10-7-2021

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

Jin Huang

Junjie Zhang
Wright State University - Main Campus, junjie.zhang@wright.edu

Jialun Liu

Chuang Li

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

Jin Huang
Wright State University
Dayton, Ohio, USA
huang.70@wright.edu

Junjie Zhang
Wright State University
Dayton, Ohio, USA
junjie.zhang@wright.edu

Jialun Liu
Wright State University
Dayton, Ohio, USA
liu.150@wright.edu

Chuang Li
Wright State University
Dayton, Ohio, USA
li.213@wright.edu

Rui Dai
University of Cincinnati
Cincinnati, Ohio, USA
rui.dai@uc.edu

ABSTRACT

Unrestricted file upload vulnerabilities enable attackers to upload malicious scripts to a web server for later execution. We have built a system, namely *UFuzzer*, to effectively and automatically detect such vulnerabilities in PHP-based server-side web programs. Different from existing detection methods that use either static program analysis or fuzzing, *UFuzzer* integrates both (i.e., static-fuzzing co-analysis). Specifically, it leverages static program analysis to generate executable code templates that compactly and effectively summarize the vulnerability-relevant semantics of a server-side web application. *UFuzzer* then "fuzzes" these templates in a local, native PHP runtime environment for vulnerability detection. Compared to static-analysis-based methods, *UFuzzer* preserves the semantics of an analyzed program more effectively, resulting in higher detection performance. Different from fuzzing-based methods, *UFuzzer* exercises each generated code template locally, thereby reducing the analysis overhead and meanwhile eliminating the need of operating web services. Experiments using real-world data have demonstrated that *UFuzzer* outperforms existing methods in either efficiency, or accuracy, or both. In addition, it has detected 31 unknown vulnerable PHP scripts including 5 CVEs.

CCS CONCEPTS

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

KEYWORDS

web security, vulnerability, detection, fuzzing, program analysis

ACM Reference Format:
Jin Huang, Junjie Zhang, Jialun Liu, Chuang Li, and Rui Dai. 2021. *UFuzzer*: Lightweight Detection of PHP-Based Unrestricted File Upload Vulnerabilities Via Static-Fuzzing Co-Analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471621.3471859

RAID '21, October 6–8, 2021, San Sebastian, Spain

ACM ISBN 978-1-4503-9058-3/21/10…$15.00
https://doi.org/10.1145/3471621.3471859

1 INTRODUCTION

Web applications with unrestricted file upload vulnerabilities allow attackers to upload a file with malicious code, which can be later executed on the server to enable various attacks such as information exfiltration, spamming, phishing, and spreading malware. Such vulnerabilities represent a pervasive threat to all web servers [8], particularly those written in scripting languages (e.g., `PHP`, `ASP`, and `JavaScript`), which require no file system permission to execute an uploaded file with compatible scripting extensions (e.g., "`.php`", "`.asp`", and "`.js`").

Detecting such vulnerabilities is of significant importance. A few systems [9, 11, 15, 17, 20] are designed to address this challenge using either static program analysis or fuzzing. Specifically, *RIPS* [9, 11] and *WAP* [20] use static taint analysis to trace whether a sensitive API is contaminated by untrustworthy inputs. These two systems are applicable to detect unrestricted file upload vulnerabilities when a file write API is considered as critical API and the super-global variable for the uploaded file is used as the untrustworthy input. *WAP* [20] also integrates machine learning into its detection process. Unfortunately, it is challenging for static taint analysis to model sanitization actions that are enforced along data flows, thereby limiting their accuracy in detecting unrestricted file uploading vulnerabilities.

Compared to *RIPS* and *WAP*, *UChecker* [15] advances the detection capability by semantically modeling applications' file uploading behaviors. Specifically, through symbolically interpreting a PHP application, *UChecker* generates constraints that model conditions to successfully exploit an unrestricted file uploading vulnerability. These constraints, originally expressed in PHP, are next translated into symbolic constraints written in the language for a satisfiability solver [12], which will be eventually evaluated by this solver. Despite the fact *UChecker* has demonstrated promising detection performance, its applicability heavily depends on the PHP-to-solver constraint translation, which is guided by manually engineered rules. 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 [19], which limits *UChecker*'s capabilities to model and solve sophisticated PHP-based regex operations.

FUSE [17] 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 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.

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

- Effective and Efficient: *UFuzzer* should achieve high detection accuracies with low 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 the 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 of 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*, 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 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, resulting in 5 CVEs.

The remaining of our paper is organized as follows. We introduce the vulnerability background in Section 2. Section 3 presents the system design. Section 4 illustrates evaluation results and Section 5 presents the detection of new vulnerable programs. The related work is discussed in Section 6. Section 7 elaborates the limitation and potential solutions of the current design. Section 8 concludes.

2 BACKGROUND

2.1 Unrestricted File Upload

Listing 1 shows a code snippet that introduces the unrestricted file upload vulnerability to a server-side PHP application. In this snippet, the server receives a file from a remote client through the `$_FILES` superglobal variable, which is 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 is indexed by "name", "type", "tmp_name", "error", and "size".

```
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 1: A code snippet for unrestricted file upload vulnerability

Once the webserver receives a file, it automatically saves the file and assigns it with a temporal file name, where such file name is indexed by "tmp_name" (i.e., `$_FILES['newfile']['tmp_name']` in Listing 1). The actual name of the uploaded file, which is given by the user on the client-side, is indexed by "name" (i.e., `$_FILES['newfile']['name']` in Listing 1). The server-side PHP application can therefore access the saved file using the temporal file name. One common operation is to rename the file based on its actual name and the PHP built-in API `move_uploaded_file(`e_{src},`$e_{dst}$`)` is frequently used to accomplish this goal, where $e_{src}$ and $e_{dst}$ denote the source and destination file name, respectively. In Listing 1, `move_uploaded_file($tmpFile,$fName)` is used to rename the temporal file using its file name offered by the external user. Other than `move_uploaded_file()`, another similar function, namely `file_put_content()`, is also commonly used.

The vulnerability arises sice the code in Listing 1 does not disable the upload of executable files (i.e., those with executable extensions such as ".php"). Hence, if an attacker submits an executable script (e.g., "test.php") this code will faithfully store this script in the target server with an executable file name (e.g., "test.php").

2.2 Heap Graphs

We employ the intermediate representation (IR) of PHP programs, namely heap graphs, which are first proposed in [15], to bootstrap our analysis. Heap graphs can be generated using the interpreter proposed in [15] by symbolically interpreting the abstract syntax trees (AST) of a PHP-based server-side web application. Rather than

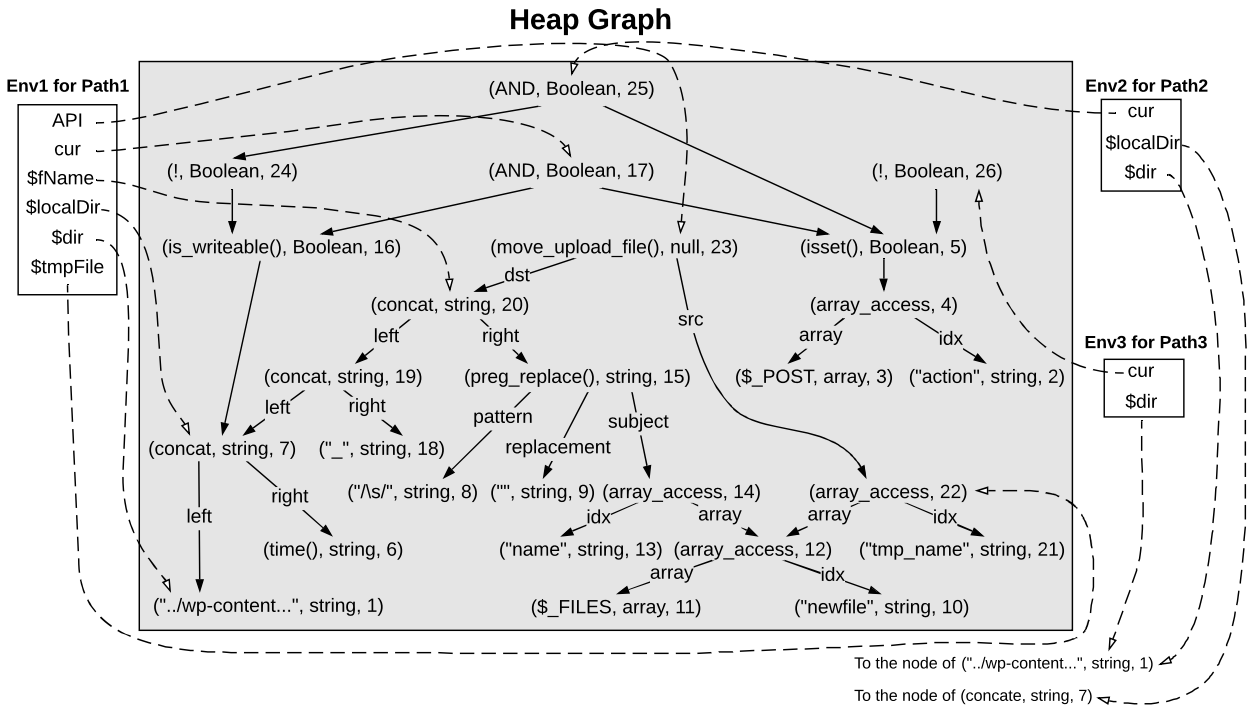

Figure 1: The heap graph for the sample code in Listing 1

symbolically interpreting AST for the *entire* web application, the interpreter performs vulnerability-oriented *locality analysis* [15] to identify a sub-AST for symbolic interpretation, aiming at mitigating the path explosion challenge. The sub-AST corresponds to statements that are relevant to unrestricted file upload vulnerabilities; irrelevant statements will be ignored. In this work, we have extended the symbolic interpreter [15] by interpreting essential object-oriented grammars including i) class declaration, ii) object initialization, and iii) the invocation of member functions. Specifically, our interpreter evaluates each class, recording its attributes, method names, and methods' bodies. We track each initialized object and its corresponding class. If a method of an object is invoked, the interpreter will identify its method body and evaluate it.

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 the 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 a variable to a node; if a vulnerability-related API appears in this path, it uses a special variable, namely *API*, to track the node of that API.

A *heap graph* is formally defined as a directed acyclic graph G, where $G = \{C, S, FUNC, OP, L, T, O_C, O_S, O_{FUNC}, O_{OP}, Edge, \mathcal{E}\}$:

- C is a set of concrete values.
- S is a set of symbolic values.
- $FUNC$ is a set of PHP built-in functions (i.e., APIs).
- OP is a set of operators (e.g., "+", "-", and ".").
- L is a set of labels, where every object in G has a *unique* label.
- T is a set of known data types (e.g., boolean and integer) and an unknown type $\perp$.
- $O_C \subset C \times T \times L$ is a set of objects (i.e., nodes) for concrete values.
- $O_S \subset S \times T \times L$ is a set of objects (i.e., nodes) for symbolic values.
- $O_{FUNC} \subset FUNC \times T \times L$ is a set of objects (i.e., nodes) for built-in functions.
- $O_{OP} \subset Op \times T \times L$ is a set of objects (i.e., nodes) for operators.
- $Edge \subset \{(l_1, l_2) | (x, t_1, l_1) \in O_{FUNC} \cup O_{OP}$ *and* $(y, t_2, l_2) \in O_C \cup O_S \cup O_{FUNC} \cup O_{OP}\}$.
- $\mathcal{E} = \{Env_1, \ldots, Env_i, \ldots, Env_n\}$ is a set of environments of all execution paths of a program.
  - $Env_i = \{Var_i, Map_i, cur_i, API_i\}$ is the environment for the i-th execution path.
  - Var_i is a set of variable names.

- $Map_i \subset Var_i \times L$ establishes a mapping between a variable name and an object.
- $cur_i \in \{l|(x,t,l) \in O_C \cup O_S \cup O_{FUNC} \cup O_{OP}\}$. cur_i represents the reachability constraint.
- $API_i \in \{l|(x,t,l) \in O_{FUNC}\}$. API_i represents the node of the API that is of security concerns.

Figure 1 presents the heap graph for the vulnerable code snippet in Listing 1 towards the end of this program. Each node in this graph is represented by a 3-tuple of (*name*, *type*, *id*). The "*name*" in this 3-tuple is the name of an operator, the name of a built-in function, a concrete value, or a symbolic value. The "*type*" indicates the type of the result for an operator or a built-in function; it also indicates the type of a concrete value or a symbolic value; it can be assigned with $\perp$ if the type is unknown. The "*id*" stores the node identifier, which is unique for each node in a heap graph. This example has three execution paths in total, resulting in three environments (see $Env1$, $Env2$, and $Env3$ in the graph). Each environment maps variables to their corresponding graph nodes. For example, the evaluation result of `$dir.time()` (":" in PHP is for concatenation) points to the node of (*concat*, *string*, 7). By traversing the graph from this node towards the bottom of the graph, we can derive the s-expression-based value of `$dir.time()` as (*concat*, "*../wpcontent/plugin/upload/*", *time*). The statement `$localDir = $dir.time()` results an association between the variable `$localDir` and the evaluation result of `$dir.time()`, where the variable `$localDir` is kept in the environment.

The *cur* in an environment points to the node from which the reachability constraint can be derived. For example, by traversing from the *cur* node in $Env1$, one can derive its reachability in the form of s-expression as (*AND* (*is_writeable*) (*isset*)), where nodes (*is_writeable*, *Boolean*, 16) and (*isset*(), *Boolean*, 5) can be further resolved into s-expressions by traversing towards the bottom of the heap graph.

Since the vulnerability-related built-in API `move_uploaded_file(`e_{src}, e_{dst}`)` only appears in the first execution path (i.e., $Env1$), $Env1$ has its *API* points to the node of (*move_upload_file*, *null*, 23), whose "src" and "dst" edges point to e_{src} and e_{dst}, respectively.

3 SYSTEM DESIGN

Figure 2 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 [15] 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 2). 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 2.

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, uninitialized 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 2. *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.

3.1 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.

The "src" edge originated from a `move_uploaded_file()` node points to the source of the file to be permanently saved. This 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 23) in Figure 1 through its "src" edge, indicating that the source file of the `move_uploaded_file()` API is tainted by external inputs.

3.2 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.

Uninitialized Variables: Our symbolic interpreter performs vulnerability-oriented locality analysis [15] 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 known a priori. Specifically, $_FILES is a pre-structured array that is indexed by 5 keys including "name", "type", "tmp_name", "error", and "size"; these 5 keys represent the original file name, the type information, the temporal filename, the error information, and the size of the file. Therefore, we 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*

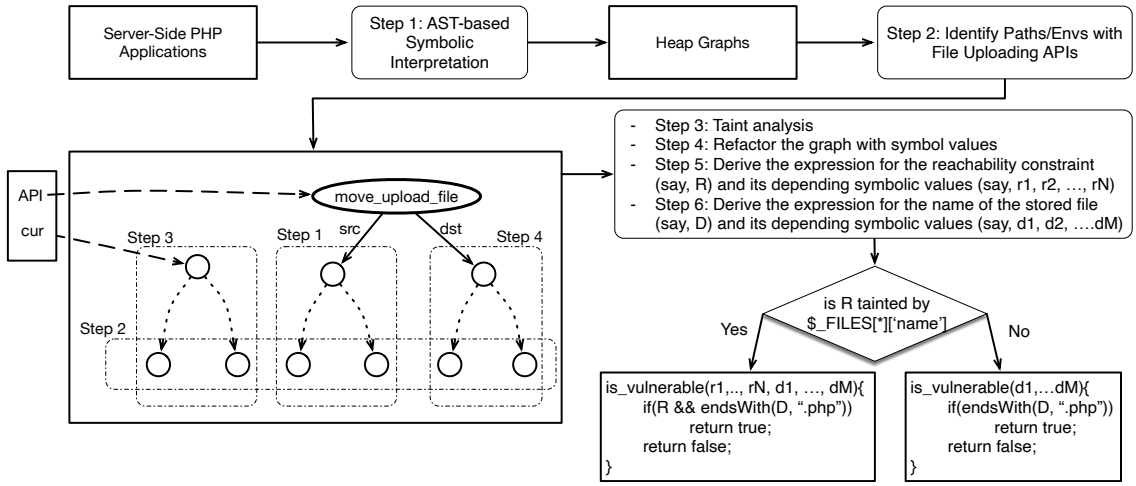

Figure 2: The architectural overview of *UFuzzer*

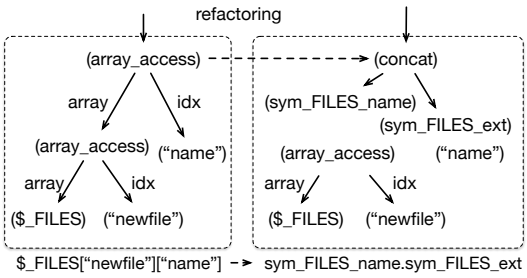

Figure 3: An example of refactoring an array_access node associated with the $_FILES superglobal variable. The sub-tree rooted in the top *array_access* denotes $_FILES["newfile"]["name"]. This *array_access* node will be replaced by a node that concatenates two symbolic values of the filename and the extension, respectively.

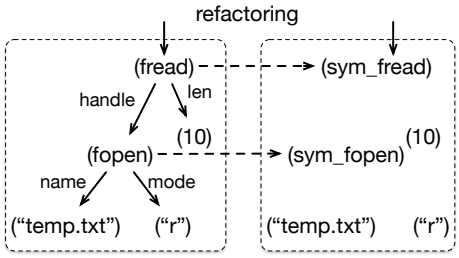

Figure 4: An example of refactoring a `fread` node with a node of symbolic value (i.e., *sym_fread*).

global variable; ii) its "index" edge points to one of the 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 the 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 3 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 represent $_FILES["newfile"]["name"]. Therefore, *UFuzzer* 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.

Operation and Validation APIs: Some APIs in the server-side web program can only function in a properly configured run-time environment, making automatic analysis extremely challenging. These APIs are often used for operations of networking, databases, and file access, 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 4 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. Hence, we symbolize these functions regardless of their arguments. For example, any node of `isset()` in the heap graph

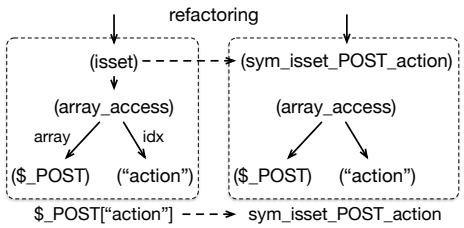

Figure 5: An example of refactoring a `isset` node with a node of symbolic value (i.e., $sym\_isset\_POST\_action$).

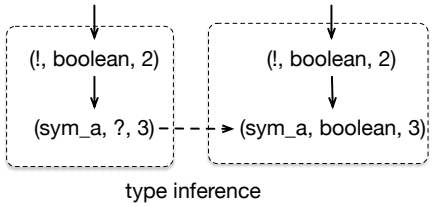

Figure 6: An example of type inference. Inferring the type of the symbolic node using its immediate operator node.

will be replaced using a symbol node with boolean type. Figure 5 presents an example. The node of `isset()`, which is corresponding to the expression of `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 6 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.

3.3 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 bound to the node that represents the reachability constraint and we name this node as v_{reach}. The *API* variable in an environment is bound 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 7 and 8 present sub-trees for v_{reach} and $v_{filename}$ derived from Figure 1, respectively.

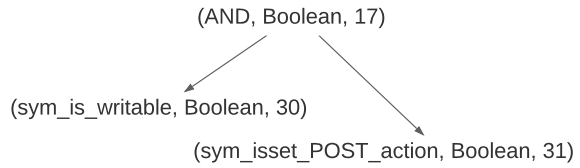

Figure 7: The sub-tree for the reachability constraint derived from Figure 1 after the node `(is_writable, Boolean, 16)` is symbolized into `(sym_is_writable, Boolean, 30)`

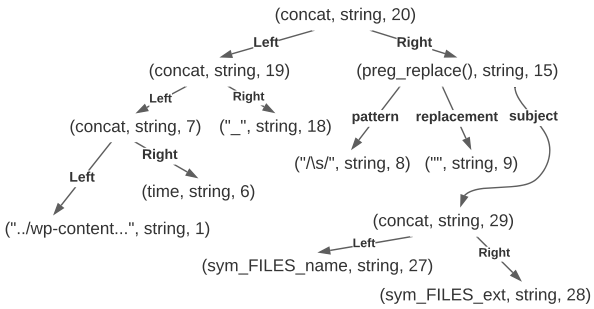

Figure 8: The sub-tree for the filename derived from Figure 1

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 [22]) 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

```
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_*(...)").

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.

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.

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})".

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*).

3.4 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

```
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

```
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 the 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 2 and Listing 3 present two fuzzing templates for Listing 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 $sym_is_writable are variables to be mutated, where the first two have the type of strings and the last two are boolean. The `pathinfo` function used in fuzzing templates is a PHP built-in API for returning the extension of a file name with parameter `PATHINFO_EXTENSION`. The "*in_array*($*ext*, *array*('*php*')" is to check whether the extension is 'php', which can be easily extended to include additional sensitive extensions (e.g., 'jsp').

```php
<?php
    function is_vulnerable($sym_file_name,
                           $sym_file_ext,
                           $sym_isset_POST_action,
                           $sym_is_writable)
    {
        $exp_reach = $sym_isset_POST_action and
                     $sym_is_writable;
        $exp_fileName = "../wp-content/plugins/upload/" .
                        time() . "_" .
                        preg_replace("/\s/", "",
                                     $sym_file_name .
                                     $sym_file_ext);
        if($exp_reach){
            $ext = pathinfo($exp_fileName,
                            PATHINFO_EXTENSION);
            if(in_array($ext, array('php'))){
                return TRUE;
            }
        }
        return FALSE;
    }
?>
```

Listing 2: The Fuzzing Template With Reachability Evaluated for Listing 1

```php
<?php
    function is_vulnerable($sym_file_name,
                           $sym_file_ext)
    {
        $exp_fileName = "../wp-content/plugins/upload/" .
                        time() . "_" .
                        preg_replace("/\s/", "",
                                     $sym_file_name .
                                     $sym_file_ext);

        $ext = pathinfo($exp_fileName,PATHINFO_EXTENSION);
        if(in_array($ext, array('php'))){
            return TRUE;
        }
        return FALSE;
    }
?>
```

Listing 3: The Fuzzing Template Without Reachability Evaluated for Listing 1

Since the reachability constraint of Listing 1 is not tainted by the name of the uploaded file (i.e., $_FILES[*]['name']), the fuzzing template in Listing 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 Listing 3 compared to that in Listing 2 without undermining the detection accuracy.

3.5 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 [23] as a drop-in fuzzer to mutate string values for `$sym_file_name`. We use different mutators in PHP Fuzzer [23] such as `EraseBytes`, `InsertByte`, `ChangeByte`, `ChangeBit`, and `ChangeASCIIInt`. For each mutator, we iterate for 50 times with a length between 5 and 70.
- An argument with the boolean type (e.g., `$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.

4 GROUND-TRUTH-AVAILABLE EVALUATION

We have implemented *UFuzzer* with approximately 28K LoC, which reuses the AST-based symbolic interpreter in [15] with minor improvements. We leveraged PHP-Parser [22] 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 27 publicly-reported vulnerable PHP applications. These 27 samples consists of 13 known samples used in [15], 3 new vulnerable applications detected by UChecker [15], and 11 more vulnerable samples we have recently collected. We have identified 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* [15], *RIPS* [9, 11], *WAP* [20], and *FUSE* [17]. We have deployed them in the same running environment of *UFuzzer*.

Evaluation Results: Table 1 presents the detection results and the running time for *UFuzzer*, *UChecker*, *RIPS*, *WAP*, and *FUSE*. The second and third columns in Table 1 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 1 summarize the detection rates and the false positive rates.

UFuzzer is 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.

```
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 4: *UChecker* fails to correctly model the regex operation in the PDW Media File Browser plugin

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 4, which is challenging to be effectively modeled and solved by satisfiability solvers (and hence *UChecker*). In contrast, *UFuzzer* can easily execute this operation in a native PHP run-time environment.

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 1 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.

	System	LoCs	Fuzzing Templates	Detected by UFuzzer	Detected by 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	5,581	1	✔ (0.78s)	✔ (1.12s)	✔ (0.10s)	✔ (1.00s)	✔ (1.38h)
	Estatik 2.2.5	9,823	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 Sales Countdown Timer 2.8.0	28,643	6	✔ (4.91s)	✔ (5.01s)	✔ (0.42s)	✖ (1.00s)	✔ (0.33h)
	N-Media Website Contact Form with File Uploader 1.3.4	1,857	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	2,337	2	✔ (0.35s)	✔ (0.70s)	✔ (0.10s)	✔(0.94s)	✖ (1.80h)
	PDW Media File Browser 1.1	20,664	1	✔ (3.59s)	✖ (4.01s)	✔ (3.68s)	✔ (1.00s)	✖ (3.03h)
	Ip Blocker Lite 10.2	5,574	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	3,560	8	✔ (0.67s)	✔ (1.21s)	✔ (0.09s)	✖ (1.00s)	✔ (0.33h)
	WooCommerce Checkout Manager 4.2.5	14,942	8	✔ (1.70s)	✔ (0.96s)	✔ (0.70s)	✖ (1.00s)	✔ (0.33h)
	WP Marketplace 2.4.1	13,956	1	✔ (2.6s)	✔ (2.60s)	✔ (0.32s)	✖ (1.00s)	✔ (0.33h)
	wp-Powerplaygallery 3.3	2,752	416	✔ (1.33s)	✔ (2.78s)	✔ (0.07s)	✖ (0.86s)	✔ (2.21h)
	WP Seo Spy 3.1	3,431	2	✔ (0.50s)	✔ (0.57s)	✔ (0.07s)	✔ (1.00s)	✔ (0.81h)
	WP Demo Buddy 1.0.2	2,208	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	13,358	64	✔ (1.79s)	✔ (2.98s)	✔ (0.30s)	✔ (2.00s)	N/A
	Asset Manager 0.2	3,784	1	✔ (0.22s)	✔ (0.81s)	✔ (0.04s)	✖ (0.87s)	N/A
	Simple Ad Manager 2.5.94	1,937	4	✔ (1.24s)	✔ (5.35s)	✔ (0.33s)	✔ (1.00s)	N/A
	SpamTask 1.3.6	3,434	2	✔ (0.61s)	✔ (0.61s)	✔ (0.15s)	✔ (1.00s)	N/A
	MailCWP 1.100	4,3191	1	✔ (5.01s)	✔ (5.80s)	✔ (1.26s)	✔ (2.00s)	N/A
	Joomla-Bible-study 9.1.1	87,626	16	✔ (13.70s)	✔ (13.72s)	✔ (1.31s)	✔ (1.00s)	N/A
	Cimy User Extra Fields 2.3.8	9,432	100000+	✖ (N/A)	✖ (N/A)	✔ (0.97s)	✖ (1.00s)	✔ (5.50h)
Non-Vulnerable	Fullscreen background slider 1.1	8,324	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	2,873	3	✖ (9.95s)	✖ (0.91s)	✔ (0.07s)	✖ (1.00s)	✖ (1.81h)
	BackupGuard 1.1.46	10,509	6	✖ (14.07s)	✖ (3.01s)	✔ (7.79s)	✖ (1.00s)	✖ (4.28h)
	WooCommerce Catalog Enquiry 3.1.0 (Fixed version)	3,545	2	✖ (6.88s)	✖ (1.09s)	✔ (0.09s)	✖ (1.00s)	✖ (1.43h)
	Telegram-chat 3.0.4	2,665	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	47,689	40	✖ (190.81s)	✖ (25.81s)	✔ (39.46s)	✖ (14.00s)	✖ (1.41h)
	Morbits SMS 1.0	71,787	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	1,254	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	10,730	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 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*.

5 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 2 presents the detection results. *UFuzzer* and *UChecker* together detect 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 identifies 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 2). Manual analysis reveals that some APIs of false negatives introduced by *UChecker* 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

No.	Application	UFuzzer	Vuln Source File : Line No.	UChecker	Verification Method	Root Cause	Admin Required?	CVE
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	CVE-2021-24223
23	Testimonials King Light 0.1	✔	testimonial-king-form.php:38	✖	Code Review	MisAPI	No	
24	WP-Curriculo Vitae Free 6.1	✔	enviarCadastro.php:86	✖	Exploiting	LS	No	CVE-2021-24222
25	Easy Form Builder 1.0	✔	newForm.php:49	✔	Exploiting	LS	No	CVE-2021-24224
26	imagements 1.2.5	✔	imagements.php:127	✔	Exploiting	SInS	No	CVE-2021-24236
27	Event Banner 1.3	✔	admin_events.php:29	✖	Exploiting	LS	Yes	CVE-2021-24251
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 2: Detecting New Vulnerable Applications. *UFuzzer* detected 31 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".

successful exploitation. While these 13 applications require attackers to gain higher privileges, they may still lead to unintended behaviors that could be potentially misused. In fact, one of such examples, `Event Banner 1.3`, receives CVE-2021-24251.

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). Its vulnerable code is presented in Listing 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 6 shows the fuzzing template that successfully detects this vulnerability, which evaluates both the reachability condition and the file extension.

```php
<?php
if ($_FILES['file']['name']) {
    if (!$_FILES['file']['error']) {
        $name = md5(rand(100, 200));
        $ext = explode('.', $_FILES['file']['name']);
        $filename = $name . '.' . $ext[1];
        $destination = '/public/images/' . $filename;
        $location = $_FILES["file"]["tmp_name"];
        move_uploaded_file($location, $destination);
        //...
    } else{
      //...
    }
}
```

```php
?>
```

Listing 5: Vulnerable Code of `Basic Laravel CMS`

```php
function is_Vulnerable_0(string $sym_file_name,
                         string $sym_file_ext,
                         int $_FILES_file_error_symbol){
    $exp_reach = ($sym_file_name . $sym_file_ext and
                  !$_FILES_file_error_symbol);
    $funCall = explode('.', $sym_file_name . $sym_file_ext);
    $exp_filename = '/public/images/' . (md5(rand(100, 200)) .
                    '.' . $funCall[1]);
    if ($exp_reach) {
        $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
        if (in_array($ext, array('php'))) {
            return true;
        }
    }
}
```

Listing 6: A Fuzzing Template for `Basic Laravel CMS`

Misusing Sanitization APIs: Misusing Sanitization APIs: `BSK Files Manager 1.0.0` [5] is a `WordPress` plugin for file management. However, it does not guarantee that the file provided is a legitimate file extension and allows high privilege users to upload arbitrary files including ".php" or ".exe" files. Listing 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 guaranteeing to return a filename that is allowed to be uploaded [38]. Therefore, a file with the executable ".php" extension can still be uploaded. The fuzzing template that

successfully revealed this vulnerability is presented in Listing 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.

```
function bsk_files_manager_file_upload_file($file,
      $destination_name_prefix, /*...*/){
    if (!$file["name"]){
        return false;
    }
    if ( $file["error"] != 0 ){
        return false;
    }
    //...
    $destinate_file_name = $destination_name_prefix.'_'.
                           sanitize_file_name($file["name"]);
    $ret = move_uploaded_file($file["tmp_name"],
                              /*...*/.$this->_files_upload_folder
                              . $destinate_file_name);
    //...
    return $destinate_file_name;
}
```

Listing 7: Vulnerable Code in `BSK Files Manager 1.0.0`

```
function is_vulnerable($sym_files_upload_folder,
                       $bsk_files_manager_file_id,
                       $sym_file_name, $sym_file_ext,
                       $sym_FILES_error,
                       /*...*/)
{
$exp_reach = (!($sym_file_name . $sym_file_ext) and
              $sym_FILES_error != 0 and /*...*/);
$exp_filename = /*...*/ . $sys_files_upload_folder .
                trim($bsk_files_manager_file_id) . '_' .
     sanitize_file_name($sym_file_name . $sym_file_ext);
if ($exp_reach) {
   $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
   if ($ext == 'php') {
       return true;
   }
}
return false;
}
```

Listing 8: A Fuzzing Template for `BSK Files Manager 1.0.0`

Sanitizing Incorrect Sources: `Imagements 1.2.5` [37] is a `WordPress` plugin that supports a visitor to leave an image-based comment in users' blogs. Listing 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 vulnerability and uses 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 the client's request. Since an attacker has 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 10.

```
add_action('comment_post', 'imagements_formverwerking');
add_filter('preprocess_comment', 'imagements_verify_post_data');
function imagements_formverwerking(){
    if(isset($_POST['checkbox'])){
        $name = $_FILES['image']['name'];
        //...
        move_uploaded_file($_FILES["image"]["tmp_name"],
                           PATH . '/images/' . $name);
    }
}
function imagements_verify_post_data($commentdata){
    if(isset($_POST['checkbox'])) {
        if($_FILES['image']['name'] != null) {
            if($_FILES["file"]["error"] > 0) {
                //...
            } else {
                if(!($_FILES['image']['type'] == 'image/x-png'||
                     $_FILES['image']['type'] == 'image/pjpeg'||
                     $_FILES['image']['type'] == 'image/jpeg' ||
                     $_FILES['image']['type'] == 'image/jpg'  ||
                     $_FILES['image']['type'] == 'image/png')) {
                    wp_die('this_file_is_no_image...');
                }
            }
        }
        //...
    }
}
```

Listing 9: Vulnerable Code in `Imagements 1.2.5`

```
function is_Vulnerable(string $sym_const_PATH,
                       string $sym_file_name,
                       string $sym_file_ext){
    $exp_filename = $sym_const_PATH . '/images/' .
                    ($sym_file_name . $sym_file_ext);
    $ext = pathinfo($exp_filename, PATHINFO_EXTENSION);
    if (in_array($ext, array('php'))) {
        return true;
    }
    return false;
}
```

Listing 10: A Fuzzing Template for `Imagements 1.2.5`

Sanitizing at the Client: `User-Management-PHP-MYSQL` [25] is an open-source web application collected from GitHub. Listing 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 12 presents the fuzzing template that reveals this vulnerability.

```
<?php
if(isset($_POST['submit'])){
    $file = $_FILES['image']['name'];
    $final_file = str_replace('_', '-', strtolower($file));
    if (move_uploaded_file($_FILES['image']['tmp_name'],
                           "images/" . $final_file)) {
        //...
    }
}
?>
<!doctype html>
<html lang="en" class="no-js">
<!--...-->
function validate() {
    var extensions = new Array("jpg", "jpeg");
    var final_ext = // Get the file extensions by JavaScript
    // return true if the final_ext is in extensions
    // return false otherwise
}
<form method="post" <!--...--> onSubmit="return_validate();">
    <!--...-->
    <button <!--...--> type="submit">Register</button>
</form>
<!--...-->
```

```
25 </html>
```

Listing 11: Vulnerable Code in User-Management-PHP-MYSQL

```
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 12: A Fuzzing Template for User-Management-PHP-MYSQL

6 RELATED WORK

The majority of web-server-oriented vulnerability detection methods employ static program analysis, where representative ones include [6, 9, 10, 14, 27, 28, 31, 35, 39]. These methods focus on a variety of well-known vulnerabilities such as SQL injection, XSS, and DoS. A few other methods [3, 4, 13, 18, 21, 34, 36] employ fuzzing or penetration testing to reveal vulnerabilities and failures in server-side web applications. A few other tools such as JBroFuzz [29], Wapiti [30], Wfuzz [16], Burp [24], and w3af [33] are also built to fuzzing HTTP requests 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 [2, 7, 10, 26, 32] analyzed unrestricted file upload vulnerabilities without delivering detection capabilities. RIPS [9, 11] and WAP [20], and UChecker [15] claim their capabilities of detecting unrestricted file upload vulnerabilities. RIPS [9, 11] leverages static taint analysis while WAP[20] 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-approximating to model the exploitation. The learning-based software defect detection commonly suffers from the data sparsity challenge [1], where program samples with the same vulnerability usually fall short for both quantity and diversity.

UChecker [15] and FUSE [17] are designed to detect unrestricted file upload vulnerabilities. UChecker symbolically interprets a PHP application and generates constraints that model conditions to successfully exploit a vulnerability. Unfortunately, the semantic gaps between PHP and the solver language introduce intrinsic challenges for constraint modeling. FUSE employs black-box fuzzing and it requires the tested web to be fully operable. Nevertheless, we acknowledge that the FUSE employs more types of exploitation inputs (i.e., PHP, XHTML, and JS) compared to the current implementation of *UFuzzer*, which currently focuses on PHP. While we plan to extend *UFuzzer* to cover more types of exploitation inputs in the future, *UFuzzer* can immediately complement UChecker and FUSE.

7 DISCUSSION

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 effort 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 interpret 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 on 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 the community's efforts towards this direction.

Focusing on the File Extension: UFuzzer investigates whether an uploaded file could have the ".php" extension, which represents 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 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.

Replicating Exploitations Using Template Inputs: UFuzzer "fuzzes" a fuzzing template that approximates the original code rather than the 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. Nevertheless, these template inputs will drastically help a fuzzer reduce the space of the fuzzing inputs. Again, our design associates a template input with an uninitialized variable, a superglobal variable, or the output of an operation/API. Therefore, an input associated with a superglobal variable can be directly used for exploitation; an input associated with an operation/API can be used to guide a fuzzer to generate the input(s) for this operation/API that will lead to wanted template inputs.

False Positives: Although *UFuzzer* has not introduced any false positives in our experiments using a large set of real-world applications, it is still possible for *UFuzzer* to introduce false positives for two major reasons. First, it does not model the configuration of web servers that could mitigate such vulnerabilities at run time. For example, a server could be configured (e.g, in `.htaccess` or `php.ini`) to completely disable file uploading for the entire web system. Nevertheless, we believe *UFuzzer*'s detected vulnerabilities, although only concerning flawed code, are useful to improve systems' fundamental security despite the fact they cannot be exploited due to runtime configuration. Second, a fuzzing template may fail to model

the context of uninitialized variables and symbolized operations, which however could involve sanitization actions. A potential solution to address this challenge is to trade efficiency for accuracy, such as interpreting more statements and modeling symbolized operations with finer granularity.

False Negatives: UFuzzer introduces one false negative in detecting new vulnerable applications. This is because the mutation process is unguided and therefore it cannot guarantee the generation of values that provably satisfy certain conditions in the fuzzing template. A 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.

8 CONCLUSION

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, contributing 5 CVEs.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages.

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

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

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

[5] bannersky. 2013. BSK Files Manager 1.0.0. [Online; accessed 30-July-2018].

[6] A. Barth, J. Caballero, and D. Song. 2009. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *2009 30th IEEE Symposium on Security and Privacy*. 360–371. https://doi.org/10.1109/SP.2009.3

[7] Davide Canali and Davide Balzarotti. 2013. Behind the scenes of online attacks: an analysis of exploitation behaviors on the web. In *Network and Distributed System Security (NDSS)*.

[8] Wikipedia contributors. 2018. Unrestricted File Upload. https://www.owasp.org/index.php/Unrestricted_File_Upload [Online; accessed 22-July-2018].

[9] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Network and Distributed System Security (NDSS)*.

[10] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security Symposium*.

[11] Johannes Dahse and Jörg Schwenk. 2010. RIPS-A static source code analyser for vulnerabilities in PHP scripts. In *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*.

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

[13] F. Duchene, R. Groz, S. Rawat, and J. Richier. 2012. XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 815–817.

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

[15] J. Huang, Y. Li, J. Zhang, and R. Dai. 2019. UChecker: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[16] 2014) Kali.org.(February 18. accessed on April 15, 2020. Wfuzz Package Description. http://tools.kali.org/web-applications/wfuzz.

[17] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. 2020. FUSE: Finding File Upload Bugs via Penetration Testing. In *Network and Distributed System Security (NDSS)*.

[18] L. Li, Q. Dong, D. Liu, and L. Zhu. 2013. The Application of Fuzzing in Web Software Security Vulnerabilities Test. In *2013 International Conference on Information Technology and Applications*. 130–133.

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

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

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

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

[23] Nikita Popov. 2019. PHP Fuzzer. *URl: https://github.com/nikic/PHP-Fuzzer (visited on 2020-12-09)* (2019).

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

[25] Ajay Randhawa. 2018. User-Management-PHP-MYSQL. https://github.com/scurite/User-Management-PHP-MYSQL [Online; accessed 05-Dec-2020].

[26] Imam Riadi and Eddy Irawan Aristianto. 2016. An Analysis of Vulnerability Web Against Attack Unrestricted Image File Upload. *Computer Engineering and Applications Journal* 5, 1 (2016), 19–28.

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

[28] Sooel Son and Vitaly Shmatikov. [n.d.]. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*.

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

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

[31] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node. js. In *Network and Distributed System Security (NDSS)*.

[32] Nasir Uddin and Mohammad Jabr. [n.d.]. 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*.

[33] w3af. (n.d.). accessed on April 15, 2020. w3af - Open Source Web Application Security Scanner. http://w3af.org/.

[34] Liu Qiang Wang Chunlei, Liu Li. 2014. Automatic fuzz testing of web service vulnerability. *IET Conference Proceedings* (2014).

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

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

[37] williewonka. 2012. Imagements. https://wordpress.org/plugins/imagements/ [Online; accessed 05-Dec-2020].

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

[39] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages.. In *USENIX Security Symposium*.