Simulation of Built-in PHP Features for Precise Static Code Analysis

Johannes Dahse
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
johannes.dahse@rub.de

Thorsten Holz
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
thorsten.holz@rub.de

Abstract—**The World Wide Web grew rapidly during the last decades and is used by millions of people every day for online shopping, banking, networking, and other activities. Many of these websites are developed with PHP, the most popular scripting language on the Web. However, PHP code is prone to different types of critical security vulnerabilities that can lead to data leakage, server compromise, or attacks against an application's users. This problem can be addressed by analyzing the source code of the application for security vulnerabilities before the application is deployed on a web server. In this paper, we present a novel approach for the precise static analysis of PHP code to detect security vulnerabilities in web applications. As dismissed by previous work in this area, a comprehensive configuration and simulation of over 900 PHP built-in features allows us to precisely model the highly dynamic PHP language. By performing an intra- and inter-procedural data flow analysis and by creating block and function summaries, we are able to efficiently perform a backward-directed taint analysis for 20 different types of vulnerabilities. Furthermore, string analysis enables us to validate sanitization in a context-sensitive manner. Our method is the first to perform fine-grained analysis of the interaction between different types of sanitization, encoding, sources, sinks, markup contexts, and PHP settings. We implemented a prototype of our approach in a tool called RIPS. Our evaluation shows that RIPS is capable of finding severe vulnerabilities in popular real-world applications: we reported 73 previously unknown vulnerabilities in five well-known PHP applications such as *phpBB*, *osCommerce*, and the conference management software *HotCRP*.**

I. Introduction

According to *W^{3}Techs*, PHP is the most popular server-side programming language of all recognized websites with a share of 81.4% [40]. Many well-known websites such as *Facebook* and *Wikipedia* as well as the most commonly used content management systems [39] are written in PHP. Due to its weakly and dynamically typed syntax and a large number of built-in features, the language is easy to learn for beginners. However, PHP has a large number of complex language characteristics that lead to many intricacies in practice. As a result, PHP applications are prone to software vulnerabilities: in the MITRE CVE database [26], about 29% of all security vulnerabilities found in computer software are related to PHP. The wide distribution of PHP and the large number of PHP-related vulnerabilities lead to a high interest of finding and patching security vulnerabilities in PHP source code (e. g., [2, 18, 34, 36, 41, 43]).

Detection of Taint-Style Vulnerabilities: A security vulnerability occurs when data supplied by the user is used in critical operations of the application and is not sanitized sufficiently. An attacker might be able to exploit this flaw by injecting malicious input that changes the behavior or result of this operation [33]. These kinds of vulnerabilities are called *taint-style vulnerabilities* because untrusted sources such as user-supplied data are considered as *tainted* and literally flow into vulnerable parts of the program (referred to as *sensitive sinks*) [5, 24, 29, 35].

Given the fact that large applications can have many thousands lines of code and time is limited by costs, a manual source code review might be incomplete and inefficient. *Static Code Analysis* (SCA) tools can help code reviewers to minimize the time and costs of a review and convey expertise in security to the user by encapsulating knowledge in a limited degree. They automate the process of finding security vulnerabilities in source code by using taint analysis [35]. Here, the data flow between sources and sinks is modeled and analyzed for sanitization, which is a hard problem specifically for highly dynamic languages such as PHP.

Current Approaches: Recent work in this area focused on the detection of only a limited number of vulnerability types such as Cross-Site Scripting (XSS) and SQL injection (SQLi) vulnerabilities [18, 41, 42] or the analysis of sanitization routines [13]. Furthermore, existing approaches are typically imprecise in the sense that some language features such as built-in sanitization or string manipulation functions and markup contexts are not modeled accurately. As a result, certain types of vulnerabilities and sanitization cannot be found by such approaches. For example, *Saner* [2] relies on manually generated test cases, which implies that it can only detect the vulnerabilities encoded within the tool. Furthermore, other approaches such as the one presented by Xie and Aiken [43] do not model built-in functions and thus miss important attack and defense vectors. Commercial tools that support the PHP language focus on the detection of vulnerabilities in three or more programming languages. Consequently, these tools are building a more generic model and are missing many PHP-specific vulnerabilities and characteristics [23].

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23262

Our Approach: In this paper, we introduce a novel approach for the precise static analysis of PHP code. Based on the insight that prior work missed vulnerabilities due to not precisely modeling the specifics of the PHP language, we perform a comprehensive analysis and simulation of built-in language features such as 952 PHP built-in functions with respect to the called arguments. This allows us to accurately analyze the data flow, to detect various sources and sinks, and to analyze sanitization in a more comprehensive way compared to prior work in this area. As a result, we find more security vulnerabilities with higher accuracy. More specifically, we perform an intra- and inter-procedural data flow analysis to create summaries of the data flow within the application to detect taint-style vulnerabilities very efficiently. We perform context-sensitive string analysis to refine our taint analysis results based on the current markup context, source type, and PHP configuration. Generalizing our approach to different languages is possible by modelling its (less diverse) built-in features while the analysis algorithms remain the same.

We implemented our approach for PHP in a tool called RIPS and evaluated it by analyzing popular and complex real-world applications such as *phpBB*, *HotCRP*, and *osCommerce*. In total, we analyzed 1 390 files with almost half a million lines of PHP code. We found that on average, every 4th line of code required taint analysis. Overall, we detected and reported 73 previously unknown vulnerabilities such as for example three SQL injection vulnerabilities in *HotCRP* and several XSS vulnerabilities in *osCommerce*. We also analyzed several web applications that were used during the evaluation of prior work in this area and found that RIPS outperforms existing tools.

In summary, the contributions of this paper are as follows:

- We demonstrate that a precise modeling of the complex characteristics of the PHP language is essential to detect weak sanitization and to find security vulnerabilities in modern PHP applications. To this end, we are the first to support the detection of 20 different types of security vulnerabilities.
- We introduce the algorithms of our tool that is specifically focusing on the specifics of the PHP language. The tool is the first to perform a fine-grained analysis of a large number of PHP built-in features. It performs string analysis for context-sensitive vulnerability confirmation of 45 different markup contexts with respect to the interaction of sink, source type, sanitization, encoding, and PHP configuration.
- We implemented a prototype of our approach in a tool called RIPS. We evaluate our approach on large, real-world applications and demonstrate that RIPS is capable of finding several previously known and unknown, severe vulnerabilities. Furthermore, we compare our results to previous work in this area and demonstrate that RIPS outperforms state-of-the-art tools.

II. Technical Background

In contrast to prior work, we include edge cases of complex taint-style vulnerabilities in our analysis. We thus first provide a brief overview of such vulnerabilities and then examine some specific features and characteristics of the PHP language to illustrate the difficulties when performing PHP code analysis.

A. *Taint-style Vulnerabilities*

In the following, we examine the concept for two common taint-style vulnerabilities where tainted data flows into a sensitive sink. More specifically, we focus on sanitization approaches and weaknesses in different scenarios that have to be identified by our tool in a precise manner.

1) SQL Injection: Web applications are often connected to a database that stores sensitive data like passwords or credit card numbers. If a web application dynamically generates a SQL query with unsanitized user input, an attacker can potentially inject her own SQL syntax to modify the query. This type of vulnerability is well known and called *SQL injection* (SQLi) [10]. Depending on the environment, the attacker can potentially extract sensitive data from the database, modify data, or compromise the web server.

To patch such a vulnerability, the user input must be sanitized before it is embedded into the query. For example, all quotes must be escaped within a quoted string such that evasion is not possible. For MySQL, the PHP built-in function `mysql_real_escape_string()` adds a preceding backslash to every single quote, double quote, and backslash to neutralize their syntactical effect.

However, if the user input is not embedded into quotes within the SQL query, no quotes are required for evasion (see Listing 1). In this case, escaping is not sufficient for sanitization and the application is still vulnerable. To find such vulnerabilities, we not only have to model sanitization routines, but also consider if they are applied to the right markup context. A complementary way to prevent SQLi vulnerabilities is to use *prepared statements* [37].

```php
$id = mysql_real_escape_string($_GET['id']);
mysql_query("SELECT data FROM users WHERE id = $id");
```

Listing 1: Insufficient sanitization of a SQL query.

2) Cross-Site Scripting: Cross-Site Scripting (XSS) [21] is the most common security vulnerability in web applications [34]. It occurs when user input is reflected to the HTML result of the application in an unsanitized way. It is then possible to inject HTML markup into the response page that is rendered by the client's browser. An attacker can abuse this behavior by embedding malicious code into the response that for example locally defaces the web site or steals cookie information.

To patch such a vulnerability, the output has to be validated. Meta characters like < and > as well as quotes must be replaced by their corresponding HTML entities (e. g., `&lt;` and `&gt;`). The characters will still be displayed by the browser, but not rendered as HTML markup. In PHP, the built-in function `htmlentities()` can be used for output validation. As with SQL queries, however, it is important to adjust sanitization to the context of the HTML markup [16].

Listing 2 depicts a snippet of an application that is vulnerable to XSS. Although sanitization is applied, the context of the injection still allows an attacker to break the markup and inject Javascript code. The function `htmlentities()` only sanitizes the characters < and > as well as double quotes by default. Note that the function does not sanitize single quotes. Thus, an attacker can break the single quoted `href`-attribute and inject an *eventhandler* that is attached to the link tag (e. g., `' onmouseover='alert(1)`). To encode single quotes to the HTML entity `&#39;`, the parameter `ENT_QUOTES` must be added to the function `htmlentities()`.

```php
$page = htmlentities($_GET['page']);
echo "<a href='$page'>click</a>";
```

Listing 2: Insufficient sanitization with `htmlentities()`.

In our example, however, the application would still be vulnerable. Instead of breaking the markup, an attacker can abuse the diversity of web browsers and inject a Javascript protocol handler into the link (e. g., `javascript:alert(1)`). This injection does not need any meta characters that are encoded by `htmlentities()`.

Note that there are several other scopes that need to be considered when using sanitization. For example, when user input is used within `style` and `script` tags, or within eventhandler attributes, additional sanitization is required. Previous work missed to take the different scopes and their intrinsic behaviors into account.

B. Intricacies of the PHP language

PHP is the fastest growing and most popular script language for web applications. It is a highly dynamic language with lots of complicated semantics [3] that are frequently used by modern web applications [12]. In this section, we introduce the most important language features our tool has to model precisely in order to correctly identify the flow of tainted data into sensitive sinks. In particular, the flow of tainted strings is of interest for taint-style vulnerabilities.

1) Dynamic and Weak Typing: PHP is a *dynamically typed* language and does not require an explicit declaration of variables. The variable type is inferred on the first assignment at runtime. Additionally, PHP is a *weakly typed* language and its variables are not bound to a specific data type. Thus, data types can be mixed with other data types at runtime. In Listing 3 the string *test* is evaluated to *0* to fit the mathematical operation and added to *1*. The integer result is stored in the variable `$var2` whose previous data type was *string*.

```php
$var1 = 1; $var2 = 'test';
$var2 = $var1 + $var2; // 1
```

Listing 3: Addition of a string and an integer in PHP.

2) Variable Variables: Variables are usually introduced with the dollar character followed by an alphanumeric, case-sensitive name. However, in PHP the name can also be an expression, for example retrieved from another variable or the return value of a function call that is only known at runtime (see Listing 4). This makes it extremely difficult to analyze the PHP language statically.

```php
$name = "x"; $x = "test";
echo $$name; // test
$y = ${getVar()};
```

Listing 4: Variable variables in PHP.

3) Dynamic Arrays: Arrays are hash-tables that map numbers or strings (referred to as *keys*) to values. The key name can be omitted when initializing an array and generated at runtime (see Listing 5). Furthermore, keys and values can be dynamic, as well as the array name itself. When performing a static analysis, it is a challenge to precisely model such a dynamic array structure and the dynamic access to it.

```php
$var = 6;
$arr = array('a', "4" => $var, 'foo' => 'c', 'd');
$arr[] = 'e';
// Array ([0] => a [4] => 6 [foo] => c [5] => d [6] => e)
print $arr[$var]; // e
```

Listing 5: Dynamically generated key names in an array.

4) Dynamic Constants: In PHP, it is possible to define *constant* scalar values as in other programming languages like *C*. However, the constant name can be dynamically defined by the built-in function `define()` and dynamically accessed by the built-in function `constant()`. Although a constant may not change once it is defined, it is possible to define constants conditionally in the program flow or dynamically generated with user input.

5) Dynamic Functions: Several functions with the same name can be defined conditionally by the developer. Thus, a totally different function may be called depending on the program flow. It is also possible to define a function `B()` within another function `A()` that is only present during the execution of `A()`. Further, the built-in functions `func_get_arg()` and `func_get_args()` allow to dynamically fetch arguments of the function call by index.

```php
1 $name = 'step' . (int)$_GET['id'];
2 $name();
3 array_walk($arr = array(1), $name);
```

Listing 6: Dynamically built and executed function name.

Listing 6 illustrates two different possibilities to *call* a function dynamically (*Reflection*). The function name is built dynamically in line 1 and is only known at runtime. It is called in line 2 by adding parenthesis to the variable `$name` and used in line 3 as *callback* function. The built-in function `create_function()` dynamically creates function code.

6) Dynamic Code: The `eval` operator and the built-in function `assert()` allows to directly evaluate PHP code that is passed as string to its first argument. Other functions such as `preg_replace()` allow the execution of dynamic PHP code when used with certain modifiers. Dynamically generated code is very challenging to analyze if the executed PHP code is only known at runtime and cannot be reconstructed statically. Furthermore, it introduces critical security vulnerabilities.

7) Dynamic Includes: The code of large PHP projects is often split into several files and directories. At runtime, the code can be merged and executed conditionally. The PHP operator `include` opens a specified file, evaluates its PHP code, and returns to the code after the `include` operator. It can be used as expression within any other expression. Furthermore, the file name of an inclusion can be built dynamically which implies that it is challenging to reconstruct it statically in complex applications. During static analysis it is crucial to resolve all file inclusions to analyze the PHP code correctly. Additionally, tainted data within the file name leads to a *File Inclusion* vulnerability.

8) Built-in Functions: Depending on the configuration and version, PHP is shipped with several *extensions* that provide built-in functions to the developer. In total, 228 extensions with 5 701 built-in functions are documented [9]. Static code analysis is losing precision whenever a built-in function is called that is not (or faulty) configured in the tool. There are plenty of built-in functions that must be recognized as sensitive sinks or sources of tainted data.

```php
list($day, $month, $year) = $_GET['time'];
printf("Today is %d %s %d", $day, $month, $year);
```

Listing 7: XSS via GET parameter `time[month]`.

Furthermore, the data flow through built-in functions and operators must be analyzed precisely, as demonstrated in Listing 7. First, the `list` operator is used to extract the day, month, and year from the GET parameter *time* that is passed as an array and assigns it to three variables. The built-in function `printf()` is then used to print the supplied date according to a certain format. An XSS vulnerability exists for the second time element stored in `$month`, while the other variables are only printed as numeric values. We need to analyze the string format first to recognize the data flow precisely, a problem not addressed by prior work in this area.

9) Superglobals: Superglobals are built-in arrays that are initialized from the PHP interpreter and are available in all *scopes*. They allow quick access to the HTTP request header, environment, and global scope which can hold tainted user data. Often developers only consider `$_GET`, `$_POST`, `$_COOKIE`, and `$_REQUEST` values as source, but forget about `$_FILES` and `$_SERVER` keys. Especially the `$_SERVER` keys `PHP_SELF` and `HTTP_HOST` are assumed as static values but can be altered by an attacker. Again, we need to know precisely which keys can be tainted and which not and thus we have to model array key accesses precisely to detect tainted and untainted keys, a problem also not considered in prior work.

III. Precise, Static Analysis of PHP Code

Taking the peculiarities of PHP into account, we now introduce our approach for static code analysis to overcome the impression of prior work in this area. We aim to detect all taint-style vulnerabilities and try to model the PHP language as precisely as possible. In this section, we describe how we addressed the challenges of accurately analyzing a highly dynamic language and how we implemented the methodology of our approach in a new code analysis engine called RIPS.

A. General Overview

Our approach uses block, function, and file *summaries* to store the results of data flow analysis within each unit and to built an abstract data flow model for efficient analysis [43]. More precisely, the following steps are taken:

1) For each PHP file in the project, an *Abstract Syntax Tree* (AST) based on PHP's open source internals is built. Furthermore, all user-defined functions are extracted and relevant information like the name and parameters are stored in the environment. The body of the function is saved as separate AST and is removed from the *main* AST of the parsed file.
2) We start transforming each *main* AST into a *Control Flow Graph* (CFG). Whenever a node of the AST performs a conditional jump, a new basic block is created and connected to the previous basic block with a block edge. The jump condition is added to the block edge and the following AST nodes are added to the new basic block.
3) We simulate the data flow of each basic block as soon as a new basic block is created (see Section III-C). The main advantage is that the analysis of a basic block is only dependent on previous basic blocks when performing backwards-directed data flow analysis. Furthermore, the analysis results are integrated into the so called *block summary* that is created during simulation. It sums up the data flow within a basic block.
4) If a call to a previously unknown user-defined function is encountered during simulation, the CFG is built from the function AST and a *function summary* is created once with intra-procedural analysis (see Section III-D). Then, the pre- and post-conditions for this function can be extracted from the summary and inter-procedural analysis is performed (see Section III-E). Finally, the construction of the main CFG continues.
5) We conduct a taint analysis beginning from the currently simulated basic block for each vulnerable parameter of a user-defined function or configured sensitive sink (see Section III-G).

Furthermore, we perform the following novel analysis steps to refine our results:

- Next to data types, we assign sanitization tags for different vulnerability types and encoding information to data symbols. This allows us to model different sanitization methods throughout the data flow.
- We model a total of 952 built-in functions to recognize a variety of data flow, sanitization, sources, and sinks. This step is critical to perform a comprehensive security analysis.
- Instead of connecting CFGs of included files into the current CFG, we model included files as functions. This prevents redundant analysis of included files and shortens the analysis paths of the CFG.
- We simulate block edges and summarize their sanitization effects (see Section III-F).
- Our string and taint analysis is performed backwards-directed. Intermediate results fetched from the block summaries are cached for each basic block which enables a highly performant analysis.
- We perform context-sensitive string analysis to refine our taint analysis results based on the current markup context, source type, and PHP configuration.

B. CFGBuilder

The `CFGBuilder` is initiated with the AST nodes of each main and function AST. It splits conditional program flow into linked basic blocks and initializes their simulation. Its algorithm is shown in Listing 8 and the evaluated statements are depicted in Listing 9.

```
function CFGBuilder(nodes,condition,pEntryBlock,pNextBlock){
  currBlock := new BasicBlock();
  if (pEntryBlock) {
    new BlockEdge(pEntryBlock, currBlock, condition);
  }
  foreach (node n in nodes) {
    if (n is a JSTMT) {
      simulate(currBlock);
      nextBlock := new BasicBlock();
      foreach (branch b in n) {
        CFGBuilder(b->nodes, b->cond, currBlock,nextBlock);
      }
      currBlock = nextBlock;
    }
    else if (n is a LSTMT) {
      addLoopVariables(n->cond, currBlock);
      simulate(currBlock);
      nextBlock := new BasicBlock();
      CFGBuilder(n->nodes, null, currBlock, nextBlock);
      currBlock = nextBlock;
    }
    else if (n is a SSTMT) {
      break;
    }
    else if (n is a RSTMT) {
      currBlock->nodes[] = n;
      simulate(currBlock);
      return;
    }
    else {
      currBlock->nodes[] = n;
    }
  }
  simulate(currBlock);
  if (pNextBlock and !currBlock->isExitBlock)
    new BlockEdge(currBlock, pNextBlock);
}
```

Listing 8: CFG builder algorithm.

```
JSTMT (J) ::= If | Switch | Try | Ternary | LogicalOr
LSTMT (L) ::= For | Foreach | While | Do
SSTMT (S) ::= break | continue | throw
RSTMT (R) ::= return
```

Listing 9: Language Statements.

First, the `CFGBuilder` creates a new `BasicBlock` which is stored as `currentBlock`. Next, it loops through all root nodes of the AST and adds all nodes that are *not* a statement to the node list of the `currentBlock`.

If a jump statement (JSTMT) is detected, a new `CFGBuilder` is initiated recursively for every branch the statement introduces and the new `currentBlock` is linked to the previous `currentBlock`. Each condition is added as first node of the basic block to ensure that it is part of the simulation process. For example, a variable declaration can occur within an `if`-condition and must be part of the AST.

Loops (LSTMT) are handled as one basic block. The loop condition is analyzed and *looped* variables are identified, such as a repeatedly incremented variable within a for-statement. For these variables, all possible values are considered during data flow analysis, e. g., when used to access an array by key. While this may introduce imprecision, our evaluation shows that this approach is sufficient to detect vulnerabilities in real-world applications (see Section IV-C1 for an example).

The `CFGBuilder` stops parsing nodes when the program flow is halted with a stop statement (SSTMT). All nodes after the stop statement are not reached during execution of the PHP code and are thus not added to the `currentBlock`. In the case of a `return` statement (RSTMT) or when the simulation of the `currentBlock` revealed an exit of the program flow, no block edge is created to the `parentNextBlock`.

The algorithm stops when all statements are parsed and all subnodes are added to a basic block. During simulation of a basic block, analysis based on previously linked basic blocks is performed throughout the CFG.

C. Simulating Basic Blocks

The simulation of a basic block is initiated during CFG construction whenever a statement occurs that splits the control flow into new basic blocks. Then we simulate the *current block* before we move on to the next basic block.

The purpose of the simulation is to create a summary of the data flow within *one* basic block of the CFG. To do so, we loop through all nodes of the basic block and parse assignments and function calls. These nodes can perform an assignment whose value is stored in the *block summary* as a *symbol*. Furthermore, we parse `global`, `exit`, and `return` statements.

1) Symbols: Symbols are our language set that represent memory locations. These can be assigned to another memory location or a basic scalar value. With the help of these symbols, static and tainted data is modeled for further analysis.

The most basic symbol `Value` represents static strings and integer values. Variables are represented by the symbol `Variable` and constants are represented by the `Constant` symbol. The symbol `ArrayDimFetch` extends the symbol `Variable` with a *dimension* and represents the *access* of an array. The dimension is a key (or multiple keys when a multi-dimensional array is accessed) that is represented by another symbol.

A declared array is transformed into a `ArrayDimTree` symbol that is a directed graph representation of the array as described by Jovanovic et. al. [18]. The edges represent the array keys and link to nodes that represent the array values. In case of a multi-dimensional array, the node is another `ArrayDimTree` with more edges. This graph structure allows to handle multi-dimensional arrays and its accesses precisely.

One location can also represent several symbols, for example when several non-static symbols are concatenated. These are stored within a `Concat` symbol.

Each symbol (except for the basic symbol `Value`) has a *type*, *encoding*, and *sanitization* status. The default type of each symbol is *string* and it is not encoded nor sanitized. If a typecast occurs, we infer the new type from the AST and assign it to the symbol. If the symbol's encoding is changed via built-in function, the encoding type is pushed to the symbol's encoding stack. On decoding, it is removed from this stack again. Furthermore, each symbol can be sanitized against different types of vulnerabilities which are mapped to a vulnerability *tag*. These tags are assigned to symbols on sanitization. In Section III-G1, we discuss how the final sanitization status of a symbol is determined.

Additionally, we introduce a *Boolean* symbol. It is used to transfer the sanitization status of a symbol sanitized by a block edge. The details are explained in Section III-F.

2) Block Summary: The *symbols* are used in the *block summary* that allows efficient backwards analysis of upcoming basic blocks in the CFG. Our block summary is represented by the following properties:

- **DataFlow** maps assigned location names to the assigned *symbol*. In case of a defined array, the array name maps to a `ArrayDimTree` symbol whose keys can be fetched.
- **Constants** maps defined constant names to the assigned *symbol*. Uniquely defined constants with static values are stored in the environment for faster access during analysis.
- **GlobalDefines** records variable names that are put into global scope. These are later used to determine interprocedural effects of a function (see Section III-E).
- **ReturnValue** records the return value of the basic block. Note, that each basic block can only have one return symbol and the `return` statement is the last node in the node list. Dead code behind a `return` or `exit` statement is removed.
- **registerGlobals** states if the basic block enables register globals [6] due to built-in functions like `extract()` or `import_request_variables()`.
- **isExitBlock** states if the basic block exits the program flow due to the `exit` or `die` operator, or by calling a user-defined *isExitFunction* (see Section III-D).

3) Data Flow Analysis: In order to summarize the effect of a basic block in the *block summary*, the data flow in this block is analyzed. Our algorithm evaluates the AST of each assignment by transforming its nodes into symbols. We visit the nodes of each AST top-down while we keep track of the data type, encoding, and sanitization tags.

```php
$y = (int)$_GET['p'];
$z = $x . $y;
```

Listing 10: A basic block with two nodes.

An example for two assignments is given in Listing 10. In the first assignment, an integer typecast is found that switches the data type of all subnodes to *int*. The subnode is evaluated to an `ArrayDimFetch` symbol. Finally, this symbol is mapped to the location `y` in the current block's *DataFlow* property. The `ArrayDimFetch` symbol has the name *_GET*, the type *int*, and one dimension with a `Value` symbol *p*.

In the second line, an assignment to location `z` is parsed. Here, a string concatenation is found and the left and right part is evaluated. While the `Variable` symbol with the name *x* on the left remains unresolved, the `Variable` *y* on the right can be resolved from the previously added `ArrayDimFetch` symbol in the *DataFlow* property `y`. Both symbols are added to a `Concat` symbol which is then mapped to the location `z` in the *DataFlow* property.

We do not model assignments by reference (*aliases*) in great detail yet since they are rarely used in modern PHP applications. However, we support function parameters passed-by-reference because these are sometimes used in custom sanitization functions and built-in functions like `array_walk()`. They are handled in a similar way to global variables (see Section III-D).

In case the location name is not static, backwards-directed string analysis is performed. If the result is one or more strings, the assigned symbol is added to these location names. Otherwise, if the result comes from user input, a *Variable Tampering* warning is issued. A *variable* variable within the assigned expression is handled in a similar way.

4) Simulating Includes and Dynamic Code: Includes are dynamic expressions in PHP and not static statements. *Includes* have a return value and can occur within conditions, assignments, or any other expression. In case the file name is not a static string, we try to reconstruct the name of the file that is included. All entry edges of the current basic block are visited recursively and all possible values are constructed from the block summaries of previous blocks. If the reconstructed file name is ambiguous, a regular expression is created and mapped to all available file names. If more than one file matches, we try to favor files in the same directory. Each possible included file is then handled as user-defined function that is called with empty arguments and all local variables in global scope.

Eval operations are handled in a similar way. First, we try to reconstruct the evaluated string by backwards-directed string analysis using previous block summaries. If neccessary, we decode multiple layers of encoded data (identified by the used built-in functions) to be able to also analyze obfuscated code. If we can parse the reconstructed string as PHP code, the code is handled in the same way as included PHP code. Dynamically generated code based on unsanitized user-input generates a *Code Execution* vulnerability report.

5) Simulating Built-in Functions: We model the data flow of 621 built-in functions. Each function is configured by name and affected parameters. Mainly, these functions can be categorized in the following seven groups:

- **alphanum (284):** Built-in functions such as `strlen()` or `md5()` effectively sanitize their argument by returning alphanumerical values only. These calls return a static `Value` symbol.
- **argument (122):** Other built-in functions such as `trim()` or `strrev()` return at least one of its argument fully or partly. A flow of tainted data is possible through these functions and the symbols of these arguments are returned. For handling the conversion between arrays and strings, this category is divided into functions that return an array, a single array element, or split a string argument into an array.
- **escape (20):** Some built-in functions sanitize against certain vulnerability types by escaping meta characters. As introduced in Section II-A1, the function `mysql_real_escape_string()` sanitizes against SQL injection vulnerabilities with **s**ingle and **d**ouble **q**uotes by adding an escaping backlash. Thus, the sanitization tags `SQLI_SQ` and `SQLI_DQ` are assigned to the returned symbol, but not `SQLI_NQ` (**n**o **q**uotes).

- **substring (6):** String functions such as `substr()` or `chunk_split()` return a substring of an argument. This can destroy previously added escaping by cutting off an escaped meta character and leaving behind an unescaped backslash. These functions are handled as *argument* functions but add a `SQLI_MI` tag.
- **encode (18):** Other functions such as `urlencode()` or `base64_encode()` sanitize against all vulnerability types by encoding all meta characters. Thus, the encoding type is assigned to the symbol's encoding stack.
- **decode (25):** Built-in functions such as `urldecode()` or `base64_decode()` can turn harmless user data into malicious data. Thus, all previously added sanitization tags are removed from the returned symbol if the encoding stack is empty. If the decoding type matches the encoding type on top of the encoding stack, the type is removed from the encoding stack.
- **callbacks (51):** Some built-in functions call other functions whose name is given as string argument. Examples are `array_walk()` or `set_error_handler()`. If the *callback* function's name can be reconstructed by string analysis, the function is analyzed intra- and inter-procedurally. If the function name is reconstructed only partly, a regular expression is generated and performed on all available function names to identify a possible subset of functions to be called.

Another 95 frequently used built-in functions cannot be generalized in the above categories or need further processing. These functions are modeled as plug-ins by our tool that are used whenever a call of the function is detected. For example, the built-in functions `htmlentities()` and `htmlspecialchars()` introduced in Section II-A2 sanitize input differently depending on their second argument. Thus, we first reconstruct the provided value in the second argument with string analysis and then add sanitization tags to the returned symbol accordingly. Other examples are built-in functions that use the format string syntax such as `printf()` and `sprintf()` that require in-depth analysis of the format string (see Section II-B8). Built-in functions not covered by our tool return the default value *1*.

D. Intra-procedural Analysis

While the data flow of built-in functions is known and configured, the data flow of user-defined functions needs to be analyzed and summarized first. If a user-defined function is called and it is not already simulated, a new CFG of the function is created. As described in the last sections, the CFG will consist of simulated basic blocks with block summaries. A *function summary* is created that summarizes the data flow of *all* basic blocks in the CFG and saved to the environment. For this purpose, a depth-first search through all basic blocks of the CFG is initiated. If a basic block has no outgoing edges, it either has a `return` statement, an `exit` statement, or it is the last block in the CFG. Based on these *end blocks*, a function summary with the following properties is created:

- **returnValues:** If the *end block* has a `returnValue`, its symbol is traced through previous basic blocks and all resolved symbols are added as function return value. This may include function parameters.
- **changedGlobalVars:** For each *end block* in the CFG, the possible set of altered global variables is generated by tracing globalized variables backwards to the basic block that put it into global scope (stated in the *GlobalDefines* summary).
- **isExitFunction:** Indicates if the function exits the program flow. This is the case if *all end blocks* are flagged as `isExitBlock`.

During taint analysis within a user-defined function (see Section III-G), the following properties can be added to the function summary. Their values are mapped to the vulnerability type of the current taint analysis.

- **sensitiveParams:** Lists the function's parameters that flow into a sensitive sink.
- **sensitiveGlobals:** Lists the local variables that are fetched from global scope and flow into a sensitive sink.

Recursive function calls are not handled for now. While this introduces unsoundness, we are not aware of any real-world web security vulnerability that only occurs within a certain level of recursion. Furthermore, since our algorithm is path-insensitive, such a vulnerability would most likely be detected.

E. Inter-procedural Analysis

After a user-defined function was simulated, the inter-procedural effects of a call of this function can be evaluated and changes to the current scope can be processed. For example, if the function was marked as a `isExitFunction` during simulation, the basic block of this call is consequently exiting the program flow. Thus, all outgoing `exitEdges` and upcoming dead code is removed from the basic block.

Furthermore, all changed global variables are copied to the basic blocks `DataFlow` property. All *sensitive globals* and *sensitive parameters* are traced backwards, starting from the current basic block where the function is called. The details of the taint analysis are explained in Section III-G.

F. Simulating Block Edges

Analog to the simulation of basic blocks, we simulate block edges when a basic block is connected to another basic block. In this simulation we try to identify sanitization by analyzing the condition of the jump, including 43 built-in functions. The following types of sanitization are recognized:

- **operators (6):** A symbol is sanitized within a basic block if it is compared to a static value in the entry edge. This is done with the `Equal` (`==`) or `Identical` (`===`) operator. The nodes `NotEqual` (`!=`) and `NotIdentical` (`!==`) only sanitize their arguments if they are sub-nodes of a `BooleanNot` node. Furthermore, a symbol is sanitized within a basic block if the entry edge requires the symbol to be `empty` or not set (`!isset`).

- **type checks (21):** A subset of built-in functions such as `is_numeric()` or `ctype_digit()` check if all characters of its argument are numerical and return *true* on success. Thus, they can be used to avoid the presence of malicious characters inside their argument for a branch.
- **file checks (11):** Some built-in functions such as `is_file()` or `stat()` check if a given argument is a valid file on the file system. If no functionality of uploading a file with an arbitrary file name is available, these functions effectively sanitize their argument (excluding file vulnerabilities).
- **whitelists (3):** The built-in functions `array_search()`, `array_key_exists()`, and `in_array()` are often used to check whether a value is in a given set of allowed values. The checked symbol is then added as sanitized symbol to the block edge. Another form of whitelisting is recognized when a specific array key is checked for presence (`isset($whitelist[$check])`).
- **regex (8):** A symbol can be checked for a range of characters with the help of regular expression patterns. These checks are performed with built-in functions such as `preg_match()` or `ereg()`. We transform the regular expression into an AST and check for every *or* branch if a configured set of characters can pass the expression. Each character is associated with different sanitization tags that are added to the target symbol of the regex function if the character cannot pass the regular expression. While our approach is not sound for all regular expressions, our evaluation has shown that most of the regular expressions used for sanitization are kept simple and no false positives or negatives were encountered.

Furthermore, user-defined functions can wrap the sanitization methods listed above. For example, the function `isValid()` in Listing 11 returns *true* if the argument is numerical and *false* otherwise.

```
1 function isValid($value) {
2   if(is_numeric($value)) {
3     return true;
4   }
5   return is_numeric($value); // false
6 }
```

Listing 11: Sanitization with a user-defined function.

To detect correct validation and to prevent false positives, the data flow through the function `isValid()` is analyzed. If the return symbol of a basic block is a constant with the value *true* or *false* (line 3), we check if the entry edge of the basic block sanitizes a symbol that at the same time is a parameter of the analyzed function. Then, a `Boolean` symbol is connected to the sanitized parameter and the constant value and added to the *returnValues* of the function. Furthermore, the built-in functions introduced previously in this section return a similar `Boolean` symbol. With the help of these `Boolean` return symbols, a user-defined function can be evaluated as sanitizing function during edge simulation.

G. *Taint Analysis*

When simulating a basic block, each function call is inspected for potential vulnerabilities. We identified 288 sensitive sinks in the PHP language which we configured by function name, sensitive parameter, and vulnerability type. For each called sensitive sink, a new taint analysis is invoked for the corresponding vulnerability type. RIPS is aware of 20 different vulnerability types that are listed in the following. These are refined to 45 different scopes, e. g., a *File Inclusion* is refined into a *Local* or *Remote File Inclusion*.

1) Code Execution
2) Command Execution
3) Connect Injection
4) Cross-Site Scripting
5) Denial of Service
6) Env. Manipulation
7) File Inclusion
8) File Upload
9) File Write
10) HTTP Resp. Splitting
11) LDAP Injection
12) Open Redirect
13) Path Traversal
14) Reflection Injection
15) Session Fixation
16) SQL Injection
17) Unserialize
18) Variable Tampering
19) XML/XXE Injection
20) XPath Injection

First, all possible strings that *flow* into the sensitive argument are reconstructed by backwards-directed data flow analysis. Furthermore, each string is inspected in a context-sensitive way for user input. If unsanitized user input was found and the markup context is exploitable, a new vulnerability is reported.

1) Data Flow Analysis: In order to find all possible values of a sensitive sink's argument, the argument (from now on referred to as *traceSymbol*) is traced backwards through all basic blocks linked as entry edge to the current basic block. In our implementation, we loop through all entry edges of the current basic block that do not sanitize the *traceSymbol* and look-up its name in the `DataFlow` property of each block summary. If a match is found, the *traceSymbol* is replaced with the mapped symbol and all sanitization tags and encoding types are copied. Then, the trace continues through all linked entry edges of the basic block. Finally, the unique sum of the return values for each path in the CFG are returned.

The algorithm stops if the *traceSymbol* maps to a static `Value` symbol or if the current basic block has no entry edges. If the *traceSymbol* is a variable or array access, it is checked if the *traceSymbol* is in the list of the 13 superglobal variables (refer to Section III-G3). It is also checked that the *traceSymbol* is of type *string*, that it is not encoded, and that it is not sanitized against the currently analyzed vulnerability type.

The sanitization status of a symbol is infered as follows: If a symbol is encoded (base64/hex/zlib...), it is sanitized against all vulnerability types. If a symbol is decoded and was previously encoded with the same encoding, its sanitization status depends on previously added sanitization tags. If a symbol is decoded without prior encoding, all sanitization tags are dropped because malicious characters can then be provided in an encoded way by an attacker.

If the *traceSymbol* maps to an unsanitized tainted source, the *traceSymbol* is saved and a linked user input tag is returned. Otherwise, if the *traceSymbol* maps to an unsanitized parameter or globaled variable of the user-defined function that the basic block is part of, a corresponding tag is returned. These tags can be analyzed context-sensitively later on.

To optimize this time intense process, we implemented caching of the result for each basic block. When a symbol is traced through a basic block, the result is stored in a basic block cache. If the same symbol is traced through this basic block again, the result is already available in the cache and the trace can be aborted. This drastically improves the performance of our analysis engine.

A configurable *maximum* amount of traversed edges introduces a path limit to optimize performance. This can lead to false negatives, but we did not encounter them in practice. Furthermore, step-by-step caching of lookup results for each basic block raises the chance that a full path analysis is not required if parts of the analysis were analyzed previously.

2) Context-Sensitive String Analysis: The obtained strings from the data flow analysis are analyzed for user input tags. For each vulnerability type, a different analyzer is invoked that identifies the context within the markup. Depending on the context, specific vulnerability tags are determined. Only if the *taint symbols* are not sanitized against the current vulnerability tag, they are marked as a tainted symbol and a vulnerability is issued.

If no user input was found, but the analyzed sensitive sink is called within a user-defined function, the strings are analyzed for parameter and global tags. When these are found in one of the strings, the corresponding symbols are added as vulnerable parameters or as vulnerable global variables to the user-defined function summary. During inter-procedural analysis these symbols are analyzed starting from the basic block of the function call. In the following, we explain the analysis of two markups. Further *markups* such as HTTP headers or file names require unique analysis, but are less complex.

a) HTML: For each XSS vulnerability, we inspect the HTML markup of the reconstructed string. The HTML markup is resolved from previous basic blocks similar to the techniques described by Minamide [25] and used by Wasserman and Su [41]. Each reconstructed string is parsed with an HTML parser into a structured HTML DOM tree. First, the text between two HTML elements is searched for user input tags. On success, the vulnerability tag is changed to `XSS_ELEMENT`. If the HTML tag name is *script* or *style*, the vulnerability tag is changed to `XSS_SCRIPT` or `XSS_STYLE` accordingly. While sanitization of meta characters within a normal HTML element is sufficient, it does not prevent attacks when these characters are injected into a `script` or `style` tag.

If the HTML element has attributes, each attribute's value is searched for user input tags. Depending on how the attribute value is quoted, the vulnerability tag `XSS_ATTR_DQ`, `XSS_ATTR_SQ`, or `XSS_ATTR_NQ` is set for a **d**ouble, **s**ingle, or **n**ot-**q**uoted value. As discussed in Section II-A2, it is also important to consider the type of the HTML attribute. A list of 49 eventhandler and 21 url attributes is configured to set the special tag `XSS_ATTR_JS` or `XSS_ATTR_URL`.

b) SQL: As described in Section II-A1, a SQLi vulnerability is also context-sensitive. Our SQL parser tries to determine if the injection happens between a **s**ingle **q**uoted value (`SQLI_SQ`), **d**ouble **q**uoted value (`SQLI_DQ`), or is embedded into the SQL query unquoted (`SQLI_NQ`). The special tag `SQLI_MI` (**m**ultiple **i**nput) is reserved for symbols that are sanitized by escaping quotes but were passed through a *substring* built-in function afterwards. This can lead to a SQL injection vulnerability if the substring reveals a trailing backslash and more than one tainted source flows into the sensitive sink.

3) Source Analysis: For further improvement, we analyze the tainted source depending on the vulnerability type. For example, client-side vulnerabilities such as *Session Fixation* or *HTTP Response Splitting* require an easy to forge source for practical attacks. Thus, tainted values originating from uploaded file names (`$_FILES`), cookies (`$_COOKIE`), or HTTP headers (`$_SERVER['HTTP_*']`) are ignored. While all HTTP headers stored in the superglobal `$_SERVER` array can be altered by the user arbitrarily, there are several CGI parameters that disallow certain characters and are not practical for exploitation of certain vulnerability types. The `$_SERVER` key's limitations are:

- `HTTP_HOST`: A slash or a backslash within the *Host* header is disallowed and will result in a bad request blocked by the web server. Thus, the *Host* header cannot be used for *Path Traversal* attacks.
- `PHP_SELF`, `PATH_INFO`: A *Path Traversal* attack within the requested path will result in a *Path Traversal* attack against the web server and will most likely fail.
- `PHP_SELF`, `PATH_INFO`, `REQUEST_URI`: The requested path and URI contains the current path as prefix. Consequently, these keys cannot be used to inject protocol handlers to an URL attribute or to exploit a *Remote File Inclusion* vulnerability because both attacks require the control of the first injected characters.

Note, that the source `$_SERVER['QUERY_STRING']` and `$_SERVER['REQUEST_URI']` are not listed with further limitations. Although browsers such as *FireFox* and *Chrome* automatically urlencode meta characters within the query string, *Internet Explorer* does not. Furthermore, these sources can be tainted arbitrarily by manually crafting an HTTP request. Our tool is also aware that `$_GET`, `$_POST`, and `$_COOKIE` parameters can be supplied as arrays by the user. Hence, we not only mark all parameter values as tainted, but also all available key names.

4) Environment-aware Analysis: A PHP application and its vulnerabilities may behave differently depending on the PHP configuration. For this purpose, our tool can be configured with four different PHP settings and a PHP version number. The version number is important to categorize certain file-based vulnerabilities that base on null-byte injections or *HTTP Response Splitting* attacks that were fixed by the PHP developers. Furthermore, the PHP settings *magic_quotes_gpc*, *allow_url_fopen*, and *allow_url_include* may restrict certain vulnerabilities, while the PHP setting *register_globals* may introduce certain vulnerabilities. Our tool also aims to detect re-implementations of these settings, for example, when sanitization is applied to the superglobals or the built-in function `extract()` is used.

TABLE I: Evaluation results for popular real-world applications.

Software	Files	LOC	TA	TBC	TBI	UBC	UBI	MP	ST	TP	FP	FN	CVE
HotCRP	72	39 938	19 420	5 171	289	170	51	293	55	7	4	0	0
MyBB	327	138 357	55 917	8 152	1 287	225	115	1 117	188	2	0	8	10
osCommerce	545	65 556	7 453	9 059	860	184	85	476	60	48	19	1	29
phpBB2	176	46 287	10 623	3 666	340	144	56	289	29	13	6	1	2
phpBB3	270	186 814	43 616	7 554	1 273	269	192	1 143	252	3	0	0	1
Total	1 390	476 952	137 029	33 602	4 049	676	294	3 318	584	73	29	10	42
Average	278	95 390	27 406	89%	11%	70%	30%	664	117	72%	28%	24%	8

Type	TP	FP	FN
XSS	48	22	5
File Write	8	0	0
Path Traversal	3	0	0
CRLF Injection	1	0	1
SQL Injection	11	7	4
Var Tampering	2	0	0
Total	73	29	10

IV. Evaluation

The benefit of a static code analysis tool is measured by its capabilities to model the programming language, analyze the model correctly, and detect different types of vulnerabilities precisely. The true positive (TP), false positive (FP), and false negative (FN) rates indicate how well a tool performs. We evaluated the precision of our tool RIPS with five popular open source applications: *HotCRP 2.60* [22], *MyBB 1.6.10* [27], *osCommerce 2.3.3* [30], *phpBB2 2.0.23*, and *phpBB3 3.0.11* [32]. A high-level overview of the evaluation results and the vulnerability distribution among the different types is listed in Table I. We counted the overall number of analyzed PHP files and the lines of code (LOC) for each application. Furthermore, we counted the number of sensitive sinks that required taint analysis (TA). The maximum memory peak (MP) is denoted in megabytes and the overall scan time (ST) is denoted in seconds. All issues were reported to the vendors in a responsible way.

In the following, we evaluate the built-in function coverage of our tool and its performance. We then highlight the most interesting true positive findings and discuss our false positives. To evaluate our false negative rate, we scanned old versions of the selected applications that are affected by known vulnerabilities. Such an evaluation approach enables us to estimate which vulnerabilities RIPS was not able to find in an automated way. Finally, we analyzed two PHP projects that were evaluated by other researchers working in this area to directly compare RIPS against other tools.

A. Performance

Our testing environment was equipped with an Intel i7-2600 CPU with 3.4 GHz and 16 GB of memory. We scanned a total of 1 390 PHP files with almost half a million LOC. On average, every 4th line of code contained a sensitive sink that required taint analysis. The average memory peak usage per project was 664 MB and the average scan time was about 2 minutes, or in other words, RIPS needed 7 MB and 1.23 seconds per KLOC. The largest evaluated software *phpBB3* with over 186 000 LOC had a scan time of less than 5 minutes and required a bit more than 1 GB of memory. Thus, we are positive that our approach scales to even larger projects.

B. Built-in Function Coverage

To evaluate the built-in function coverage of our tool, we logged the name of every called function. If the function was not declared within the software, it was considered as a built-in function. We then examined if the function name is covered by our tool or was ignored during analysis. Table I shows the *total* number of built-in functions our tool covered (TBC) and ignored (TBI). Furthermore, we counted the *unique* number of built-in functions our tool covered (UBC) and ignored (UBI).

On average, every 13th line of code contains a built-in function call, excluding the lines that call user-defined functions (which can lead to further built-in function calls). The high amount of built-in function usage emphasizes the need for precise function simulation during code analysis.

Within the five analyzed applications, a call to 970 *unique* built-in functions was detected. Our tool simulates 70% of these unique functions, which covers 89% of all defined calls within the applications. The remaining calls ignored by our tool are mainly related to database, image, and sort functions and — to the best of our knowledge — these functions do not affect the analysis results.

C. True Positives

In total, 72% of the reported issues in our evaluation are true positives. A true positive was counted for every vulnerable line of code. This means that a vulnerability inside a function was counted only once if the function was called in an exploitable context and not for every call. Sometimes, a valid report was counted even if the vulnerability is not exploitable. For example, if the same input is used in two differently constructed SQL queries but the application exits after a SQL query fails, and it is not possible to craft an injection that fits both SQL queries, two valid reports were counted nonetheless. In this case, fixing only the first SQL query would allow to exploit the second SQL query and thus both reports are important.

We now examine selected vulnerabilities in three different projects to illustrate their complexity and severity. It is evident that these vulnerabilities could only be detected with our novel approach of precisely simulating different language features and their interaction.

1) phpBB2: phpBB is a well-known open source bulletin board software [32]. It is developed in the current version *phpBB3*, however, its predecessor *phpBB2* is still widely used and also integrated into popular software like *PHP-Nuke* [31]. In total, RIPS reported 13 vulnerabilities in the latest *phpBB2* version 2.0.23. The vulnerabilities also affect the latest *PHP-Nuke* version.

Our tool detected six rather harmless SQL injection vulnerabilities in the installer based on a user-supplied database table prefix. Additionally, two critical SQL injection vulnerabilities in the administration interface were detected. The simplified code of one of these SQL injections is shown in Listing 12.

```
1  $style_name = urldecode($_GET['style']);
2  $install_to = urldecode($_GET['install_to']);
3  $template_name = $$install_to;
4  for($i = 0; $i < count($template_name); $i++) {
5    if($template_name[$i]['style_name'] == $style_name) {
6      while(list($key, $val) = each($template_name[$i])) {
7        $db_fields[] = $key;
8        $db_values[] = addslashes($val);
9      }
10   }
11 }
12 $sql = "INSERT INTO " . THEMES_TABLE . " (";
13 $sql .= implode(',', $db_fields);
14 $sql .= ") VALUES (";
15 $sql .= "'" . implode("','", $db_values) . "'";
16 $sql .= ")";
17 mysql_query($sql);
```

Listing 12: Simplified code of a SQL injection in *phpBB2*.

In line 3, a *variable* variable based on unsanitized user input is assigned to the variable `$template_name`. The application assumes that `$template_name` is an array that stores several templates. First, it loops through all elements of `$template_name` and compares the `style_name` of the template with the provided GET parameter *style*. If the specified template was found, the application saves the template's array key names to the array `$db_fields` and all array values sanitized to the array `$db_values`. Then, all `$db_fields` are used as column identifiers in a SQL *INSERT* query and all `$db_values` are used as values to insert within quotes. A vulnerability occurs because the `$db_fields` are not sanitized but can be influenced by an attacker. For exploitation, the *install_to* parameter is set to *_GET* such that the variable `$template_name` points to the GET parameters controlled by the user. Then, the SQL injection can be exploited as shown in Listing 13. The vulnerability is not present in the *phpBB3* code base.

```
admin_styles.php?style=rips&install_to=_GET&0[style_name]=
        rips&0[template_name)VALUES('sqli','sqli')-- -]=1
```

Listing 13: SQL injection exploitation through an array key.

The rather complicated code demonstrates the importance of simulating PHP's built-in features. First of all, the data flow through several built-in functions such as `urldecode()`, `list()`, `each()`, and `implode()` has to be analyzed precisely. The challenge is to model the array handling of these functions. If one of these functions is not simulated or simulated imprecisely, the vulnerability is not detected. Moreover, we encountered a *variable* variable and a *while* loop that requires analysis of *variable* elements. Last but not least, sanitization is applied in line 8 but not in line 7 and the SQL query requires context-sensitive string analysis to decide whether the sanitization is sufficient or not.

2) HotCRP: HotCRP is a popular conference management software that is used by several top tier conferences. Our current prototype reported 7 XSS and 4 SQLi vulnerabilities in the latest version 2.60. Six of seven reported XSS vulnerabilities reflect a user supplied parameter unsanitized to the HTML response and are true positives.

One out of four reported SQLi vulnerabilities is a true positive. It affects an `INSERT` query when a new paper is added by an unprivileged user. Because MySQL error reporting is enabled and the user passwords are stored in plaintext in the database, an attacker can easily read the conference administrator login credentials (see Figure 1). This enables an attacker to take the conference administration account over and to review, edit, delete, or accept submitted papers of her choice.

```
1 $v = defval($_REQUEST, "emailNote", "");
2 echo "<input type='text' name='emailNote' size='30' value
       ='",
3   htmlspecialchars($v=="" ? "Optional explanation" : $v),
4   "' />";
```

Listing 14: Weak sanitization in *HotCRP*.

An XSS vulnerability shown in Listing 14 demonstrates our ability to detect weak sanitization. The user-defined function `defval()` returns user input that is embedded to the HTML page. The user input is sanitized with the built-in function `htmlspecialchars()` in line 3, however, the second parameter is not set to escape single quotes (see Section II-A2). Previous work would miss this vulnerability because `htmlspecialchars()` is handled context-insensitive as valid sanitization method.

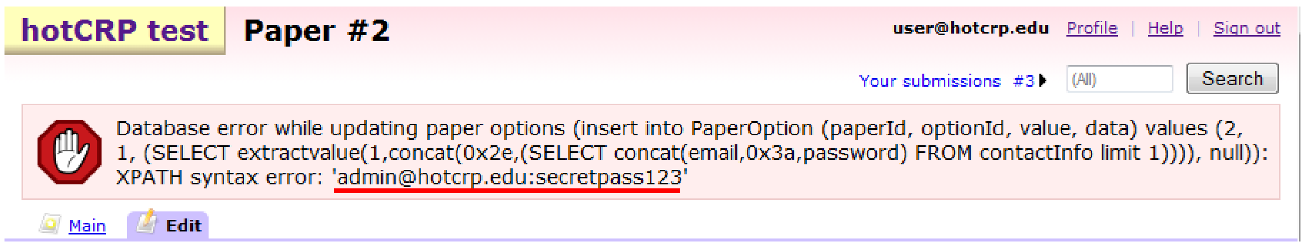
3) osCommerce: osCommerce is a popular online store software that allows to sell products and services. We were able to identify 48 vulnerabilities in the latest version 2.3.3.

RIPS detected a SQL injection vulnerability in the installer and in the administration interface. Combined with an XSS vulnerability, the second SQLi vulnerability allows an unprivileged attacker to retrieve the administrator's password hash by sending a malicious link to an administrator.

Additionally, 40 XSS vulnerabilities were detected in the administrator interface and in the installer. The root cause is shown in Listing 15.

```
$HTTP_GET_VARS = array_map('addslashes', $_GET);
echo '<tr onclick="document.location.href=\'' . BASE_URL.
'page=' . str_replace('&amp;', '&', htmlspecialchars(
     $HTTP_GET_VARS['page'])) . '\'">';
```

Listing 15: An XSS vulnerability in eventhandler context.



Database error while updating paper options (insert into PaperOption (paperId, optionId, value, data) values (2, 1, (SELECT extractvalue(1,concat(0x2e,(SELECT concat(email,0x3a,password) FROM contactInfo limit 1))), null)): XPATH syntax error: 'admin@hotcrp.edu:secretpass123'

Fig. 1: SQL injection in HotCRP leaks plaintext administrator password to unprivileged user.

The GET parameter *page* is used in the eventhandler `onclick` of a table row. First, it is not possible to break out of the outer double quotes of the eventhandler because `htmlspecialchars()` is used. Second, although the parameter `ENT_QUOTES` is not set to encode single quotes, it is not possible to break out of the inner single quotes in the Javascript code because *osCommerce* uses the function `addslashes()` for each user supplied parameter that adds a preceding backslash to each single quote. Third, it is not possible to inject a `javascript:` protocol handler because the constant `BASE_URL` is used as prefix of the new location.

However, our tool validly reported an XSS vulnerability, because the injection context is an eventhandler. Here, the browser interprets HTML entities within the Javascript code as their original character representation. Thus, we are able to inject the HTML entity `&#39;` to break out of the inner single quotes and inject our own Javascript code. The attack is shown in Listing 16 with an urlencoded payload that is triggered if the user clicks on the table row. Again, the vulnerability demonstrates the importance of context-sensitive string analysis and the correct handling of PHP's built-in functions.

```
admin/customers.php?page=%26%2339%3B-alert(1)-%26%2339%3B
```

Listing 16: Urlencoded payload for XSS exploitation.

Additionally, our tool reported various file vulnerabilities in the installer and the administration interface, for example, a *File Write* vulnerability that allows to write arbitrary PHP code into language files leading to *Remote Code Execution*. Although we counted these as valid reports, the affected code represents a feature and is not interpreted as a security vulnerability by the developers.

Furthermore, an XSS is reported for each SQL query that contains user input and could result in an error, because the SQL query is printed unsanitized in the error handler. Although these reports are valid and were successfully verified with the present SQL injection vulnerability, we ignored them and did not counted them as true positive. Contrarily, they could be counted as false positive, because the SQL queries do not fail in typical situations.

D. False Positives

In total, 28% of the reported vulnerabilities in our evaluation turned out to be false positives. The root causes for these invalid reports are:

- Path-insensitive data flow analysis
- Sanitization through database whitelist
- Wrong content-type

The root cause for 19 false positives in *osCommerce* is shown in Listing 17. Here, a user-defined function sanitizes its first parameter based on the second argument. Because RIPS performs path-insensitive data flow analysis and is based on function summaries, it wrongly integrates both possible `return` values into the function summary that is then used regardless of the second argument.

```
function tep_output_string($string, $protected = false) {
  if ($protected == true) {
    return htmlspecialchars($string);
  } else {
    return $string;
  }
}
```

Listing 17: The root cause for false positives in *osCommerce*.

In *HotCRP* an XSS vulnerability was reported erroneously. Here, a user-supplied email address is printed to the HTML page unsanitized, however, the email address is checked for presence in the database first. Because the format is checked before a new email address is added to the database, the email address is sanitized indirectly. Furthermore, three reported SQLi vulnerabilities are false positives. Our prototype was unable to detect path-sensitive sanitization of tainted values [11].

Another reported XSS vulnerability was counted as false positive. Although user input is printed unsanitized to the HTML page, the vulnerability is not exploitable because the HTML response header `content-type` is changed to `text/plain`. Thus, a browser will not render injected HTML and prevent XSS attacks.

E. False Negatives

Evaluating false negatives is a difficult task because the number of existing vulnerabilities in a software is unknown. To obtain an estimated result, we collected all CVE entries related to injection flaws in the five selected PHP projects from the CVE database [26]. We then run our tool against the affected versions of the software and searched for a vulnerability report that matches the CVE details. During this process, we encountered the following obstacles. First of all, no CVE entries exist for *HotCRP*. Second, only very few CVE entries for *phpBB* are relevant, because most of them describe vulnerabilities in external plugins. For *MyBB* and *osCommerce* a fair amount of CVE entries is available, but certain old versions of *MyBB* are unavailable on the Internet.

In total, we examined 42 CVE entries in 7 different software versions. Our tool correctly identified 32 of the described vulnerabilities in an automated way, resulting in an estimated false negative rate of 24%. However, if we exclude *MyBB*, the false negative rate is only 6%. The root causes for false negatives are:

- field-insensitive data flow analysis
- second-order vulnerabilities

Our tool misses 8 out of 10 vulnerabilities in *MyBB* because it does not fully support analysis of object-oriented code yet and is field-insensitive. All false negatives in *MyBB* are based on the same problem: RIPS misses the data flow of GET and POST parameters because they are written to and retrieved from a class object. Two other false negatives stem from the fact that our tool does not handle the data flow through externally stored data like in databases. Thus, it misses second-order vulnerabilities such as *Persistent XSS*.

TABLE II: Compared evaluation results for previously studied real-world applications.

						RIPS				Jovanovic et al.				Xie & Aiken	
						XSS		SQLi		XSS		SQLi		SQLi	
Software		Files	LOC	TB	UB	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
NewsPro	1.1.4	23	5 047	827	56	5	0	18	0	4	14	14	34	8	0
NewsPro	1.1.5	23	5 077	841	57	4	0	6	0	-	-	-	-	-	-
myBloggie	2.1.3b	91	11 487	1 218	122	15	0	26	3	13	3	31	11	16	0
myBloggie	2.1.4	92	11 772	1 235	124	13	0	8	0	-	-	-	-	-	-
Total		229	33 383	4121	134	37	0	58	3	17	17	45	45	24	0
Average		57	8 346	1030	90	100%	0%	95%	5%	50%	50%	50%	50%	100%	0%

F. Comparison

In previous work on this topic, several evaluation results for different software applications were reported [20, 41, 43]. Comparing our results to previous work is not straight forward for several reasons. First of all, often we have no access to the implemented prototype. Second, we do not know exactly how the amount of detected vulnerabilities was counted, and as discussed in Section IV-C this is a hard problem itself. As a result, comparing only the numbers of found true and false positives may be misleading.

For a better approach, we chose to evaluate software that was analyzed by other researchers with the following criteria: (1) the software is still available on the Internet, (2) there is a follow-up version, and (3) the follow-up version introduces security patches and does not add new main features. We can then compare our results to the stated results in previous work for the exact software version, but more importantly, we can assume that any vulnerability we detect in the follow-up version was missed by previous work.

The software *NewsPro* [38] and *myBloggie* [28] match our criterias and was evaluated by Jovanovic et al. [20] and by the work of Xie and Aiken [43]. Pixy supports the detection of XSS and SQLi vulnerabilities, while the prototype of Xie and Aiken only detects SQLi vulnerabilities. Our results compared to the others are listed in Table II. The total TP rate of 98% and FP rate of 2% stem from the fact that the code of both software is relatively small and simple compared to our selected real-world applications. This is shown by the LOC as well as the total (TB) and unique (UB) amount of built-in functions used.

1) NewsPro: Utopia NewsPro is a news management system and we evaluated version *1.1.4* and the follow-up version *1.1.5*. RIPS reported 5 XSS vulnerabilities in version *1.1.4* whereas Pixy reported 4 XSS vulnerabilities. According to the CVE details [26], the XSS vulnerabilities reported by Pixy mainly base on the deprecated *register_globals* setting, which is disabled for our prototype by default. Because the follow-up version *1.1.5* contains 4 of our 5 XSS vulnerabilities, we conclude that Pixy missed these issues. Furthermore, Pixy had a false positive rate of 77% while RIPS reported no false positives in *NewsPro*.

Additionally, our tool reported 18 SQLi vulnerabilities with no false positives. The prototype of Xie and Aiken reported only 8 SQL injections and seem to miss certain vulnerabilities. Even with *register_globals* enabled, which introduces far more security issues, Pixy detected only 14 SQLi vulnerabilities with a false positive rate of 71%. We believe that RIPS detected more SQL injections compared to prior work, aided by the fact that 6 of our detected SQL injections are still present in the follow-up version *1.1.5*.

2) myBloggie: We evaluated the weblog system *myBloggie 2.1.3 beta* and its follow-up version *myBloggie 2.1.4*. According to the authors, Pixy reported 13 XSS vulnerabilities in *myBloggie 2.1.3 beta*. Because 13 of our 15 detected XSS vulnerabilities in *myBloggie 2.1.3 beta* are still present in the follow-up version, we assume that we identified different vulnerabilities than Pixy. A closer look at the released advisory reveals that Pixy reported issues based on the deprecated PHP setting *register_globals* [17]. Our vulnerabilities base on the source `$_SERVER['PHP_SELF']` which is not modeled by Pixy. In Section 6.2 of their work it is wrongly stated that "the predefined PHP variables `$_SERVER['PHP_SELF']` and `$_SERVER['HTTP_HOST']` are untainted, since they cannot be controlled by an attacker" [19]. Pixy encountered 3 false positives whereas our prototype only reported true positive XSS vulnerabilities.

Furthermore, RIPS reported 26 SQLi vulnerabilities. Three false positives occured due to path-sensitive sanitization. Because the prototype of Xie and Aiken is path-insensitive as well but did not encounter these false positives, we conclude that our prototype analyzed more data flow. Their prototype reported only 16 SQLi vulnerabilities which supports this assumption.

Pixy detected 31 SQLi vulnerabilities in *myBloggie*. Because 8 of our 26 detected SQLi flaws are still present in the follow-up version, we approximate that Pixy detected only 18 of the 26 SQLi vulnerabilities. Another 13 SQLi vulnerabilities probably base on the *register_globals* setting. Furthermore, 36% of the SQLi vulnerabilities in *myBloggie* reported by Pixy are false positives.

Finally, we do not know if one or more vulnerabilities was detected by Pixy or the prototype of Xie and Aiken but was missed by RIPS. A rather complicated XSS vulnerability in *myBloggie 2.1.3 beta* described in detail by Jovanovic et al. [20] was detected by our tool.

V. Related Work

Due to its widespread usage, the PHP language has received a lot of attention over the last years. In the following, we discuss work related to (mainly static) taint analysis of PHP code and clarify how RIPS advances the state-of-the-art. Furthermore, prior work on string analysis is also related to our approach and we discuss this area as well.

a) Taint Analysis: Huang et al. developed a static code analysis tool for PHP called *WebSSARI* [14] based on a CQual-like [7, 8] type system. Compared to RIPS, it has certain limitations. First of all, it works only intra-procedural and not inter-procedural. That means that *WebSSARI* is able to handle the program flow through user-defined functions, but it does not consider the context from where the function is called. Second, it does not handle dynamic features of PHP such as dynamic arrays or dynamic includes, which implies that many vulnerabilities will be missed for large PHP applications. In a follow-up paper, Huang et al. presented a related approach based on bounded model checking [15] that has similar limitations.

Xie and Aiken presented a static analysis algorithm for detecting SQL injection vulnerabilities in PHP applications using block and function summaries [43]. It is both intra-procedural and inter-procedural and handles more dynamic features of PHP. However, we found several limitations in their approach. For example, the *include* operator is described as statement, but can be used as an expression in PHP. Thus, merging CFGs if the *include* operator appears within an expression is error prone. A growing CFG can also lead to path explosion, thus we propose another approach. Further, their implementation only supports SQLi vulnerabilities in a context-insensitive way and does not model built-in functions. In contrast, RIPS covers a variety of vulnerabilities context-sensitive and precisely models built-in function, two features that enable us to detect new vulnerabilities in complex PHP applications. The direct comparison of RIPS with the approach by Xie and Aiken discussed in Section IV-F demonstrates that our approach outperforms their method.

Jovanovic et al. developed *Pixy*, an open source, static code analyzer for PHP written in Java [18, 20]. A lot of work has been put into modeling aliases which are supported by our tool only in a limited way. However, we found only very few recent PHP applications actually using aliases and only in a rather simple manner. The down-side of *Pixy* is that it only supports Cross-Site Scripting and SQL injection vulnerabilities and has only 29 built-in functions configured leading to false negatives as demonstrated in the previous section. False positives occur due to missing or imprecise modeling of built-in functions and markup context analysis.

An extended version of *Pixy* called *Saner* was created by Balzarotti et al. [2] to improve the detection of user-defined sanitization. It uses manually created, predefined test-cases to check sanitization routines with dynamic analysis. *Apollo* [1] combines symbolic and concrete execution techniques together with explicit-state model checking. The authors tested their tool with *phpBB2* version 2.0.21 and detected several vulnerabilities. RIPS detected 13 novel vulnerabilities in version 2.0.23 of *phpBB2*, indicating that our approach is capable of discovering flaws not identified by *Apollo*.

b) String Analysis: Yu et al. also extended *Pixy* to perform automata-based string analysis for checking the correctness of sanitization routines without dynamic analysis [44]. The drawback of *STRANGER* and *Saner* is that they are only as good as their test-cases. Even if all configured attack patterns for one vulnerability type are filtered correctly, other attack patterns could exist that bypass the sanitization undetected.

Wasserman and Su [41] presented an efficient approach to detect SQL injection vulnerabilities. The key idea is to first generate an approximation of the query strings a program may generate using a context free grammar, and then analyze if all potential strings are safe with the help of information flow analysis. In another paper, Wasserman and Su [42] presented a static code analysis approach to detect XSS vulnerabilities caused by weak or absent input validation. In this work, the authors combine work on taint-based information flow analysis with string analysis.

Christensen et al. [4] were one of the first to apply string analysis to detect SQL injection vulnerabilities by analyzing reflective code in Java programs. Minamide extended these techniques and presented a string analysis technique for PHP applications that over-approximates the HTML output of the application [25]. The goal of this work was to detect XSS vulnerabilities, but it suffered from the over-approximation and resulting false positives. In contrast, RIPS performs a more precise analysis and we successfully detected several kinds of XSS vulnerabilities.

VI. Conclusion

Web applications are the interfaces between users and web servers that handle critical data such as credit card numbers, email addresses, and passwords. An attacker can abuse security vulnerabilities in these applications to access and steal this sensitive data. To avoid attacks, the source code has to be analyzed to identify security vulnerabilities.

In this paper, we presented a novel static code analysis tool called RIPS that is able to automatically and accurately detect taint-style vulnerabilities in PHP applications. It models the dynamic PHP language precisely by using state-of-the-art code analysis techniques and it detects 45 different scopes in 20 different vulnerability types with fine-grained context-sensitive taint analysis. An evaluation showed that current vulnerabilities in popular applications are based on complex PHP features and existing tools are not capable of finding such vulnerabilities due to their incompleteness in modeling the PHP language. In contrast, our tool was able to detect 73 previously unknown vulnerabilities with a low false positive rate of 28% in large applications with over 185 000 LOC.

Future work will address the false positives reported by RIPS by analyzing path-sensitive data flow and sanitization [11]. The true positive rate can be increased by adding support for aliases and extending our limited support of OOP code. Also, the detection of more vulnerability types such as *second order* vulnerabilities will be addressed in the future.

References

[1] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Softw. Eng.*, 36(4), 2010.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, 2008.

[3] P. Biggar and D. Gregg. Static Analysis of Dynamic Scripting Languages. 2009.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *International Conference on Static Analysis*, 2003.

[5] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[6] M. Egele, M. Szydlowski, E. Kirda, and C. Kruegel. Using Static Program Analysis to Aid Intrusion Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2006.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. *SIGPLAN Not.*, 34(5), May 1999.

[8] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive Type Qualifiers. *SIGPLAN Not.*, 37(5), May 2002.

[9] T. P. Group. PHP: Manual Quick Reference. http://php.net/quickref.php, as of July 2013.

[10] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.

[11] D. Hauzar and J. Kofron. On Security Analysis of PHP Web Applications. In *IEEE Workshop on Security, Trust, and Privacy for Software Applications (STPSA)*, 2012.

[12] M. Hills, P. Klint, and J. Vinju. An Empirical Study of PHP Feature Usage. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

[13] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*, 2011.

[14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on the World Wide Web (WWW)*, 2004.

[15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Verifying Web Applications Using Bounded Model Checking. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

[16] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *International Conference on the World Wide Web (WWW)*, 2007.

[17] N. Jovanovic. TUVSA-0603-002 - MyBloggie: Multiple XSS Vulnerabilities. http://www.iseclab.org/advisories/TUVSA-0603-002.txt, as of July 2013.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Workshop on Programming Languages and Analysis for Security*, 2006.

[20] N. Jovanovic, C. Kruegel, and E. Kirda. Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications. *Journal of Computer Security, Vol 18, N5, August 2010*, 08 2010.

[21] A. Klein. Cross-Site Scripting Explained. *Sanctum White Paper*, 2002.

[22] E. Kohler. HotCRP Conference Management Software. http://www.read.seas.harvard.edu/~kohler/hotcrp/, as of July 2013.

[23] J. A. Kupsch and B. P. Miller. Manual vs. Automated Vulnerability Assessment: A Case Study. *International Workshop on Managing Insider Security Threats*, 2009.

[24] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.

[25] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *International Conference on the World Wide Web (WWW)*, 2005.

[26] MITRE. Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/, as of July 2013.

[27] MyBB. Open Source Discussion Board. http://www.mybb.com/, as of July 2013.

[28] myWebland Group. myBloggie Weblog System. http://mybloggie.mywebland.com/, as of July 2013.

[29] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[30] osCommerce. Creating Online Stores Worldwide. http://www.oscommerce.com/, as of July 2013.

[31] PHP-Nuke. CMS Portal Solution. http://www.phpnuke.org/, as of July 2013.

[32] phpBB. Free and Open Source Forum Software. http://www.phpbb.com/, as of July 2013.

[33] D. Ray and J. Ligatti. Defining Code-Injection Attacks. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2012.

[34] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *ACM Symposium On Applied Computing (SAC)*, 2012.

[35] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[36] S. Son and V. Shmatikov. SaferPHP: Finding Semantic Vulnerabilities in PHP Applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2011.

[37] S. Thomas, L. Williams, and T. Xie. On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities. *Information and Software Technology*, 51(3):589–598, 2009.

[38] UtopiaSoft. Utopia News Pro. http://www.utopiasoftware.net/newspro/, as of July 2013.

[39] W3Techs. Usage of Content Management Systems for Websites. http://w3techs.com/technologies/overview/content_management/all, as of July 2013.

[40] W3Techs. Usage of Server-side Programming Languages for Websites. http://w3techs.com/technologies/overview/programming_language/all, as of December 2013.

[41] G. Wasserman and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[42] G. Wasserman and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering*, 2008.

[43] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, 2006.

[44] F. Yu, M. Alkhalaf, and T. Bultan. STRANGER: An Automata-based String Analysis Tool for PHP. In *Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.