Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection

Yunhui Zheng Xiangyu Zhang
Department of Computer Science, Purdue University

***Abstract*—Remote code execution (RCE) attacks are one of the most prominent security threats for web applications. It is a special kind of cross-site-scripting (XSS) attack that allows client inputs to be stored and executed as server side scripts. RCE attacks often require coordination of multiple requests and manipulation of string and non-string inputs from the client side to nullify the access control protocol and induce unusual execution paths on the server side. We propose a path- and context-sensitive interprocedural analysis to detect RCE vulnerabilities. The analysis features a novel way of reasoning both the string and non-string behavior of a web application in a path sensitive fashion. It thoroughly handles the practical challenges entailed by modeling RCE attacks. We develop a prototype system and evaluate it on ten real-world PHP applications. We have identified 21 true RCE vulnerabilities, with 8 unreported before.**

I. Introduction

The Internet has been an important social and business platform for many daily activities, which are usually performed through web applications (apps). Once a web app is put online, it can be accessed and used by anyone around the world. A large volume of valuable and sensitive user data are processed and stored by web apps, making them attractive targets of security attacks. According to a risk report from IBM [3], in the past a few years, security attacks to web apps have become the dominant kind of attacks (i.e., 55% of all the attacks reported in 2010).

Among the web app attacks, *Remote Code Execution* (RCE) is one of the most harmful threats [2]. It takes advantage of defects of a web app to inject and execute malicious server-side script in the context of the targeted app. Consequently, the attacker could gain accesses to resources authorized to the app (e.g. user data in a database). According to a report from the *Open Web Application Security Project* (OWASP), PHP RCE is the most widespread PHP security issue since July 2004 and thus has been ranked the number one threat on the web apps security problem list [2]. We have also observed popular apps suffering from RCE defects. One of them has been downloaded for over six million times.

Essentially, RCE is a special kind of *Cross Site Scripting* (XSS) attacks. The root cause is the same as the typical XSS and SQL injection attacks, which is that invalid client-side inputs are undesirably converted to scripts and executed. However, RCE attacks are usually much more sophisticated. A successful RCE attack may require coordinations between multiple requests to more than one server side scripts. There may also be requirements on the session of these requests. In other words, the attacks are stateful, crossing multiple rounds of communication between a server and a client. Furthermore, it demands manipulating both the string and non-string parts of the client side inputs. In some cases, the inputs have to be so crafted that they are not even legitimate inputs allowed by the client side interface.

There have been a lot of works on detecting SQL injection [20], [31], [18], [33], XSS [9], [16], [21], [23], [26], [33], and HTTP request tampering [8], [11] attacks. However, due to the unique characteristics of RCE attacks, they fall short in detecting and confirming these attacks.

Dynamic tainting based techniques [11], [8] can monitor information flow inside web app execution to determine if any client side inputs can flow to critical places. They can be used to detect runtime instances of RCE attacks, but cannot expose vulnerabilities before real attacks are launched.

While static analysis has the potential of exposing vulnerabilities, most of them [20], [31], [27], [33], [34], [35], [17] cannot cohesively reason about the string and non-string parts of an application and many lack path sensitivity, whereas RCE attacks require satisfying intriguing path conditions, involving both strings and non-strings. Recently, researchers have proposed techniques that can model both strings and non-strings in dynamic symbolic execution of web apps [25], [12]. However, they focus on modeling only the executed path, whereas vulnerability detection requires modeling all program paths. A more thorough discussion of the limitations of existing techniques can be found in Section III.

In this paper, we propose a path- and context-sensitive interprocedural static analysis that detects RCE vulnerabilities in PHP scripts. It features the capabilities of reasoning both the string and non-string parts of an application in a cohesive and efficient manner, and reasoning across multiple scripts and requests. It is able to guide exploit generation (i.e. generating requests with concrete inputs) to confirm the reported vulnerabilities. It analyzes and encodes PHP scripts as two kinds of constraints: non-string and string constraints. A novel algorithm is developed to solve these constraints in an iterative and alternative fashion. Real exploits can be composed from the satisfying solutions.

The contributions of the paper are summarized as follows.

- We develop a static analysis to automatically detect RCE vulnerabilities in PHP code. The analysis is interprocedural, context- and path-sensitive, leveraging a string solver and a SMT solver.
- We propose to abstract a web app into two separate subprograms: one capturing the non-string semantics and

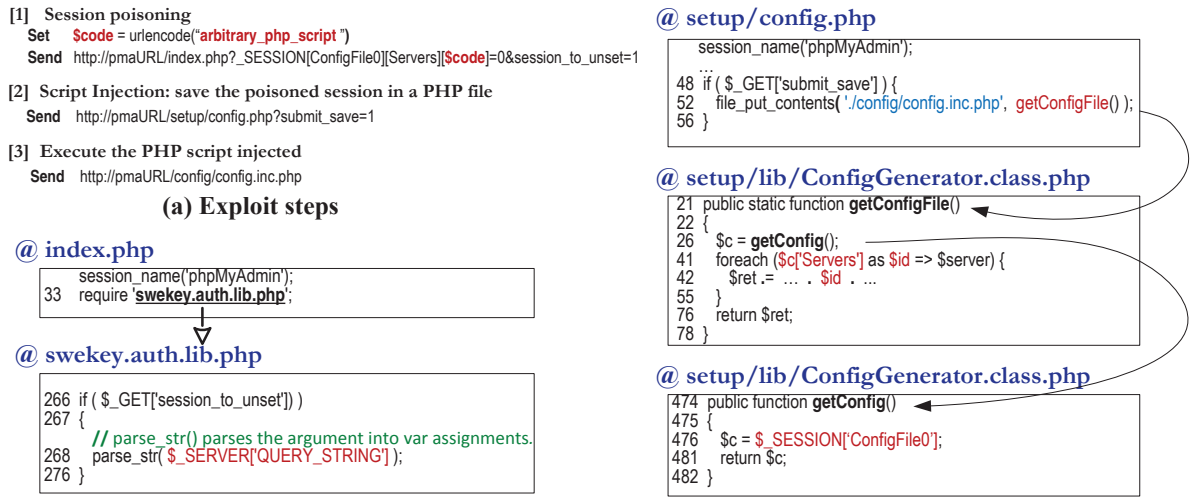


Fig. 1: RCE in phpMyAdmin v3.4.3 (simplified)

the other modeling the string related behavior. The two sub-programs are encoded separately. We also develop a novel algorithm that solves the two sets of constraints simultaneously.

- We address a number of practical challenges, including analyzing across scripts and requests to simulate stateful attacks, handling dynamic conditional script inclusion, and modeling session constraints.
- We have evaluated the technique on 10 real world web applications. We successfully identify 21 RCE vulnerabilities with 8 that have not been reported in the past. We have confirmed all these vulnerabilities by constructing real exploits based on the analysis results. The overhead of our technique is reasonable.

II. Motivating Example

We use two examples to motivate our approach.

A. RCE in phpMyAdmin

Recently, a RCE vulnerability was reported for *phpMyAdmin v3.4.3* [1], which is a MySQL database management tool using a web interface. The vulnerable versions have been downloaded over six million times according to SourceForge.

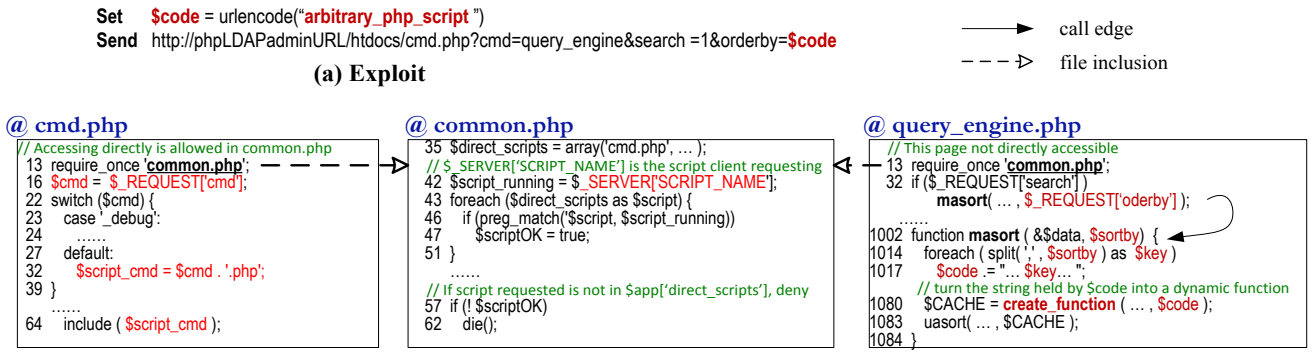
Fig.1 (a) describes an exploit to the vulnerability, which consists of three steps. The first two are **session poisoning** and **script injection**. In the first step, a crafted request is sent to `index.php` to change the configuration of the server. Instructed by the command session_to_unset, a key-value pair is stored to a special session array _SESSION[ConfigFile0][Servers] that is supposed to store the list of servers under administration. The key-value pair contains a piece of PHP script $code as the key, which gets stored to the current session. In the second step, another request is sent to `config.php` to save the current configuration, including the information stored in the session array _SESSION[ConfigFile0][Servers] by the previous step. Consequently, the provided code piece is stored to a PHP file, which gets executed by the request in the third step.

The relevant code snippets are shown in Fig.1 (b-c). Fig.1 (b) shows the relevant snippets in `index.php` and `swekey.auth.lib.php` that are executed in the session poisoning step. `index.php` first specifies the current session name. Then it executes the included script `swekey.auth.lib.php`. Depending on the value of the incoming parameter session_to_unset, method parse_str() is called at line 268, parsing its argument to variable assignments. For example, parse_str('a=1&b=2') has the same effect as that of executing '$a=1; $b=2;'. The variable $_SERVER['QUERY_STRING'] stores the query string from the client. Hence the invocation at line 268 defines $_SESSION['ConfigFile0']['Servers']['$code'] to 0. As PHP treats arrays as hash mappings indexed by keys, the key value $code is stored to the session array.

Fig.1 (c) shows the relevant snippets in `config.php` that are executed in the script injection step. At line 52, there is a write to a PHP file guarded by a predicate at line 48. In the file write, the string returned by getConfigFile() is written. At line 41 inside getConfigFile(), $c['Servers'] is aliased to $_SESSION['ConfigFile0']['Servers']. Then in the foreach loop, the previously stored key value $code is assigned to $id and defined as part of the return value $ret. In this way, the string ($code) composed by the client in the previous request is written to a PHP script.

A hidden complexity is that the session of the requests have to be identical. In handling client side requests, if not explicitly specified, a default session name ('PHPSESSION') is assigned. Therefore, in the session poisoning step, one cannot directly send the request to `swekey.auth.lib.php`, even though the request can be correctly parsed and the arguments can be stored into the (default) session. Because the default session is different from the session specified in the second step. We have to call `index.php` instead to ensure we are referring to the same named session 'phpMyAdmin' as in `config.php` in the second step.

[1]CVE-2011-2506: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2506



(b) Relevant code snippets

Fig. 2: RCE in phpLDAPadmin v1.2.1.1 (Simplified)

From the example, we make a number of observations. First, determining if a vulnerability is a true positive demands reasoning both string and non-string parts. Observe that the path conditions at line 48 in `config.php` and line 266 in `swekey.auth.lib.php` entail non-string reasoning while the file name and the file content at line 52 in `config.php` require string reasoning. Second, we have to reason across requests and scripts, and handle sessions properly. Neither `index.php` nor `config.php` is vulnerable by itself. Third, in order to successfully construct an exploit to confirm the vulnerability, we need to know the concrete inputs to satisfy the path conditions. For instance, the parameter session_to_unset in the step (1) request in Fig.1 (a) is to satisfy the condition at line 266. The parameter in the step (2) request is to satisfy the condition at line 48. In other words, the analysis ought to be path-sensitive.

B. RCE in phpLDAPadmin

Another type of RCE vulnerabilities is related to eval() that executes a string provided as its parameter. There is a vulnerability[2] in *phpLDAPadmin v1.2.1.1* rated "critical" by the developer. *PhpLDAPadmin* provides user-friendly web interfaces to manage a LDAP server. It is a popular tool and has been installed for more than *242* thousand times.

As shown in Fig. 2 (a), the vulnerability is exploited by a request to `cmd.php`, which is supposed to accept and execute a command from the client. In the exploit, an invalid command is provided such that a PHP script named by the command (i.e. `query_engine.php`) gets executed. The script is supposed to be internal and cannot be requested directly. It can accept the rest of the parameters in the exploit request and execute the malicious script provided by variable $code.

The relevant code snippets are shown in Fig. 2 (b). The script `cmd.php` first gets executed. It includes `common.php` for access control, which is a common design pattern for PHP programs. The access control is conducted by comparing the current script acquired from $_SERVER['SCRIPT_NAME'] at line 42, with a white-list specified in array $direct_scripts. Observe that at line 35 in `common.php`, `cmd.php` is listed and the execution is allowed to proceed. Lines 16-39 in `cmd.php` determine the command indicated by the client request. If it is not a pre-defined command, such as "`_debug`", the code constructs a PHP file name with the command argument (line 32) and tries to load and execute the file at line 64. In this case, the script `query_engine.php` is executed. Inside `query_engine.php`, `common.php` is again included and executed for access control. Since `query_engine.php` is included by `cmd.php`, it inherits all its privileges. Hence, despite the script itself is not in the while-list, the execution is allowed to continue. At line 32 in `query_engine.php`, if the client parameter search is set, the masort() function is called to execute a sorting function constructed dynamically by lines 1014-1080. The construction allows the string provided in the client parameter $orderby to be included as part of the constructed function and executed at line 1083. Note that variable $sortby is an alias to $orderby and uasort() at line 1083 is equivalent to eval().

Note that the root cause of the defect is not just the masort() function because executing a dynamically constructed function is the intended semantics of masort(). It is the combination of the mistake in access control, i.e. allowing the client to indirectly invoke `query_engine.php`, and the masort() function that constitutes the vulnerability. Hence, our technique ought to be able to analyze access control mechanism, which is essentially through predicates. For this example, the request is so crafted (i.e. the cmd parameter has to be 'query_engine', not even a legitimate command) to get through the access control. It is unlikely that an analysis incapable of modeling path conditions would identify the exploit.

The example also illustrates the necessity of handling dynamic file inclusion (line 64). We have to model the fact that the name of the file to be included is a variable in the program. Based on the value of the variable, different files need to be modeled and encoded as part of the constraints.

III. Problems in Existing Static Analysis

Existing static techniques fall short in RCE vulnerability detection and exploit generation.

String Analysis is Needed. Web apps are different from regular programs in that they heavily rely on string operations. Inputs from the client side are mostly strings. The outputs of web apps are mainly strings as well, such as SQL queries, html pages, and JavaScript code pieces. In some sense, the main functionality of a web app is often string processing. However,

[2]CVE-2011-4075: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-4075

PHP Script	Non-string Abstraction	String Abstraction

01 **if** ($_POST['a'] < 3) {
02 $role = "visitor";
03 $log = "v";
04 }
05 **else** {
06 **if** ($_POST['a'] < 5) {
07 $role = "user";
08 $log = $_POST['msg'];
09 }
10 **else {**
11 $role = "admin";
12 $log = "a";
13 }
14 }
15 **if** ($role == "admin")
16 writeFile("f.php", $log);
	101 b_1 = _POST_a < 3 ;
	102 **if** (b_1) {
	103 T_r=0;
	104 T_l=0;
	105 }
	106 **else** {
	107 b_2 = _POST_a < 5 ;
	108 **if** (b_2) {
	109 T_r=0;
	110 T_l=1;
	111 }
	112 **else {**
	113 T_r=0;
	114 T_l=0;
	115 }
	116 }
	117 **if** (b_3)
	118 **assert1** (T_l=1);
		201 **if** (b_1) {
		202 role:= "visitor";
		203 log:= "v";
		204 }
		205 **else** {
		206 **if** (b_2) {
		207 role:= "user";
		208 log := _POST_msg;
		209 }
		210 **else {**
		211 role := "admin";
		212 log := "a";
		213 }
		214 }
		215 b_3=compare(role, "admin");
		216 **if** (b_3)
		217 **assert2** ("f.php" in "*.php")

Fig. 3: Constraint Generation Example

```
cfg log1:="v";          cfg role1:="visitor";     //line 02-03
cfg log2:=post_msg;     cfg role2:="user";        //line 07-08
cfg log3:="a";          cfg role3:="admin";       //line 11-12
cfg log4:=log2|log3;    cfg role4:=role2|role 3;  //line 13
cfg log5:=log1|log4;    cfg role5:=role1|role 4;  //line 14

assert("f.php" in "*.php"); //Assertions not in HAMPI format
assert(log5==post_msg);     //Simplified for reading
```

Fig. 4: Path insensitive encoding

a lot of existing static analysis for regular programs [32], [14] do not focus on properly modeling strings.

String Analysis Alone is Not Sufficient. On the other hand, string analysis [20], [31] alone is not sufficient as most web apps still have substantial non-string operations, which are for execution path control, access control, and arithmetic computation. The string and non-string operations cohesively depend on each other. Most existing string analysis are based on context free grammar (CFG) and only model string operations, abstracting away the non-string part of a web app. This could lead to false positives. Moreover, since they do not reason about path conditions, users can hardly use them to construct exploits because that requires knowing the inputs satisfying various path conditions.

HAMPI. Hampi [20] is a string analysis engine developed in the past. A lot of recent web app analysis are built upon it [25]. It models a string assignment as a grammar rule definition. Assignments to the same variable in the two branches of a conditional statement become alternatives in the right hand side of a rule.

Take the script in the left column of Fig. 3 as an example. It determines the role of the client according to the value of $_POST['a'], defines the content in $log and finally writes to a PHP file if the role is "admin". To detect if the file write is vulnerable to RCE attacks, at line 16, we assert that (1) the file written is a PHP file and (2) the content written to the file is from $_POST['msg'] provided by the client. The first assertion is trivially true. However, path sensitive and path insensitive analysis yield different answers to the second one.

In standard string analysis such as HAMPI, the CFG generated for the program is shown in Fig. 4 and the second assertion is satisfiable with the CFG. Therefore, a vulnerability is reported. However, it is a false positive. Consider the predicate at line 15, the reachability condition of line 16 requires $role = "admin", which implies $log = "a". This can only be prevented by a path-sensitive analysis modeling both string and non-string behavior.

Solving String and Non-String Constraints Together is Difficult. While we need to reason about string and non-string behavior together, existing solutions are not adequate. One class of solutions models strings as bit-vectors which can be handled by existing SMT solvers to achieve solving string and non-string constraints together. However, a precondition in bit-vector logic is that the lengths of bit-vectors are fixed and decidable. However, this does not hold for strings. For example, str := post_msg | "123" is a CFG rule. The length of str varies depending on the alternatives. In one of the alternatives, it is determined by the client side input, which is uncertain. This causes difficulty in reasoning about constraints built on top of str. Most existing techniques in this category are developed in the context of dynamic symbolic execution [25], [12]. Since they only need to model the executed path, string lengths or strong hints of string lengths can be acquired from the execution. These techniques can hardly be applied in our context as we need to model all possible paths. Another plausible solution is to translate both string and non-string constraints to a third party constraint language [28] and solve them together. However, third party languages are often limited in expressiveness. For example in [28], the M2L constraint language used can only model a very small set of arithmetic operations, namely addition and subtraction, and it does not allow more than one variables in an expression.

Incapabilities in Modeling RCE Specific Characteristics. As we can observe from the two cases in the previous section, RCE attacks have specific characteristics that need to be properly handled. For example, multiple requests to more than one scripts need to be analyzed together; dynamic file inclusion needs to be modeled. As far as we know, none of the existing techniques can meet the challenges.

IV. Design

Given a web app, we first identify the operations sensitive to RCE attacks, called *sinks*. We consider two kinds of sinks: file writes and dynamic script evaluations (e.g. eval()). For a file write, if we can determine along some feasible path it writes to a .php file and the content contains values from the client, it is considered problematic. For a dynamic script evaluation, if the string evaluated as a script contains values from the client along some feasible path, it is considered problematic.

Our technique is an inter-procedural, context-sensitive and path-sensitive static analysis. It reasons about both string and non-string behavior. To begin with, the technique creates two abstractions of the given app: one for the string related behavior and the other for the non-string related behavior. The non-string abstraction includes additional taint semantics to reason about the input correlation for each variable. The two abstractions are encoded separately. Then we solve them together via a novel and sound, *iterative* and *alternative* algorithm. For a potentially vulnerable file write, we query the string constraints if the file name ends with the PHP extension

and query the non-string constraints to determine if the written content is tainted and the file write is reachable. The solution has to consistently satisfy all these queries.

A. Abstractions

For each PHP script, we create two abstractions: string and non-string abstractions. Intuitively, one can consider they are simplifications of the original program that handle only string type and non-string types, respectively. To facilitate abstraction, we have implemented a number of auxiliary analysis. One is type inference [22] as PHP programs are dynamically typed. It leverage the known interface of string operations. For example, if a statement is a string concatenation, the variable holding the return value as well as the arguments are of string type. Transitively, other correlated variables can be typed. The second is a standard context-sensitive alias analysis. The third one is a field name analysis that identifies all the field names or constant indices for an array. We assume a program is normalized such that the predicate in a conditional statement is a singleton boolean variable.

TABLE I: String Abstraction Rules. Operator ':=' represents definition and '.' concatenation. Variables $s1, $s2, $s3 are of string type, $a, $a1 and $a2 are of array type, and $b is of boolean type. F1,..., and fn are constant field names of $a, $a1, and $a2 identified through the field name analysis.

Statements	Abstraction
(1) $s1 = $s2;	s1 := s2
(2) $s1 = concat($s2, $s3)	s1 := s2.s3
(3) $s1['c'] = $s2;	s1_c :=s2
(4) $s1 = $s2['c'];	s1 :=s2_c
(5) $a1 = $a2;	a1_f1 :=a2_f1 ... a1_fn :=a2_fn
(6) $b= ($s1==$s2)	b = *compare*(s1, s2)
(7) **if** (b) {...} **else** {...}	**if** (b) {...} **else** {...}
(8) **foreach** ($a **as** $s1⇒$s2) /*loop body*/	s1:='f1'; s2:=a_f1; /* loop body */; ... s1:='fn'; s2:=a_fn;/*loop body*/;

String Abstraction. The abstraction retains the control flow structure, statements of string type, and string comparisons in the original program. Other statements are abstracted away. It thus models the string related behavior in the program.

The detailed abstraction rules are presented in Table I. String copies and concatenations (rules (1) and (2)) are straightforward. For an array access indexed by a constant field name, such as $a['c'], we create a new variable a_c to denote the field (rules (3) and (4)). Currently, we have limited support for dynamically constructed field names/indices: we use free variables to denote such accesses. For an array copy (rule (5)), we generate the definitions for the fields identified by the field name analysis. For a string comparison (rule (6)), we retain the same boolean result variable and make use of the *compare()* primitive that compares two strings. We will explain in later sections how this will be turned into string constraints and solved by HAMPI. Rule (7) means that we retain the structure of a conditional statement. For a regular loop, we unroll the loop body once and turn it into a conditional statement. This is a standard solution to handling loops in constraint based analysis [13]. A foreach loop involving strings, "**foreach** ($a **as** $s1⇒$s2)", means that $a is a mapping and the loop iterates over each entry of the mapping and instantiates $s1 with a key and $s2 the corresponding value. We unroll the loop n times for the n fields we have identified for the array (rule (8)). In each iteration, the field name is associated with s1 and the field value is associated with s2.

Note that the string abstraction is incomplete due to the dynamic nature of the language. For example, it is difficult to reason about string values in field accesses through dynamically constructed field names/indices. Fortunately, from our experience, most field accesses have constant names. For example in phpLDAPadmin, 3656 out of 4636 field accesses have constant names. In our experiments, this has not led to any false positives.

The third column in Fig. 3 shows the string abstraction for the program in the first column. It is essentially a simplified version of the original program with the integer comparisons excluded and some unbounded boolean variables used.

TABLE II: Non-String Abstraction Rules. Variables $x, $y, $z are of non-string type, $a is of array type, $s1, $s2 and $s3 are of string type, and $b is of boolean type. Operator ⋄ denotes a binary operation. T_x represents the boolean taint bit of $x. F1,..., and fn are constant indices identified through the field name analysis.

Statements	Abstraction
(8) $x = $y;	x = y; $T_x = T_y$
(9) $x = $y ⋄ $z;	x = y ⋄ z; $T_x = T_y \vert T_z$
(10) $x = $a['c'];	x = a_c; $T_x = T_a \vert T_{a\_c}$
(11) **if** (b) {...} **else** {...}	**if** (b) {...} **else** {...}
(12) **foreach** ($a **as** $x⇒$y) /*loop body*/	x=f1; y=a_f1; $T_x = T_a$; $T_y = T_a \vert T_{a\_f1}$;/*loop body*/; ... x=fn; y=a_fn; $T_x = T_a$; $T_y = T_a \vert T_{a\_fn}$;/*loop body*/;
(13) $s1 = $s2;	$T_{s1} = T_{s2}$
(14) $s1 = concat($s2, $s3)	$T_{s1} = T_{s2} \vert T_{s3}$
(15) **foreach** ($a **as** $s1⇒$s2) /*loop body*/	$T_x = T_a$; $T_y = T_a \vert T_{a\_f1}$; /*loop body*/; ... $T_x = T_a$; $T_y = T_a \vert T_{a\_fn}$; ...

Non-String Abstraction. The Non-String abstraction retains the same control flow structure as the original program and all the statements of a non-string type. It also introduces taint semantics into the abstracted program. The taint semantics is standard, that is, a resulting variable is tainted if any of the operands are tainted. This is to model the correlations to client inputs. Hence, variables and statements of string type are not completely abstracted away. Instead, a boolean variable is introduced to represent the taint of a string variable. String operations are abstracted to corresponding taint operations.

Table. II presents the rules. Rules (8) and (9) are standard. In rule (10), the result variable $x is tainted if either the array $a is tainted or the specific field is tainted. Intuitively, it means if the entire array comes from the client, an array field comes from the client. This allows us to model the taint propagation semantics of array accesses with dynamically constructed field names/indices. For instance, in the sample code in Fig. 1. Statically, we don't know what variables are written by line 268 without knowing the concrete request. However, it is straightforward to assume that the $_SESSION array is tainted. According to rule (10), $c is tainted at line 476, which leads to $id being tainted at line 42, disclosing that $id could come

from the client. Rule (12) abstracts **foreach** loops with non-string operands. Similar to that in string abstraction, it unrolls the loop n times for the n constant fields identified through the field name analysis. Observe that the taint propagation of $y is similar to rule (10). Rules (13-15) are for string operations, which are abstracted to taint operations.

There are some operations that have string operands but non-string results, such as getting the index of a string in another string, our implementation currently uses free variables to denote the results of such operations.

The second column in Fig. 3 shows the non-string abstraction of the sample program. Observe that the two abstractions use the same set of predicate variables, which allows us to reason about the two parts together.

B. Constraint Encoding and Solving

With the two abstractions, the next step is to encode them separately and then solve them together. We develop a novel algorithm to drive the solution process. The algorithm queries the STP solver and the HAMPI string solver iteratively and alternatively to derive a consistent path-sensitive solution for both sets of constraints. Intuitively, one can consider the algorithm first solves the non-string constraints and produces a solution for the path conditions, the solution is then used to derive the HAMPI encoding *for that path* from the string abstraction. Doing so, the imprecision caused by path-insensitivity can be avoided. If HAMPI fails to resolve the constraints for the path, it means that the solution is infeasible. The algorithm alternates to the STP solver to explore a new solution. In the algorithm, we leverage the observation that string constraints are strong in pruning search space. In many cases, *a not fully path-sensitive string encoding* may not have any solutions so that we can completely avoid exploring the individual paths in that sub-space.

The encoding of the non-string part is standard (i.e. first translates the program to its SSA form and models individual statements to bit-vector operations) and hence omitted.

Encoding and Solving String Constraints. Before encoding, the abstracted program is translated to its SSA form in which a unique variable name is assigned to each definition and ϕ operators are used at joint points to multiplex the different values along different branches. The encoding is driven by the assignment for the boolean variables in the abstraction. The process is detailed in Algorithm 1.

Function **genStrCSTR()** generates HAMPI constraints (i.e. a CFG) for a statement in the SSA form. It is a recursive function, driven by the abstract syntax tree (AST) of the statement. In particular, for an assignment statement (lines 1-2), a HAMPI CFG definition, denoted by keyword *cfg*, is inserted to the CFG. For a conditional statement (lines 3-10), depending on the value assignment of the predicate variable, one of the branch is encoded. If the variable value is not specified, both branches are encoded. Note that, if all predicate variables are specified, we essentially encode a full program path. If only some are specified, we say that we encode a *partial path*, denoting a set of full paths. In lines 11-17, ϕ operators are encoded, which may introduce alternatives in the resulting grammar (line 17). Lines 18-22 encode string comparisons. Because HAMPI doesn't support direct comparison of two strings, we use rules "assert v in s_1; assert v in s_2" to query the equivalence of s_1 and s_2 with v a free variable. Note that HAMPI will instantiate v when it finds a satisfying solution.

Algorithm 1 Generate and solve string constraints

Input: S : a statement in the string abstraction in SSA form
R : assignment to boolean variables, indexed by var.
Output: the CFG C_{str}.

genStrCSTR(S, R)

1: **if** $S \equiv$ "$s_1 := s_2$" **then**
2: $C_{str} \leftarrow C_{str} \circ \{cfg\ s_1 := s_2\}$
3: **if** $S \equiv$ "**if** (b) S_1 **else** S_2" **then**
4: **if** $R[b] \equiv$ *true* **then**
5: genStrCSTR(S_1,R)
6: **else if** $R[b] \equiv$ *false* **then**
7: genStrCSTR(S_2,R)
8: **else**
9: genStrCSTR(S_1,R)
10: genStrCSTR(S_2,R)
11: **if** $S \equiv$ "$s_1 := \phi(b, s_2, s_3)$" **then**
12: **if** $R[b] \equiv$ *true* **then**
13: $C_{str} \leftarrow C_{str} \circ \{cfg\ s_1 := s_2\}$
14: **else if** $R[b] \equiv$ *false* **then**
15: $C_{str} \leftarrow C_{str} \circ \{cfg\ s_1 := s_3\}$
16: **else**
17: $C_{str} \leftarrow C_{str} \circ \{cfg\ s_1 := s2 \mid s_3\}$
18: **if** $S \equiv$ "$b = compare(s_1, s_2)$" **then**
19: **if** $R[b]$ is *true* **then**
20: $C_{str} \leftarrow C_{str} \circ \{assert\ v\ in\ s_1;\ assert\ v\ in\ s_2;\}$
21: **else**
22: $C_{str} \leftarrow C_{str} \circ \{assert\ v\ in\ s_1;\ assert\ v\ not\ in\ s_2;\}$

Input: P : string abstraction
R : assignment to boolean variables.
Output: SAT or UNSAT.

solveStrCSTR(P, R)

//generate HAMPI CFG
23: **foreach** top level statement $S \in P$ **do**
24: genStrCSTR(S,R)
25: **return** *QueryHampi*(C_{str})

Function **solveStrCSTR()** determines if a (partial) path, denoted by the (partial) specification of path conditions R, is feasible from the perspective of the string abstraction P. It builds the CFG by calling **genStrCSTR()** on all the top level statements and then solves it by calling HAMPI.

Iterative Solving. The iterative driver algorithm is presented in Algorithm 2. Function **driver()** takes the non-string constraints N, (the conjunction of the non-string encoding, the reachability assertions, and the taint assertions), and the string abstraction P, then produces a satisfying solution S if there is one. A reachability assertion dictates a sink under consideration is reachable (e.g. asserting line 118 is reachable in Fig. 3). A taint assertion dictates the content of a file write or eval()

is tainted (e.g. the assertion at 118 in Fig. 3).

Lines 1-5 are the fast path to detect unsatisfying cases. Note that the invocation of the string solver at line 4 considers the path-insensitive encoding. Lines 6-7 are also the fast path, checking if the fully path-sensitive string encoding with the path specified by R is satisfiable. If so, we simply terminate with R. If neither fast path can be taken, the recursive method **iterSolver()** is called to derive a path-sensitive solution.

Algorithm 2 Iterative and Alternative Solving

Input: N : Non-string constraints.
P : String Abstraction.
Output: a satisfying solution or UNSAT.

driver(N, P)

1: $(Sat_n, R) \leftarrow$ *querySTP*(N)
2: **if** $Sat_n \equiv$ UNSAT **then**
3: exit UNSAT
4: **if** *solveStrCSTR*$(P, \phi) \equiv$ UNSAT **then**
5: exit UNSAT
6: **if** *solveStrCSTR*$(P, R) \equiv$ SAT **then**
7: exit R
8: *iterSolver*(N, P, R, ϕ)

Input: R : a known satisfying solution for N
S : the generated final solution
Output: function returns implies UNSAT

iterSolver(N, P, R, S)

9: **if** $R \equiv \phi$ **then**
10: **exit** S
11: $b \leftarrow$ *select*(R), $b_v \leftarrow R[b]$
12: **if** *solveStrCSTR*$(P, S \circ \{b = b_v\}) \equiv$ SAT **then**
13: *iterSolver*$(N \circ \{\text{ASSERT}(b \iff b_v)\}, P, R - \{b\}, S \circ \{b = b_v\})$
14: $(Sat_n, R') \leftarrow$ *querySTP*$(N \circ \{\text{ASSERT}(b \iff \neg b_v)\})$
15: **if** $Sat_n \equiv$ UNSAT **then**
16: **return**
17: **if** *solveStrCSTR*$(P, S \circ \{b = \neg b_v\}) \equiv$ UNSAT **then**
18: **return**
19: *iterSolver*$(N \circ \{\text{ASSERT}(b \iff \neg b_v)\}, P, R' - \{b\}, S \circ \{b = \neg b_v\})$

Method **iterSolver()** takes an existing solution R to the non-string part N as a reference to derive the final solution S. The algorithm tries to speculate a (true/false) solution for a selected predicate at one iteration. The speculation is guided by the provided solution R. In other words, it tries to follow the satisfying path for the non-string part as much as possible until the string constraints become unsatisfiable. Then it backtracks and tries a different speculation.

Lines 9-10 are the termination condition, it means that if we have successfully speculated all predicates, we acquire a solution. Line 11 selects a predicate from the provided solution R. Right now our selection is based on the dependence distance to the sink under consideration. Line 12 speculates its value based on R and queries the string solver. Note that S contains all the predicates that have been speculated thus far and it does not specify any predicate that has not been speculated. Essentially, it is equivalent to querying the string engine with *partial path-sensitivity*. This is to leverage the observation that in many cases even partial path-sensitive string constraints are difficult to satisfy, allowing us to prune search space. At line 13, we continue speculation by recursively calling **iterSolver()**. Note that N is updated with the speculation, the predicate is removed from R, and the speculated path (the final solution) S is lengthened with the speculation.

In lines 14-19, when mis-speculation occurs, the negation of the selected predicate is explored. If both branches of the selected predicate have been tried but a satisfying solution could not be found, the method returns, which is equivalent to backtracking to the previous iteration.

Our experience shows that the algorithm can quickly converge in both the SAT and UNSAT cases (Section VI).

step	b_1	b_2	b_3	STP	HAMPI
1	f	-	-		SAT
2	f	t	-		UNSAT
3	f	f	-	UNSAT	
4	t	-	-	UNSAT	

TABLE III: Solving the example in Fig. 3.

Example. Consider the example in Fig. 3. The STP solver first generate a solution $\{b_1 = f,\ b_2 = t,\ b_3 = t\}$ (line 1 in Algorithm 2). The path insensitive string encoding (i.e. the one in Fig. 4) has a satisfying solution too. But, the path sensitive HAMPI encoding (shown as follows) is not satisfiable, disclosing the path is not a correct solution.

```
var v : 0 .. 20;
cfg role1 := "user";          // line 207
cfg log1 := post_msg ;        // line 208
assert ("f.php" in "*.php);
assert (v in role1);
assert (v in "admin");
```

Hence, the algorithm resorts to the iterative solver. Table III shows the process. At the beginning, it tries to follow the SAT solution by STP. At step 1, the string constraints with only $b_1 = f$ specified are SAT. So that the algorithm tries to further speculate $b_2 = t$, but this time the string constraints are UNSAT. It then alternates to the STP solver, exploring $b_2 = f$, which turns out to be UNSAT. It backtracks and explores $b_1 = t$ with the STP solver, which is UNSAT too. It then terminates with UNSAT.

Our technique analyzes sinks one by one. To reduce complexity, for each vulnerable candidate (sink), we use a PHP slicer, which was implemented in our prior work [35], to prune the irrelevant parts before abstraction and encoding. Since our technique does not handle some string operations such as indexOf(), it is unsound. However in practice, the number of false positives is low (Section VI).

V. Handling Practical Challenges

In this section, we discuss how to overcome a number of practical challenges for RCE vulnerability detection.

Handling Dynamic Inclusion. At runtime, through dynamic file inclusion, several PHP scripts may be combined together as the running script. We need to model such effects. For example, in Fig. 5, at lines $1-2$ in index.php, based on the value of $_REQUEST['role'], different script files may be

```
@ admin.php                                  @ user.php
11 function accessControl() {                21 function editData() { ... }
12   if ($_SESSION [ 'user' ] ! = 'admin')
13     header("Location: login.php");
14 }
15 function editData() { ... }
16 accessControl();
@ index.php
 1 if ( $_REQUEST['role'] == 'admin' )    include ( 'admin.php' );
 2 else  require ( 'user.php' );
 3 editData();
```

Fig. 5: Example to illustrate dynamic inclusion and access control.

```
      /* b1=(_REQUEST['role']=='admin') */
1     if (b1) {
2       inc=1;
3       /*inlining admin.php*/
4       /* b2=(_SESSION['user']!='admin') */      lines
5       if (b2) {                                 1 and 2
6         exit;
7       }
8     } else
9       inc=2;
10    b3= (inc==1);
11    if (b3)
12      /*inlining editdata() in admin.php*/      line 3
13    b4= (inc==2);
14    if (b4)
15      /*inlining editdata() in admin.php*/
```

Fig. 6: The non-string abstraction of the program in Fig. 5.

included at runtime. As a result, the function invoked at line 3 may refer to different code bodies.

The solution is to have conditional inlining at the call site that has multiple method bodies (e.g. line 3 in Fig. 5). However, we need to model the fact that the condition of inlining is not the reachability condition of the call, but rather the reachability condition of the inclusion site. We handle it by introducing a dummy variable at the inclusion site to denote the choice and later using the variable to guard the invocation.

Example. Fig. 6 shows the non-string abstraction of the program in Fig. 5. Lines 1-9 abstract lines 1-2 in the original code and lines 10-15 abstract the original line 3. Note that the definitions of b_1 and b_2 are not in the non-string abstraction, but rather the string abstraction. The page redirection at line 13 in Fig. 5 is abstracted as exit because when the page is redirected, none of the following statements gets executed. We introduce a dummy variable inc to denote the inclusion option and use it to guide the inlining in lines 10-15.

Reasoning Across Requests. RCE attacks may require coordination of multiple requests. Since in PHP requests are processed independently, cross-request attacks usually leverage sessions to preserve data. In PHP, a session can be determined by two parts: the *session name* and the *session id*. When a server and a client communicate, the session name and id are set in the HTTP header. The session name can be explicitly defined in a PHP script, otherwise the default value (PHPSESSID) is used. Also, a unique session id is assigned to each new visiting client. In order to reason about dependences across requests, we need to ensure the relevant requests are referring to the same session. Note that as long as the attacker ensures exploit requests are sent within the session expiration window, the session ids of these requests are automatically identical. Therefore, we only need to check if the session

statement	abstraction	
	string	non-string
[f1] initially	_SN_NAME_f1:=PHPSESSID	
[f1]session_name('C')	_SN_NAME_f1:='C'	
[f1]parsestr(...)	b=(s==_SN_NAME_f1)	T_{sn}=b?1:0
[f2]$x=_SESSION	s:=_SN_NAME_f2	$T_x = T_{sn}$

TABLE IV: Session abstraction.

names of these requests can be made identical.

Our solution is that for a statement in which a session value may be set, such as line 268 in the phpMyAdmin example in Fig. 1, we abstract the statement to a taint bit set operation guarded by a condition that the session name from the other request must be identical to the current session name. Therefore, any read from session in the other request is tainted only when it has the same session name. Table IV explains the process. The 1st column lists the relevant statements with f1/f2 the script. The 2nd and 3rd columns show the string and non-string abstractions, respectively. Initially, each script has a default session name (1st row). Any invocation to session_name() is abstracted as setting the current session name (2nd row). A statement or library call that allows the client to set session values is abstracted as a guarded taint bit set (3rd row). A session read in the other request f2 sets the global variable s to its session and copies the session taint bit.

VI. Evaluation

Our system makes use of LLVM, the PHP compiler (*phc*) [4], the STP solver and the HAMPI string solver[20]. *Phc* is used to translate PHP to C, allowing us to leverage the existing analysis in LLVM (e.g. alias analysis). The main analysis is implemented in LLVM. It takes the C program and transforms it to constraints. The solving algorithm is implemented in C.

We apply our technique on a set of real world applications as listed in Table V. Observe that some of them are large, with a few hundred files and over 200k LOC. These web apps are selected as we were able find some RCE reports about them on the Internet. One of our goals is to see if we can identify these reported vulnerabilities. All experiments are run on an Intel Dual Core i5 2.5GHz machine with 8GB memory. The experimental results are publicly available at [1].

TABLE V: Program characteristics.

application	PHP files	PHP LOC			
		avg	stdev	max	total
aidiCMS v3.55	273	157	280	2976	42843
phpMyFAQ v2.7.0	347	690	2956	30222	239380
zingiri webshop v2.2.2	457	139	304	4517	63768
phpMyAdmin v3.4.3	527	432	1498	26136	227716
phpLDAPadmin v1.2.1.1	97	293	522	3108	28456
phpScheduleIt v1.2.10	171	383	394	2157	65493
FreeWebshop v2.2.9 R2	190	198	412	4971	37636
ignition v1.3	30	118	375	2092	3542
monalbum v0.8.7	41	105	104	452	4288
webportal v0.7.4	514	59	89	1022	30266

Table VI presents the detection results. *Constraint* presents the average number of *variables* and *constraints* in the formula. *Sink* is the number of the places that are potential vulnerable. They are the number of file writes and dynamic script evaluations. They are analyzed one by one. *Report* is the number of vulnerable sinks that our technique reports. Among those reported, *FP* presents the number of false

TABLE VI: Analysis Result.

application	constraint(avg)		avg solve	avg						non-string		string	
	variable	constraint	iteration	time(s)	sink	report	FP	known	new	report	FP	report	FP
aidiCMS v3.55	95.2	96.6	0.0	7.5	55	5	2	1	2	5	2	11	8
phpMyFAQ v2.7.0	58.6	59.0	0.8	9.4	25	5	2	1	2	6	3	7	4
zingiri webshop v2.2.2	159.5	159.5	6.5	22.8	68	2	1	1	0	2	1	3	2
phpMyAdmin v3.4.3	167.0	160.0	0.0	1.6	65	1	0	1	0	1	0	1	0
phpLDAPadmin v1.2.1.1	491.0	493.0	38.0	87.6	6	1	0	1	0	2	1	1	0
phpScheduleIt v1.2.10	135.5	178.0	3.0	3.0	52	4	0	4	0	25	21	4	0
FreeWebshop v2.2.9 R2	185.8	198.0	15.3	30.8	38	4	1	1	2	5	2	12	9
ignition v1.3	62.0	69.7	0.0	1.6	8	3	0	1	2	5	2	3	0
monalbum v0.8.7	174.0	200.0	0.0	11.8	2	1	0	1	0	1	0	1	0
webportal v0.7.4	13.0	11.0	0.0	0.3	39	1	0	1	0	1	0	2	1
				TOTAL	358	27	6	13	8	53	32	45	24

TABLE VII: Constraint solving comparison.

application	algo. 2		non-guided solving	
	iteration	time(s)	iteration	time(s)
aidiCMS v3.55	0	3.33	0	3.37
phpMyFAQ v2.7.0	5	4.63	69	30.49
zingiri webshop v2.2.2	13	45.56	53	98.27
phpMyAdmin v3.4.3	0	1.09	0	1.01
phpLDAPadmin v1.2.1.1	39	84.78	187	102.53
phpScheduleIt v1.2.10	9	6.30	96	50.00
FreeWebshop v2.2.9 R2	62	86.03	1402*	1843.83*
ignition v1.3	1	0.75	4	1.90
monalbum v0.8.7	0	0.01	0	0.01
webportal v0.7.4	0	0.30	0	0.29

* One UNSAT case times out. It cannot be solved in 30 minutes.

positives, *Known* is how many of them have been reported. *New* presents those that have not been reported in the past. *Iteration* reports the average number of iterations needed to determine SAT/UNSAT (precluding the cases that are UNSAT for either the string or the non-string constraints alone). *Time* is the average analysis time including the abstraction and constraint solving time.

For comparison, in the last four columns we also present the results generated by considering only the string part (which is equivalent to using HAMPI alone for detection [20]) and only the non-string part (which is equivalent to a path-sensitive static taint analysis [29]).

For the new true positive reports, we have constructed exploits from the SAT solutions (for path conditions) to confirm them. The exploits are also available at [1]. They are concrete HTTP requests that can execute an arbitrary piece of payload script.

We have the following observations from the results.
(1) Our technique is effective. Its false positive rate is only 22%. It identifies 21 real RCE vulnerabilities, including all the reported ones (13) and 8 new ones.
(2) Its overhead is reasonable. Observe that the average number of iterations is small. Sometimes, it is zero if the solutions can be found in the fast path. It indicates our algorithm can quickly converge for both SAT and UNSAT cases. Another performance study can be found later.
(3) Considering either the string or the non-string part alone produces a lot of FPs, indicating the need of reasoning both parts cohesively.

False Positive. Our technique produces FPs. Currently we cannot model some environment related library functions such as file_exists() because they require dynamic information. We have limited support for string functions such as substr() and getExtension() due to the limit of HAMPI. We introduce free

```
@ modul/tinymce/plugins/ajaxfilemanager/ajax_save_text.php

11  $path = $not_important . $_POST['name'];
21  if ( getFileExt($_POST['name']) == "php" )
    { ... }
28  else
    {
33    if( file_exists($path) )  { ...}
36    else {
38      $fp = fopen($path , "w+");
40      fwrite($fp, $_POST['text']);
        ...
      }
    }
```

Fig. 7: FP in adidCMS

variables in the constraints to denote the outcome of these functions. This leads to false positives sometimes.

A false positive in adidCMS is presented in Fig.7. This piece code saves the content ($_POST['text']) posted by the client into a file whose name ($_POST['name']) is also provided by the client. If we don't consider the path conditions, the file write at line 38 can write the content from the client into a PHP file named by the client. However, if the file name ends with .php, the predicate at line 21 will capture the dangerous behavior. However, we currently cannot model getFileExt() and file_exists() , leading to the false positive.

Evaluating the Constraint Solving Algorithm. We perform another experiment to evaluate our solving algorithm. Recall we use the STP solution to guide the overall process and leverage the string solver to prune search space (Algorithm 2). We compare the algorithm with a simple algorithm that also solves string and non-string constraints. The simple algorithm acquires a SAT solution from STP and then validates it using HAMPI. If UNSAT, it acquires a different SAT solution from STP, until a solution is found or STP reports UNSAT. The results are shown in Table VII. The iteration number and time in the table are the sum of those for analyzing individual sinks for each benchmark. The runtime is the solving time, not including the abstraction time. Observe that our algorithm is in general much better. The simple algorithm may run into deep troubles for UNSAT cases (e.g. `FreeWebShop`) when the search space is large.

VII. Related Works

The work in [28] also statically models the string and non-string behavior of a program. It translates both to a common M2L constraint language. However, M2L has limited expressiveness (Section III). Researchers have also modeled both strings and non-strings in the context of dynamic symbolic execution [25], [12]. However, they require knowing string

lengths beforehand and they only model the executed path whereas we need to encode all possible paths.

String operations can also be modeled as finite state transducers(FST). The work in [30] introduces symbolic representations in automata to handle the infinite alphabets problem in the classical FST. In [24], string constraint solving is used to repair HTML generation errors. ViewPoints [5] applies static string analysis to compute inconsistency between client-side and server-server input validation functions. However, they all have very limited support for non-string behavior.

RCE attacks are a special kind of Cross Site Scripting (XSS) attack and thus our work is related to detecting SQL injection [20], [31], [18], XSS [9], [16], [21], [23], [26] and HTTP request parameter tampering [8], [11] attacks. Among these works, the dynamic analysis based approaches [11], [9], [16], [21], [23], [26], [8], [10], [7], [6] require running the program. The effectiveness of these techniques is dependent on the concrete executions monitored.

Static techniques [20], [31], [17], [35] consider all possible executions. In particular, [20], [31], [33], [19] abstract away non-string computation and reason about string manipulations in a path insensitive way. In [17], researchers used a static constraint solving based technique to precisely identify the interface of a web app. In [35], a path-, context- and field-sensitive static analysis was proposed to detect resource contention problems. However, both works have limited support for string reasoning and only consider constant strings.

Similar to our work, researchers in [15] also observed that modeling session is important in analyzing web applications. However, the work aimed at test suite generation.

VIII. Conclusion

We propose a path- and context-sensitive analysis to detect Remote Code Execution (RCE) attacks in web apps. The analysis reasons about the string and non-string behavior of a program cohesively. It first creates two abstractions of the program to model the string and non-string behavior, respectively, which are encoded to constraints separately. A novel algorithm is developed to resolve the two sets of constraints together. The technique handles a lot of RCE specific challenges by extending the abstractions. Our experiment shows that the technique is very effective in detecting RCE vulnerabilities in real-world PHP applications, producing much fewer false positives compared to alternative techniques. And the underlying constraint solving algorithm is very efficient.

References

[1] http://www.cs.purdue.edu/homes/zheng16/rce/index.html.

[2] OWASP PHP Top 5. https://www.owasp.org/index.php/PHP_Top_5.

[3] IBM threat reports. http://www.ibm.com/services/us/iss/xforce/trendreports/.

[4] Phc: open source PHP compiler. http://www.phpcompiler.org/.

[5] M. Alkhalaf, T. Bultan, S. Roy Choudhary, M. Fazzini, A. Orso and C. Kruegel. ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies In *ISSTA'12*.

[6] G. Antoniol, M. D. Penta and M. Zazzara. Understanding Web Applications through Dynamic Analysis. *IWPC'04*

[7] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar and M. Ernst. Finding bugs in dynamic web applications. *ISSTA'08*

[8] M. Balduzzi, C. T. Gimenez, D. Balzarotti and E. Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *NDSS'11*.

[9] D. Bates, A. Barth and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW'10*.

[10] C. Bezemer, A. Mesbah and A. Deursen Automated security testing of web widget interactions. In *FSE'09*.

[11] P. Bisht, T. Hinrichs, N. Skrupsky and V. N. Venkatakrishnan. WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In *CCS'11*.

[12] N. Bjørner, N. Tillmann and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In *TACAS '09*.

[13] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS'04*.

[14] M. Das, S. Lerner, M. Seigel. ESP: path-sensitive program verification in polynomial time. In *PLDI'02*.

[15] S. G. Elbaum, S. Karre and G. Rothermel. Improving Web Application Testing with User Session Data. In *ICSE'03*.

[16] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *NDSS'09*.

[17] W. Halfond, S. Anand and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *ISSTA'09*.

[18] W. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *ICSE'06*.

[19] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *ASE'10*.

[20] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer and M. D. Ernst. HAMPI: a solver for string constraints. In *ISSTA'09*.

[21] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *SP'09*.

[22] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[23] Y. Nadji, P. Saxena and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS'09*.

[24] H. Samimi, M. Schafer, S. Artzi, T. Millstein, F. Tip and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE 2012*.

[25] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song. A Symbolic Execution Framework for JavaScript. In *SP'10*.

[26] P. Saxena, D. Molnar, B. Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS'11*.

[27] F. Sun, L. Xu and Z. Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *USENIX Security 2011*.

[28] T. Tateishi, M. Pistoia and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *ISSTA'11*

[29] O. Tripp, M. Pistoia, S. Fink, M. Sridharan and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI'09*

[30] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar and N. Bjørner. Symbolic finite state transducers: algorithms and applications. In *POPL'12*.

[31] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI'07*.

[32] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. In *ACM Trans. Program. Lang. Syst. May, 2007*.

[33] F. Yu, M. Alkhalaf and T. Bultan. Patching Vulnerabilities with Sanitization Synthesis. In *ICSE'11*.

[34] F. Yu, T. Bultan and B. Hardekopf. String Abstractions for String Verification. In *SPIN'11*.

[35] Y. Zheng and X. Zhang. Static Detection of Resource Contention Problems in Server-Side Scripts. In *ICSE'12*.