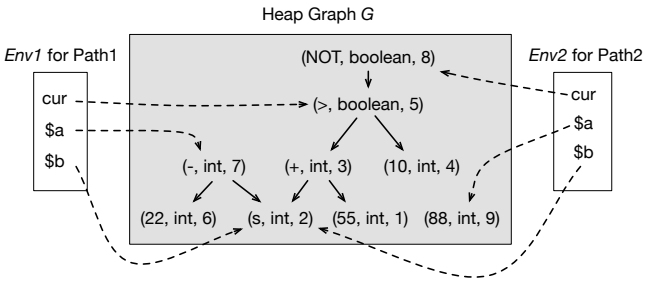

Fig. 4: The heap graph and environments for the sample code in Listing 2

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

TABLE I: Core PHP Syntax Interpreted by *UChecker*

which is specifically (> (+ s 55) 10). This facilitates the usage of SMT solvers, such as Z3 [8] and Yices [13], 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.

2) Operations for Heap Graph and Environments

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

$Find(G,l)$ 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.

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

$Get\_Map(Env,v)$ 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.

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

$Add\_Map(Env,(v,l))$ 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., $Map = Map \cup (v,l)$).

$ER(G,Env,l)$ 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 $u = Create\_OP\_Obj$(AND,Boolean) (i.e., the created object is (AND,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) AST-based Interpretation

We next design an interpreter to generate the heap graph (i.e., G) and a set of environments ((i.e., $\mathscr{E} = \{Env_1,\ldots,Env_i,\ldots,Env_n\}$)) by traversing ASTs using operations defined for heap graph and environments. Our interpreter processes the root node (i.e., a file or a function) identified by the locality analysis. It explores all paths towards the execution of a file upload built-in function (i.e., `move_uploaded_file()` or `file_put_content()`).

UChecker recursively evaluates each node in the AST, where the evaluation function is denoted as $eval(node,G,\mathscr{E})$. $node$ 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; $\mathscr{E}$ is a set of environments.

UChecker starts with the initialization of G and $\mathscr{E}$. 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 Ø. $\mathscr{E}$ is initialized with one path $\mathscr{E} = \{Env\}$. For Env, both Var and Map are initialized as Ø; $cur = null$ (i.e., the reachability constraint is empty).

UChecker processes core PHP syntax. We use syntax presented in Table I to illustrate the design of the $eval(node,G,\mathscr{E})$ function. Without the loss of generality, we consider $\mathscr{E}$ has n paths upon evaluating an AST node,

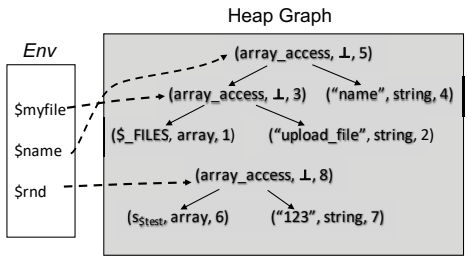

Fig. 5: The heap graph for array access statements in Listing 3

which is denoted as $\mathscr{E} = \{Env_1, \ldots, Env_i, \ldots, Env_n\}$. 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 $\mathscr{E}$ 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".

$eval(x,\ G,\ \mathscr{E})$: When *UChecker* sees a variable x, it queries each Env_i in $\mathscr{E}$ 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, *UChecker* returns a vector of labels denoted as $< l_1, \ldots, l_i, \ldots, l_n >$, where l_i is for Env_i.

$eval(e_1\ op\ e_2,\ G,\ \mathscr{E})$: *UChecker* evaluates e_1 and e_2 using G and $\mathscr{E}$. We denote $< l_1, \ldots, l_i, \ldots, l_n >= eval(e_1, G, \mathscr{E})$ and $< r_1, \ldots, r_i, \ldots, r_n >= eval(e_2, G, \mathscr{E})$. Then for each path (i.e., Env_i), *UChecker* 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 $Add\_Edge(e_{i,r})$). *UChecker* preserves the order of these two edges for k_i so that it can differentiate between the "left" and "right" operand. Finally, *UChecker* 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.

$eval(x[e],\ G,\ \mathscr{E})$: *UChecker* 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, *UChecker* will evaluate e for n paths in $\mathscr{E}$, getting a vector of labels denoted as $< r_1, \ldots, r_i, \ldots, r_n >= eval(e, G, \mathscr{E})$, where r_i is the label of the returned object for the ith path (i.e., Env_i).

UChecker 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, *UChecker* 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, *UChecker* 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 $myfile = $_FILES['upload_file'];
2 $name = $myfile['name'];
3 $rnd = $test['123'];
```

Listing 3: Array Access Statements

Figure 5 visualizes the heap graph and the environment for a PHP statement in Listing 3. `$myfile` refers to `$_FILES['upload_file']`, where `$_FILES` is recognized as a superglobal variable. Therefore, *UChecker* creates an object with a special name of `$_FILES`, resulting the object $(\$\_FILES, array, 1)$ as in Figure 5. The object $("upload\_file", string, 2)$ 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 $("upload\_file", string, 2)$, 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 $("name", string, 4)$. 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, $("123", string, 7)$, is also created. $(arrray\_access, \perp, 8)$ is finally created to combine $(s_{\$test}, array, 6)$ and $("123", string, 7)$ as array name and index, respectively.

$eval(x := e,\ G,\ \mathscr{E})$: *UChecker* will first evaluate e and get a vector of returned labels, i.e., $< l_1, \ldots, l_i, \ldots, l_n >= eval(e, G, \mathscr{E})$. Then for each Env_i, *UChecker* will add an association between x and l_i into the Map of Env_i (i.e., $Add\_Map(Env_i, (x, l_i))$).

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

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

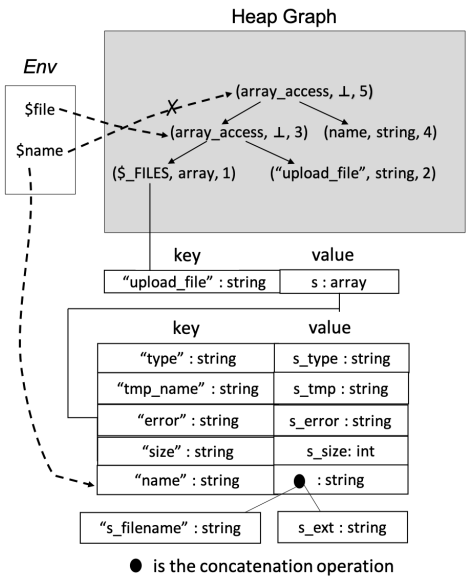

Fig. 6: An example of pre-structured array built for $_FILES in Listing 3. s, s_{type}, s_{tmp}, s_{error}, s_{size}, s_{path} s_{name}, and s_{ext} are symbolic values.

- We next evaluate the "`false`" branch. For each path (e.g., Env_i) in $\mathscr{E}_F$, 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 $\mathscr{E}_F$, we first create a "`NOT`" operator node using $r_i = Create\_OP\_Obj(\texttt{NOT}, boolean)$ and add r_i into G using $Add\_OP\_Obj(G, r_i)$, 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, *UChecker* calls $ER(G, Env_i, r_i)$. Following this, *UChecker* will extend all environments in $\mathscr{E}_F$ with their corresponding '`false`" branching conditions, resulting a set of new environments named $\mathscr{E}'_F = \{ER(Env_1, r_1), \ldots, ER(Env_n, r_n)\}$. Then, *UChecker* will recursively call $eval(S_2, G, \mathscr{E}'_F)$ and produce a set of new environments denoted as $\mathscr{E}''_F = \{Env_{F,1}, \ldots, Env_{F,v}\}$, where $v \geq n$.
- After both branches are evaluated, *UChecker* will join their resulted environment using $\mathscr{E} = \mathscr{E}''_T \cup \mathscr{E}''_F$

It is worth noting that *UChecker* 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.

4) Assigning Symbolic Values

UChecker 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 *UChecker* conducts symbolic execution on a fraction of PHP programs identified by locality analysis. *UChecker* also performs light-weight type inference to assign a type to a symbolic value based on its associated operator or built-in function.

UChecker handles $_FILES as a special case since its structure is known a priori. Specifically, as discussed in Section II, $_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 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 6 presents how $_FILES, i.e., $(\$\_FILES, array, 1)$ in Figure 5, 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 $(".", s_{filename}, s_{ext})$.

C. 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 $\mathscr{E}$ to obtain objects for all paths, denoted as $< l_{s,1}, \ldots, l_{s,i}, \ldots, l_{s,n} >= eval(e_{src}, G, \mathscr{E})$, 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 $\mathscr{E}$ to obtain labels for resulted objects, denoted as $< l_{d,1}, \ldots, l_{d,i}, \ldots, l_{d,n} >= eval(e_{dst}, G, \mathscr{E})$, 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

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	$trl((".", e_1:t_1, e_2:t_2))$	(str.++ $trl(e_1:t_1)$ $trl(e_2:t_2)$) where $t_1 = t_2 = string$
String replace	$trl(\text{“}str\_replace", e_1:t_1, e_2:t_2, e_3:t_3))$	(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	$trl((''intval", e:t))$	(str.to.int $trl(e:t)$) where $t = string$
Index of string	$trl((\text{“}strpos", e_1:t_1, e_2:t_2))$	(str.indexof $trl(e_1:t_1)$ $trl(e_2:t_2)$) where $t_1 = t_2 = string$
String length	$trl(\text{“}strlen", e:t)$	(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	$trl(\text{“}And", e_1:t_1, e_2:t_2)$	(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	$trl(\text{“}==", e_1:t_1, e_2:t_2)$	(= $trl(e_1:t_1)$ $trl(e_2:t_2)$) 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	$trl((\text{“}in\_array", needle, haystack:array))$	(or $trl(\text{“}=", needle, e_1:t_1), \ldots, trl(\text{“}=", needle, e_n:t_n)$ 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	$trl((\text{“}substr", str:t_1, start:t_2))$	(str.substr $trl(str)$, $trl(start)$, (str.len $trl(str)$)) where $t_1 = t_2 = string$
Substring	$trl((\text{“}substr", str:t_1, start:t_2, len:t_3))$	(str.substr $trl(str)$, $trl(start)$, $trl(len)$) where $t_1 = t_2 = string$ and $t_3 = int$
Tail Element	$trl((\text{“}end", haystack:array))$	$trl(e_n:t_n)$ 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	$trl((\text{“}basename", e:string))$	the filename if e can be recognized as an absolute file path; a symbol value in Z3 with the type of string otherwise.

TABLE II: Examples of rules to translate PHP-based constraints into Z3-based constraints.

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 [14] to verify the last two constraints.

```
1 $path_array = wp_upload_dir();
2 $filename = $_FILES['upload_file']['name'];
3 $pathAndName = $path_array['path'] . "/" . $filename;
4 if(strlen($filename) > 5){
5     move_uploaded_file($_FILES["upload_file"]['tmp_name'],
          $pathAndName);}
```

Listing 4: An example PHP file with unrestricted file upload vulnerability

Listing 4 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 III-B4), $_FILES['upload_file']['tmp_name'] returns (i.e., $(".", s_{name}, s_{ext})$), 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 $(".", s_{path}, (".", "/", (".", s_{name}, s_{ext})))$.

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:

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

D. 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 translation 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 II. 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, $trl((\text{“}basename", e:string))$ 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 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)

IV. Evaluation

We have implemented *UChecker* with approximately 30K LoC in PHP. *UChecker* leverages PHP-Parser [15] to parse source code for AST generation and Z3 [14] 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.

A. Ground-Truth-Available Experiments

We have collected totally 13 publicly-reported vulnerable PHP applications. The vulnerable samples include 11 WordPress plugins [16], a Joomla extension (i.e., Joomla-Bible-study 9.1.1 [17]), and a Drupal module (i.e., Avatar Uploader 6.x-1.2 [18]). 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 III 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 III 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 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 III 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 III). 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 III. 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 5 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 5: add_action("admin_menu", 'func') makes func only accessible to the administrator in the plugin Event Registration Pro Calendar 1.0.2

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).

B. 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

	System	LoC	% of LoC Analyzed	Paths	Objects	Objects / Path	Memory (MB)	Time (second)	Detected As Vulnerable
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 III: 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.

detection measures for these 3 plugins are presented in the bottom 3 rows in Table III.

WooCommerce is a `WordPress` eCommerce plugin, which by itself supports third-party plugins to extend its functionality. `WooCommerce Custom Profile Picture 1.0` [19] 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 6. 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 if($_FILES['profile_pic']){
2     $picture_id = wc_cus_upload_picture($_FILES['
          profile_pic']);
3 }
4 function wc_cus_upload_picture( $foto ) {
5     $profilepicture = $foto;
6     $wordpress_upload_dir = wp_upload_dir();
7     $new_file_path = $wordpress_upload_dir['path'] . '/' .
           $profilepicture['name'];
8     //...
9     if( move_uploaded_file( $profilepicture['tmp_name'],
           $new_file_path ) ) {
10        //...
11        }
12 }
```

Listing 6: Vulnerable Code of `WooCommerce 1.0 Custom` Profile Picture

File Provider 1.2.3 [20] 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 administrator 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 7. 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 function upload_file(){
2     $folderId = sanitize_text_field($_POST['folderId']);
3     $folderPath = get_file_path($folderId); // User
          declared method: get the upload path
4     $nome_final = $_FILES['userFile']['name'];
5     if (!move_uploaded_file($_FILES['userFile']['tmp_name'
          ],
6         $folderPath . '/' . $nome_final)) {
7         echo '<div class="error">...</div>';
8     } }
```

Listing 7: Vulnerable Code of `File Provider 1.2.3`

WP Demo Buddy 1.0.2 [21] is used to create demo instances, whose relevant source code is displayed in Listing 8. 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 function file_Upload($type)
2 {
3     global $wpdb;
4     $upload_dir = get_option('wp_demo_buddy_upload_dir');
5     $ext = pathinfo($_FILES[$type]['name'],
          PATHINFO_EXTENSION);
6     if ($ext !== 'zip') return;
7     $info = pathinfo($_FILES[$type]['name']);
8     $newname = time() . rand() . '_' . $info['basename'] .
          '.php';
9     $target = $upload_dir . $newname;
10    move_uploaded_file($_FILES[$type]['tmp_name'], $target
          );
11    $ret = array($newname, $info['basename']);
12    return $ret;
13 }
```

Listing 8: Vulnerable Code of `WP Demo Buddy 1.0.2`

C. Comparison With Other Detection Solutions

We have experimented with two publicly available PHP vulnerability scanners including RIPS [22], [23] and WAP [24], 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.

V. Related Work

Static program analysis has been adopted to detect variety of vulnerabilities [25], [26], [27], [9], [28], [29] [30], [31], [23]. Specifically, Zheng et. al [25], [26] and Xie et.al [27] leverage static program analysis to detect PHP web applications that are vulnerable to SQL injection and XSS attacks. Son et. al [9] proposed a method to identify PHP web applications with semantic vulnerabilities such as infinite loops and the missing of authorization checks. Dahse et. al [28] designed a system to detect SQL injections and XSS using data flow analysis. Barth et. al [29] designed a system to detect XSS attacks by analyzing the structure of the content submitted to the server. Compared to these methods, *UChecker* focuses on a different vulnerability, the unrestricted file upload vulnerability. Staicu et. al [31] studied Node.js applications vulnerable to injection attacks that exploit `exec` or `eval` APIs. Similar to *UChecker*, 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 at detecting different vulnerabilities. Dahse et. al [23] 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 [32] 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 [33] conducted benchmark-based assessments to compare capabilities of publicly-available static vulnerability detection tools.

Many methods detect vulnerable PHP applications [34], [35], [36] and JavaScript applications [37] using dynamic program analysis, which fundamentally differs from the analysis methodology employed by *UChecker*. Although dynamic analysis tends to be more efficient compared to static analysis, it commonly suffers from low coverage of program paths and high uncertainties introduced by runtime environment. In *UChecker*, we have compensated for the possible performance bottleneck rooted-in static analysis by designing a vulnerability-oriented locality analysis.

A few existing projects [38], [39], [5], [28], [40] involve the study of unrestricted file upload vulnerabilities. For example, Canali et. al built a honey pot [40] to study actual malicious upload attempts. Despite insights gained from these studies, a systematic solution, which can detect vulnerable web applications before their deployment, is still missing. *UChecker* aims to deliver this very capability.

VI. 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 IV-A, 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.

VII. Conclusion

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.

REFERENCES

[1] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis, "No honor among thieves: A large-scale analysis of malicious web shells," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 1021–1032. [Online]. Available: https://doi.org/10.1145/2872427.2882992

[2] W. contributors, "Unrestricted file upload," 2018, [Online; accessed 22-July-2018]. [Online]. Available: https://www.owasp.org/index.php/Unrestricted_File_Upload

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

[4] B. Williams, D. Damstra, and H. Stern, *Professional WordPress: design and development*. John Wiley & Sons, 2015.

[5] I. Riadi and E. I. Aristianto, "An analysis of vulnerability web against attack unrestricted image file upload," *Computer Engineering and Applications Journal*, vol. 5, no. 1, pp. 19–28, 2016.

[6] Patrick, "Unrestricted file upload (rce)," 2018, [Online; accessed 22-July-2018]. [Online]. Available: https://hackerone.com/reports/343726

[7] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[8] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[9] S. Son and V. Shmatikov, "Saferphp: Finding semantic vulnerabilities in php applications," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '11. New York, NY, USA: ACM, 2011, pp. 8:1–8:13. [Online]. Available: http://doi.acm.org/10.1145/2166956.2166964

[10] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 652–661.

[11] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[12] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998.

[13] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[14] Y. Zheng, X. Zhang, and V. 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*. ACM, 2013, pp. 114–124.

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

[16] B. Williams, O. Richard, and J. Tadlock, *Professional WordPress Plugin Development*. Wrox Press Ltd., 2011.

[17] C. Report, "Unrestricted file upload vulnerability in the joomla content editor," 2018, [Online; accessed 30-July-2018]. [Online]. Available: https://www.cvedetails.com/cve/CVE-2012-2902/

[18] ——, "Unrestricted file upload vulnerability in the avatar uploader module before 6.x-1.3," 2018, [Online; accessed 30-July-2018]. [Online]. Available: https://www.cvedetails.com/cve/CVE-2015-2087/

[19] Ecomerciar, "Woocommerce custom profile picture," 2017, [Online; accessed 30-July-2018]. [Online]. Available: https://wordpress.org/plugins/woo-custom-profile-picture/

[20] dimdavid, "File provider 1.2.3," 2017, [Online; accessed 30-July-2018]. [Online]. Available: https://wordpress.org/plugins/file-provider/

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

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

[23] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *NDSS*, vol. 14. Citeseer, 2014, pp. 23–26.

[24] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[25] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 32–41.

[26] W. Gary and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 171–180.

[27] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *USENIX Security Symposium*, vol. 15, 2006, pp. 179–192.

[28] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *USENIX Security Symposium*, 2014, pp. 989–1003.

[29] 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*, May 2009, pp. 360–371.

[30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 513–528.

[31] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js," in *Network and Distributed System Security (NDSS)*, 2018.

[32] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of html generation errors in php applications using string constraint solving," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 277–287.

[33] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking static analysis tools for web security," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1159–1175, 2018.

[34] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 249–260.

[35] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 261–272.

[36] A. Petukhov and D. Kozlov, "Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing," *Computing Systems Lab, Department of Computer Science, Moscow State University*, pp. 1–120, 2008.

[37] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications." in *NDss*, 2010.

[38] N. Uddin and M. 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*. IEEE, 2016, pp. 1–5.

[39] O. Andreeva, S. Gordeychik, G. Gritsai, O. Kochetova, E. Potseluevskaya, S. I. Sidorov, and A. A. Timorin, "Industrial control systems vulnerabilities statistics," *Kaspersky Lab, Report*, 2016.

[40] D. Canali and D. 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)*, 2013, pp. n–a.