On Security Analysis of PHP Web Applications

David Hauzar and Jan Kofroň
Faculty of Mathematics and Physics, Charles University in Prague
{hauzar, kofron}@d3s.mff.cuni.cz

***Abstract*—In recent years, focus of business world has been moved towards the Internet. Web applications provide a generous interface non-stop thus offering to malicious users a wide spectrum of possible attacks. Consequently, the security of web applications has become a crucial issue.**

The state-of-the-art tools for bug discovery in languages used for web-application development, such as PHP, suffer from a relatively high false-positive rate and low coverage of real errors; this is caused mainly by unprecise modeling of dynamic features of such languages and path-insensivity of the tools. In this paper, we will demonstrate weak points of the tools and describe our novel approach to these issues. We will show how our technique handles some of the situations where other tools fail and illustrate it on examples.

I. Introduction

Recently, as business world has moved its focus towards the Internet, a number of applications have been moved online, and this trend is still continuing. According to the CENSUS [17], the online retail sales in the US in 2010 reached over 160 billion US dollars. Safety and security of the web applications involved in such transactions is therefore of the top priority.

A typical web application is available and operational 24/7, thus not putting any time pressure on malicious users; a generous interface these applications provide further widens the hacker's field. Amongst the 25 most common programming errors, those specific to web applications form a significant part of this group [5]; the examples include improper neutralization of SQL commands, cross-site request forgery, and missing authorization.

The most common programming language used at the server side is PHP [13]. PHP features many special attributes that make it different from common programming languages, especially as far as dynamism is concerned. The examples are inclusion of a file specified by a runtime-computed filename and the *eval* construct allowing runtime construction of code that is executed afterwards. This makes it hard or sometimes even impossible to apply the same techniques and tools for finding bugs or for correctness verification as in the case of "non-web" programming languages.

A. Problem statement and goals

A lot of attention has been paid to the development of methods and tools that would help debugging these applications and establishing their correctness in some sense, since the methods for "non-web" languages cannot be easily applied. The current state-of-the-art tools, however, still suffer from low error coverage, a relatively high false-positive rate, and often also from a weak support of language constructs, such as classes, dynamic includes, and the *eval* statement [9], [18].

In this paper, we propose a method for the identification of bugs inside web applications caused by data flow of unsanitized inputs from the user to sinks (SQL queries, URL constructions, output in general, etc.) inside web applications written in PHP. We describe our method and demonstrate benefits of our approach over those present in related tools and illustrate our method on an example.

This work was partially supported by Charles University Foundation grant 431011 and by institutional money from PRVOUK.

II. Errors inside web applications

A huge number of security holes inside web applications can be grouped under one category which allows data to propagate from a user input (*sources*, e.g., form fields on a web page) into database queries, URLs, JavaScript code, etc. (*sinks*) without checking if they are malicious [14]. These can be prevented by filtering user input, escaping the output, and by keeping track of the input data [14].

Filtering input is a process of preventing invalid data from getting into sinks. Blacklist filtering excludes malicious data, while whitelist filtering excludes all data except for that explicitly listed; thus it is distinctively safer than blacklist filtering due to the possibility of a missing item in the list. Escaping and encoding special characters that the application outputs prevents injection of malicious code and data.

Taint analysis is a technique which makes it possible to discover the data paths from *sources* to *sinks* [10], [21], [11], [3]. It marks data coming from sources as tainted and then propagates the taint markings. Data is tainted if it can be influenced by a user and, at the same time, it is not sanitized. Note that some sources represent a larger security threat than others and it is necessary not only to "taint" data but also to distinguish between different taint sources.

III. State of the art

Huang et al. [10] developed a static analysis for PHP applications in WebSSARI tool. Xie [21] discusses the limitations of their approach, in particular that it is intraprocedural and it does not model dynamic features such as dynamic arrays, objects, dynamic variables, and dynamic includes. To identify vulnerabilities, the approach performs taint analysis. Their approach does not allow for a custom sanitization, because data are only considered to be sanitized if they are processed with a specified sanitization function.

The approach of Xie et al. [21] uses inter-procedural analysis to find SQL injection vulnerabilities in PHP applications. They model automatic conversion of particular scalar types, uninitialized variables, simple tables, and include statements. However, they leave important parts of PHP unmodeled. In particular, they do not model references, object oriented features of PHP, and they ignore recursive function calls. To model sanitization process, the approach performs taint analysis. Sanitization can occur via calls to specified sanitization functions, casting to safe types, and a regular expression match. This means that the approach maintains a database of sanitizing regular expressions.

Wasserman et al. [19], [20] use grammar-based string analysis following Minamide [12] to find a set of possible string values of a given variable at a given program point and gain this information to detect SQL injections. However, the employed analysis has an incomplete support for references and does not track type conversions.

Pixy [11] performs taint analysis of PHP programs and it provides information about the flow of tainted data using dependence graphs [3]. It uses literal analysis to resolve include statements and performs alias analysis. However, it does not model aliases between variables and members of arrays. Next, Pixy lacks type inference, does not model PHP's variable-variables construct as well as variable-indices and provides only a very limited support of object oriented features. Moreover, similarly to WebSSARI it performs only simple taint analysis and does not allow for custom sanitization routines.

Balzarotti et al. [3] extended Pixy to perform the analysis of the sanitization process and thus are able to deal with a custom sanitization. They combine static and dynamic analysis techniques to verify PHP programs. They perform string analysis through language-based replacement and represent values of variables at concrete program points using finite state automata. They also track what parts of strings are tainted. Static analysis that they employ is based on Pixy and has the same limitations. Moreover, the database may not contain strings corresponding to attacks that were not considered in advance. Consequently, it can both miss vulnerabilities and cause false alarms.

Yu et al. [22] developed an automata-based approach for verification of string operations in PHP programs and incorporate the widening operator to tackle the problem of handling variables updated in loops. Similarly to [3], they extended Pixy to perform the analysis of the sanitization process; however, they do not employ the dynamic phase.

Biggar et al. [4] perform context sensitive, flow sensitive, interprocedural static analysis of PHP in order to gain information usable for code optimizatons in their PHP compiler. They combine alias analysis, type inference and literal analysis, model arrays, PHP's variable-variables construct, objects, references, scalar operations, casts, and weak type conversions. However, their analysis is closely tailored with their intent—to gain information usable for code optimizations. They gather information that must hold; information that may hold is tracked only in a very limited way. In most cases, they approximate information that may hold as unknown. This is not appropriate when the intent is to explore all possible behaviors of the code.

```
1  $users[1] = 'fred'; $users[2] = $_GET['inp'];
2
3  $t_users = & $users;
4  echo $t_users[1]; // Pixy reports the XSS vulnerability
5
6  $name = 'bob'; $index = 1;
7  if ($tainted) { $name = &$_GET['n']; $index = 2; }
8  echo $tainted ? htmlspecialchars($name) : $name; // Pixy reports the XSS vulnerability
9  echo $tainted ? htmlspecialchars($users[$index]) : $users[$index]; // Pixy reports the XSS vuln.
10
11 $ext = ".php"; $filename = 'inc' . $ext;
12 while (strpos($filename, '..')) $filename = preg_replace('..', '.', $filename);
13 include($filename);
14 echo $users[2]; // Pixy reports the XSS vulnerability
15
16 printFirstIndex('tainted', $users[1], $users[2]); // Pixy misses the XSS vulnerability
17 function printFirstIndex($varName, $untainted, $tainted) { echo $$varName; }
18
19 $user_ids = 2;
20 $user_ids[2] = $_GET['inp']; // because $user_ids is scalar, this line does nothing
21 echo $user_ids[2]; // Pixy reports the XSS vulnerability
```

Fig. 1. Dynamic features of PHP causing false alarms and missed vulnerabilities in Pixy tool.

The approach of Artzi et al. [1] generates test inputs automatically, monitors web applications for crashes, and validates that the output conforms to the HTML specification. The approach utilizes symbolic execution to capture logical constraints on inputs, based on these constraints, it creates new inputs that would increase the code coverage. By running an application on concrete inputs and using PHP runtime, they avoid the problem of modeling dynamic statements of PHP.

To our knowledge, a path-sensitive approach to a static analysis for PHP has not been yet published. However, there has been a lot of research done in the context of other languages. Examples of approaches to path-sensitive static analysis include [6], [7], [15], [8], [2], [16].

A. Demonstrating existing tool on examples

In this section, we show the limits and weak points of the Pixy tool [11] on a few PHP code fragments. We decided to demonstrate just the Pixy tool, since its analysis engine represents, to our knowledge, the best analysis engine available for finding vulnerabilities in a PHP code. At the end of Sect. IV, we demonstrate how these situations are handled when following the approach proposed in this paper.

Consider the example in Fig. 1. Due to the fact that Pixy does not model array aliasing correctly, a possible XSS attack is reported at line 4. Another issue related to arrays is caused by not handling variable indices. A different source of false positives is path-insensitivity; the `$name` variable is sanitized by the routine `htmlspecialchars` in all cases whenever it is tainted. However, Pixy reports a possible XSS attack at line 8. The last false-positive-alarm illustration starts at line 11. There, a file named "*inc.php*", which contains a sanitization of variable `$users[2]`, is included (note that the body of the while cycle is not executed at all, since the string `$filename` does not contain the "`..`" substring). Since Pixy omits modeling of the strings, it is not able to evaluate the `$filename` value, ignores the `include` statement, and reports an error at line 14 regardless of the

content of the included file; note that this can cause also missing a vulnerability.

The first case of missing vulnerabilites is caused by incorrect handling of the *variable-variable* construct represented by the `$$varName` at line 17. Another source of false negatives stems from insufficient modeling of the type system. The fragment starting at line 19 demonstrates this issue. The last limitation of Pixy we mention here is that Pixy does not model attributes of objects. So, according to the use of the objects, both false-negatives and positives can occur.

IV. Our approach

Our aim is to provide the developer with sufficient information so that she/he can assure a correct sanitization. In our case, this means employing analysis that computes data flow information using *dependence graphs* [3], identifies sources of sensitive data, sinks, and at each program point maintains:

- the taint and the sanitization status for each variable,
- the set of possible values of each variable,
- the set of conditions defined on the program's variables that must hold, and
- the set of possible types of each variable.

In the following, we describe how we gain this information and how we use it to detect vulnerabilities.

A. Outline

The main challenge of the analysis is the combination of an arbitrary user input and the dynamism of PHP. To address this problem, our analysis consists of the following steps:

1) Construction of the control-flow graph (CFG).
2) Static analysis of constructed CFG.
3) Detection of vulnerabilities.
4) A path-sensitive validation of vulnerabilities.

We based our approach on a combination of existing work, and extended it to face the aforementioned issues. We approach the problem of construction of CFG in presence of dynamic statements in a similar way as Pixy [11] does. First, only such dynamic statements that are directly given by literals are considered. Then, we gain information about possible values of variables, types, and aliases using static analysis of the CFG. We use this information to resolve dynamic statements and construct more precise CFG that is analyzed again. We repeat this process as long as new dynamic statements are being resolved or an iteration limit is reached. In addition to what the Pixy tool does, our analysis gains type information and we are able to resolve also polymorphic method calls using this approach.

We extended modeling of PHP data structures introduced in Biggar [4] by precise modeling of aliasing and adding certainty information to each edge of the points-to graph. *Certainty* tracks the fact whether given information at a given program point holds for each execution path from an entry point of the application to this program point; the way it is maintained is described below.

Our static analysis stems from the one introduced in Biggar [4]. We adapted it to track the certainty and taint information and augmented the literal analysis to propagate symbolic values through built-in operations.

We infer the sets of conditions that hold at each program point using conditional statements. At the start of a *then* branch corresponding to a given conditional statement, the condition corresponding to this statement is added; similarly, negation of the condition is added to the start of the *else* branch. At the join point, the condition is removed.

In the last step, we use certainty information gained during the analysis to identify vulnerabilities that may be false-positives because of path-insensitive analysis and validate them path-sensitively.

B. Modeling of PHP data structures

To model variables, array cells, and object fields, we use a points-to graph similar to the one introduced in [4]. The points-to graph contains three types of nodes. A *storage node* represents a symbol-table, an array, or an object. An *index node* represents a variable, an array cell, or an object field. Each index node is a child of a single storage node. Finally, a *value node* represents a scalar value. Index nodes are connected with storage and value nodes that constitute their values using *value edges*.

Aliasing is modeled using *reference edges* between storage and index nodes. Each reference edge has a *certainty tag* and a *direction tag* (left, right, both, or none). Each storage node contains an *unknown* field representing the values of the index nodes that have statically unknown indices, e.g., `$a[$dyn]` and `$$dyn` if the value of the variable `$dyn` is unknown. Finally, each index node can be *strong* or *weak*. An example of a points-to graph is described in Sect. IV-F.

Initially, all nodes are strong. A node becomes weak at a join point when one of its reference edges becomes uncertain (it is not present in all joined branches) and is directed from the node (see the node `name` in Fig. 2; it became weak at join point at line 7, Fig. 1). Further, a node may become weak after an assignment—the rules are described below. The set of values of a weak index node includes all storage and value nodes reachable from the index node using all *weak paths*, and the values in the unknown field of the parent storage node. A *weak path* is formed by reference and value edges where just the direction of the last reference edge must correspond to the direction of the path. The set of values of a strong index node is defined in the same way except that the uncertain reference edges are not taken into account. See Fig. 2 for an example—the set of values of the node `name` is `bob` and `u1`, the set of values of the node `n` is `u1`. Two index nodes are *aliases* if they are reachable from each other using the reference edges; they are *certain* aliases if they are reachable from each other using *certain* reference edges only.

Assigning a source index node `$s` to a target index node `$t` (i.e., `$t=$s`) replaces all value edges of the target node and its *certain* aliases with edges to all the values of the source node. In case the value is the storage node corresponding to an array, the storage node is copied and then a value edge to the copy is created. Next, the direction of all reference

edges from the target node and its *certain* aliases is changed to the direction towards the target node or the *certain* reference. Finally, the target node and all its *certain* aliases are marked as strong and all uncertain aliases as weak. See Fig. 2 for an example—assignment `$name=$_GET['n']` would change the node `name` to strong, the node `n` to weak and the reference edge would have opposite direction.

Creating a reference between a source and a target node (i.e., `$t=&$s`) copies all the values of the target node to all index nodes connected with the target node by a reference edge directed to the target node—this is done to not affect value sets of these nodes. Next, it removes all the edges from the target node and adds a new oriented reference edge from the target node to the source node. In order to follow the semantics of PHP references (an analogy to UNIX filesystem hard links), if a removed reference edge causes disconnection of nodes previously connected via the node whose edge is removed, new reference edges between disconnected nodes are added to keep the original reference information (except for the modified one) in the resulting graph. The strong/weak tag is copied from the source to the target node.

An assignment statement may represent assigning multiple source index nodes to multiple target index nodes (i.e., `$t[$i]=$s[$i]` where `$i` has possibly multiple values). In the case of assignment to multiple target nodes, the method is the same same as in the case of assignment to a single target node except that a weak update of the target nodes and their aliases is performed. That is, the target nodes and all their aliases are marked as weak; the original direction of the reference edges as well as the original value edges are preserved. In the case of assignment to the unknown index node (e.g., `$a[rand()]=1`), the set of the assigned values is added to the *unknown* field.

Our approach maintains the information about the cause of a given variable having a particular value (i.e., that a variable `$x` has the value `p` because of the alias with a variable `$y`). Hence, in the presence of uncertain references, it is possible to perform more precise updates than in [4].

C. Static analysis

Our static analysis stems from context-sensitive, control-flow-sensitive, path-insensitive inter-procedural static analysis introduced in [4]. For each program variable and each program point, we track information about its aliases, literal values, types, the certainty of this information and taint and sanitization status of value nodes using the points-to graph.

Our analysis uses a combination of concrete and symbolic execution when propagating literal values through operations. If all inputs of an operation are concrete, the explicit version of the operation is used, otherwise the symbolic version is used. By using concrete operations, we reduce the imprecision; here, we use the reference PHP implementation as in [4] and [1]. As to modeling the symbolic versions, we model arithmetic operations as well as operations with strings. For modeling string operations, we use automata-based approach presented in [22].

Data in the sink	**Exploit**
Tainted and match an attack pattern.	SQL injection, XSS.
Tainted and not sanitized.	*Data not escaped:* SQL injection, XSS.
Tainted or can be a null value.	*Data potentially not filtered, can be manipulated by a user:* Sensitive information leakage, semantic URL attack, spoofed form submissions, spoofed HTTP request.
Related to a current session and not guarded.	CSRF attack, session fixation, session hijacking.

TABLE I
POTENTIAL VULNERABILITIES IDENTIFIED BY THE ANALYSIS.

Information about the taint status is propagated through built-in operations in the following way. We use different taint markings for different sources of data. Contrary to other approaches, we do not remove the taint status after processing data with any operation. This has two reasons: (1) A correct escaping operation is identified not only by the source of data but also by the sink. (2) We use the taint information to detect the data that can be manipulated by the user. The information is used to detect vulnerabilites additional to those caused by improper escaping. Instead of removing the taint status, we track the sanitization status. Thus, for each sanitization status and for each taint marking, we track whether data with the taint marking are sanitized using an appropriate sanitization routine.

D. Detection of vulnerabilities

We detect several categories of vulnerabilities shown in Table I. The first category of vulnerabilities corresponds to data at sinks that match a given attack pattern. The list of attack patterns can be configured by the developer.

The second category of vulnerabilities corresponds to data that are not correctly escaped. For each taint marking and critical command there is a configurable list of escaping operations. If the data is not correctly escaped but does not match any attack pattern, it is likely sanitized using a custom sanitization routine. Since the list of attack patterns may not be complete and there can be an error in custom sanitization, these alarms can be also reported. To facilitate the check whether the data is sanitized using custom sanitization routine we track sanitization status also for string replacement operations and supply a list of such operations to the developer as potential sanitizers.

The third category of vulnerabilities is identified by data in sinks that are correctly sanitized, but are tainted or can be null. The developer can analyze the filtering status of data by inspecting their possible values and the conditions that must hold at a given program point. This way, errors in blacklist filtering can be discovered.

The last category of vulnerabilities is related to sessions; i.e., critical commands not guarded by any condition comparing the token in the request with some data stored at the server, e.g., using a session mechanism. Last but not least, we detect general programming errors such as dereferences of null pointers.

E. Path-sensitivite validation of vulnerabilities

In this phase, for each vulnerability that is uncertain we try to prove the unfeasibility of paths leading to the vulnerability. We identify the program points that contribute to the uncertainty of the vulnerability. These program points correspond to (1) join points of branching statements where some branches do not lead to the vulnerability or causes of the vulnerability are different and to (2) an access to data that cannot be certainly identified and that can lead to the vulnerability. An example of the case (1) is at line 7, Fig. 1. The fact whether the variable `$name` is tainted is uncertain and it depends on the condition `$tainted = true`. An example of the case (2) is at line 9, Fig. 1. Again, the fact whether the variable `$users[$index]` is tainted remains uncertain and it depends on the same condition.

We collect the conditions that must hold in order for these program points to lead to a vulnerability and the conditions that must hold to reach the critical command corresponding to the vulnerability. Then we use an SMT solver to find a solution of the conjunction of these conditions. If the SMT solver proves these conditions unsatisfiable, the vulnerability is unfeasible. However, if the SMT solver proves the formula, the vulnerability can be still unfeasible, due to the dependencies between variables in the conditions.

F. Demonstration of our approach

Using the code fragments in Fig. 1 we show how our approach models PHP data structures, references, operations, and how it resolves dynamic statements. We also show how it is capable of distinguishing between certain and uncertain vulnerabilities.

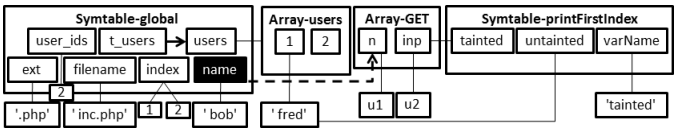

Fig. 2. Points-to graph after analyzing the code in Fig. 1 (ignoring the include statement). Index nodes contained in the storage node Symtable-global correspond to global variables, nodes contained in the storage node Array-users correspond to indices of an array, and nodes contained in the storage node Symtable-printFirstIndex correspond to local variables. Weak index nodes are black. Certain edges are showed using solid lines, uncertain edges using dashed lines, reference edges are bold.

Fig. 2 shows a points-to graph after executing the code in Fig. 1. The statement at line 3 creates a reference edge from the node `t_users` to the node `users`. This edge is used to access the value node `'fred'` at line 4. Because the node is not tainted, no vulnerability is identified. At the join point at line 7, edges going from the nodes `name` and `index` are merged, the reference edge from the node `name` becomes uncertain, and the index node `name` becomes weak. Thus, the value set of the node `index` is {`1, 2`}, while the value set of the node `name` is {`bob, user input`}. The size of a value set larger than one indicates that the values are uncertain. The corresponding uncertain vulnerabilities at lines 11 and 12 will be validated path-sensitively. The condition that must hold to make the variable $name$ tainted is $tainted = true$, the condition that must hold to reach the appropriate critical command is $tainted = false$. The conjunction of these conditions is unsatisfiable, thus no vulnerability is reported. The second alarm is filtered out in a similar way.

All inputs of the concatenation operation at line 11 are concrete; hence the analysis can use the concrete version of the instruction which results in a concrete, precise value stored in the value node connected to the index node `filename`. Similarly, the analysis uses the concrete version for the `strpos` operation at line 12. Consequently, the value of `filename` is known at line 13—the dynamic include can be resolved. Now, more precise CFG is constructed and the static analysis step is performed with this refined CFG. Note that because we model the important operations symbolically, our analysis is able to handle even more complex cases where inputs of operations are not concrete.

The points-to graph is used to correctly handle also the variable-variable construct at line 17. The value of the local variable `$varName` is used as the index to the storage node `Symtable-printFirstIndex`. The result corresponds to the value node `$_GET['inp']` which is tainted, the vulnerability is therefore reported.

The analysis performs type inference, hence, it is known that the variable `$user_ids` is scalar at line 20, and does not perform the assignment. Clearly, this statement is likely a bug, so the analysis reports a warning in such cases.

G. Evaluation

To evaluate our approach, we analyzed a snippet of code from a real web application using the Pixy tool. We identified sources of imprecision and showed whether they can be handled by our approach by tracking it by hand.

Our approach handles the causes shown in rows (1)-(7) of the Table II in the following way: (1) it performs taint analysis, thus, the corresponding vulnerability can be detected in a similar way as in Pixy. (2) Reading data from uninitialized variable results in reading the `null` value[1]. This is considered vulnerability only in several contexts discussed in Section IV-D. With respect to (3), our approach makes it possible to find the conditions under which data is tainted. Hence, it is possible to filter out the cases when data is not sanitized with routine `htmlspecialchars` and `HTMLSAFE` is `false`. Cases in (4) can be correctly handled by path-sensitive phase of our approach. With respect to (5), our approach makes it possible to filter out cases when data does not match any attack pattern in the critical command. Additionally, our approach helps the developer to check custom santization by showing all pattern matching functions applied on data at a given program point. As well as Pixy, our approach correctly reports an access to an undefined field of the `$lang` array at the last line of the case (6). Unfortunately, the same alarm will be emitted also if we fix the problem by validating the input of the function `mktime`. The fix is shown in the comment at the first line of the example. Then, the output of the function `date` is a name of a month and because the array `$lang` is initialized for every such an index, the alarm is false positive. It would

[1]We assume that the value of a directive `register_globals` is `off`.

	Description	Code example	Pixy	Us
1	Sanitization after an assignment instead of sanitization before.	`intval($id = $_GET['id'])`	ok	ok
2	Using of uninitialized variables.	`if (isset($_GET['i'])) $i = $_GET['i'];` `echo $i;`	x	ok
3	Conditional sanitization based on configuration.	`if (HTMLSAFE)  $var = htmlspecialchars(...);`	x	ok

4	Variable is defined under the same condition under which is used.	`if ($editcom) {`
		`  if (isset($_GET['id'])) $id = ...;`
		`  else die();`
		`}`
		`...`
		`if ($editcom) func($id);`
			x	ok

5	Sanitization using regular expressions.	`$var = preg_replace();`	x	ok

6	Access to member of array given by return value of library function.	`//if ($m<1 || $m>12) die();`
		`$mt=date('F', mktime(0,0,0,$m,1,$y));`
		`echo $lang["$mt"];`
			ok/x	ok/x

7	Use of unsanitized data from session in SQL command.	`$username = $_SESSION['username'];` `db_com($username);`	x	ok

TABLE II
CAUSES OF ALARMS AND MISSED ERRORS WHEN ANALYZING MYBLOGGIE. OK STANDS FOR A CORRECT RESULT, X AN INCORRECT ONE.

be possible to correctly handle this situation by specifying the preconditions and postconditions of functions `mktime` and `date`. Our approach tracks the restriction over the values of the variable `$m` and it thus makes it possible to check the preconditions of the function `mktime`. Finally, our approach uses taint marking for data from sessions and makes it possible to report the vulnerability corresponding to (7).

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a new approach to discovery of bugs inside web applications written in PHP. While being based on known techniques, to our knowledge we are the first who combined them into a single one and improved them to face the most critical issues. We proposed precise modeling of aliasing and taint analysis capable of detecting the most of vulnerabilities presented in the literature, e.g., [14]. Last but not least, the novel contribution of our approach is its path-sensitivity. Even though false alarms caused by path-insensitivity of existing tools were reported, we are not aware of any approach that addresses this issue. We demonstrated issues of existing approaches and showed how they are handled by our approach. Although demonstrated on PHP, the approach is general and can be modified and applied also on other languages for web development.

We believe that the analysis is scalable, expensive path-sensitive step of the analysis is performed only when it is necessary, i.e., to confirm vulnerabilities that can be false alarms. False positives can still appear even after the path-sensitive step. However, precise analysis combined with path-sensitive step and employed vulnerability detection promises both a lower false-positive rate and higher error coverage compared to related approaches as showed in Sect. IV-G.

Once a prototype implementation is completed, we will evaluate scalability of the method on several real web applications. Future work will also investigate and evaluate the existing techniques to analyze and refine path-conditions and possibly adapt them to be usable in the context of PHP applications. For practical reasons, it should be also possible to manually resolve problematic parts of the analyzed code. This could be achieved by introducing hints in the form of code annotations. Code annotations should be used, e.g., to specify the values, types, sanitization status of a given variable at a given program point.

REFERENCES

[1] Artzi et al. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. on Soft. Eng.*, 36(4), 2010.
[2] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *Static Analysis*, LNCS. Springer, 2008.
[3] D. Balzarotti et al. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. *S&P'2008*, 2008.
[4] P. Biggar and D. Gregg. Static analysis of dynamic scripting languages, 2009.
[5] Common weakness enumeration. http://cwe.mitre.org/top25/.
[6] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI '02*. ACM, 2002.
[7] D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In *Static Analysis*, LNCS. Springer, 2006.
[8] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08*. ACM, 2008.
[9] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *PRDC'07*. IEEE CS, 2007.
[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04*, 2004.
[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *S&P'06*. IEEE, 2006.
[12] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05*. ACM, 2005.
[13] PHP—Personal Home Pages. http://www.php.net.
[14] C. Shiflett. *Essential PHP security*. O'Reilly, 2006.
[15] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 2006.
[16] M. Taghdiri, G. Snelting, and C. Sinz. Information flow analysis via path condition refinement. In *FAST 2010*. Springer-Verlag, 2010.
[17] U.S. Census Bureau News, May 2011. http://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf.
[18] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *DSN '09*, 2009.
[19] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. PLDI '07. ACM, 2007.
[20] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. *ICSE '08*, 2008.
[21] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium,*, 2006.
[22] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. *TACAS'10*, 2010.