Towards Web Application Security by Automated Code Correction

Ricardo Morgado, Ibéria Medeiros and Nuno Neves
LASIGE, Faculty of Sciences, University of Lisboa, Portugal
ricardo.morgado.12@gmail.com, imedeiros@di.fc.ul.pt, nuno@di.fc.ul.pt

Keywords:
Web application vulnerabilities, static analysis, code correction, software security.

Abstract:
Web applications are commonly used to provide access to the services and resources offered by companies. However, they are known to contain vulnerabilities in their source code, which, when exploited, can cause serious damage to organizations, such as the theft of millions of user credentials. For this reason, it is crucial to protect critical services, such as health care and financial services, with safe web applications. Often, vulnerabilities are left in the source code unintentionally by programmers because they have insufficient knowledge on how to write secure code. For example, developers many times employ sanitization functions of the programming language, believing that they will defend their applications. However, some of those functions do not invalidate all attacks, leaving applications still vulnerable. This paper presents an approach and a tool capable of automatically correcting web applications from relevant classes of vulnerabilities (XSS and SQL Injection). The tool was evaluated with both benchmark test cases and real code, and the results are very encouraging. They show that the tool can insert safe and right corrections while maintaining the original behavior of the web applications in the vast majority of the cases.

1 Introduction

In recent years, web applications have become increasingly popular and an essential part of our lives. Often, web applications require users to provide personal details, such as our home address and credit card number. This data is stored on the application's database and may be exposed if someone with malicious intentions can successfully perform an attack. Therefore, web applications are an appealing target for attackers because, if they succeed, they can potentially compromise the personal details of up to millions of users. Injection vulnerabilities, such as SQL Injection (SQLi) and Cross-Site Scripting (XSS), rank high in the list of web application security risks according to the OWASP Top 10 (van der Stock et al., 2017) and are the most prevalent and preferred of attackers.

Vulnerabilities are usually caused by misinformed developers who make mistakes when writing the code and do not possess sufficient knowledge on software security. Simultaneously, there are also the limitations of time and budget for testing within organizations, which contribute to exacerbate the problem. Lastly, the sources for programming information available to developers can sometimes have confusing recommendations, which can also influence the security of the code (Acar et al., 2016) (Fischer et al., 2017). The consequence is a growing number of vulnerabilities being reported every year, with a particular incidence on web applications.

In our context, PHP is particularly relevant, as it is the most used server-side language of web applications, powering around 79% of the websites[1]. The fact that PHP is a "weakly-typed" language makes it easier to introduce mistakes in some situations, especially when dealing with badly-documented code. All programming languages, including PHP, contain a wide range of functions (and other methods) that can be used to invalidate attacks. However, most developers do not master when and how to use them, conse-

[1]https://w3techs.com/technologies/details/pl-php/all/all

quently leaving applications with vulnerabilities.

There are several tools available to analyze PHP source code and to find potential bugs. Such tools are often hard to use, do not provide the information developers need, and report vulnerabilities that do not exist (i.e., false positives). Some tools are based on taint analysis, while others employ techniques like dynamic analysis (Schwartz et al., 2010) and symbolic execution (Zheng et al., 2013) (Huang et al., 2019). They provide reports in various formats and almost all of them leave the burden of fixing the bugs to developers. Given that most developers are unaware of the right way to remove a bug, this process many times does not completely eliminate the vulnerability.

To alleviate this problem, tools could detect and correct such bugs. However, the small number of tools that perform automatic correction of the source code, often have limitations in the sense that they insert unsound fixes, producing syntactically invalid new programs that can not be executed (Medeiros et al., 2014).

This work presents an approach to automatically repair web applications by employing a mixture of taint tracking and instruction simulation of PHP programs. The solution focuses on two prevalent types of web application vulnerabilities, namely XSS and SQLi. Our tool called PHPCorrector, determines where and what correction is most appropriate for a particular bug, and is able to deal with existing forms of sanitization. An experimental evaluation was performed with benchmark test cases from the NIST SARD dataset and six large web applications. The results demonstrate that PHPCorrector can repair appropriately the great majority of the identified bugs, leaving a few cases where the applications became only partially protected. We believe that these results are highly encouraging, giving evidence that our approach is an useful step towards automatic correction of web applications.

The main contributions of the paper are: 1) a study of the different sanitization methods of PHP and their pitfalls; 2) an approach to automatically develop a fix for XSS and SQLi in web applications; 3) a tool capable of correcting automatically PHP web applications while maintaining their original functionality; and 4) an evaluation with both benchmark test cases and real web applications, demonstrating benefits of our approach.

2 Background

This section gives a brief overview of injection vulnerabilities – SQLi and XSS – and afterwards it explains the most relevant PHP sanitization methods for these bugs and their pitfalls.

2.1 Injection Vulnerabilities

SQL Injection vulnerabilities (SQLi) occur when a crafted input of an attacker can reach a SQL interpreter as part of a query, tricking the database into executing unintended commands. This often allows the bypass of authentication mechanisms, the access to confidential information, or the shut down of the database server. SQLi flaws are usually introduced in web applications who fail to properly validate input data before inserting it into a query. Listing 1 shows a classic example of a SQLi vulnerability, where the user input, received through $\$\_GET['id']$, is placed in the SQL statement that is forwarded by `mysqli_query` to the database. This type of vulnerability can be prevented, for example, by applying proper sanitization to the input and by using parameterized queries to interact with the database.

```
1  $query = "SELECT * FROM Employee WHERE id =
       " . $_GET["id"];
2  $result = mysqli_query($conn, $query);
```

Listing 1: Example of a SQLi vulnerability.

Cross-Site Scripting vulnerabilities (XSS) occur when an application includes untrusted user data as part of a web page without proper validation or encoding (van der Stock et al., 2017). XSS lets an attacker trick the victim's web browser into executing his malicious code. For instance, a traditional PHP XSS flaw exists when a user input is inserted in a web page with the `echo` function. There are several forms of XSS, with new variants continuing to appear periodically (Steffens et al., 2019), and therefore, they are one of the most prevalent vulnerabilities in the web today (WhiteHat Security, 2019). XSS can be averted by properly encoding potentially malicious input before adding it to a web page.

2.2 PHP Sanitization Methods

2.2.1 Generic

There are few sanitization methods that can help with both kinds of vulnerabilities. For numeric inputs, PHP contains the `intval` and `floatval`

functions that preclude many of the SQLi and XSS attacks. Both functions receive a string as argument and return the result of converting that string to an integer or a float, respectively. If they are unable to do the transformation, they return zero, thus making any malicious input innocuous while leaving benign string inputs untouched (i.e., containing just numbers). Alternatively, the conversion can be achieved with casts to numeric types. The casts will execute in a similar way as the previously described functions, thus making inputs harmless.

For string inputs that have a well-known format, such as a date or zip code, there is the possibility of using the `preg_match` function to compare them with a regular expression. In order for this technique to be safe, the developer has to use a correct regular expression.

Lastly, if the input can only assume one of a limited number of values, developers can use white lists for these values. To do so, it is created an array of valid values and used the `in_array` function to verify if the input is part of the array.

2.2.2 Cross-Site Scripting

Sanitization. protection from XSS can be attained with functions that encode special characters like `<` or `>`, making them inoffensive when rendered in the browser.

HTML-encoding functions convert all HTML's special characters to their respective representation as HTML entities, thus preventing attacks that abuse unintended utilization of these values. For example, the `<` character is converted to `<`, thus being properly displayed on the browser. There are two functions in this group: `htmlspecialchars` and `htmlentities`. They receive the same arguments (a string to be sanitized and a set of flags), but differ in the number of characters they convert. The former converts the following characters: `&`, `"`, `'`, `<`, `>`, whereas the latter converts not only these characters but also more than two hundred additional ones, such as `Á` and `Ç`. The flags influence safety because they specify the way the function encodes quotes and the way it deals with the HTML itself (whether it considers the HTML as HTML 4.01 or HTML 5, for example).

URL-encoding functions encode all non-alphanumeric characters to make them harmless when rendered in the browser. As an example, the `<` character is converted to `%3C`, thus being shown on the screen as `%3C` and not being mistakenly understood as the beginning of a tag. There are three functions in this group: `http_build_query`, `urlencode`, and `rawurlencode`. The first one receives an array and returns a URL-encoded query string with all key-value pairs contained in the array. The other two functions get a single string as an argument and differ only in the way they encode spaces. `urlencode` substitutes spaces by `+` while `rawurlencode` encodes spaces as `%20`. All these functions will make their inputs safe, but they may cause usability problems because an URL is shown on the screen in it's URL-encoded form. For this reason, they should only be used in some special situations.

Pitfalls. HTML-encoding functions can stop some variants of XSS when the result is included inside *the content* of any HTML tag, except the `<script>` and `<style>`[2] tags. However, they will allow attacks to go through when the result is placed in *an unquoted part* of any HTML tag's definition. They will also fail if they are called without the `ENT_QUOTES` flag and their result is included inside of a string quoted with single quotes. The most widely recommended way to call these functions safely is to use solely the `ENT_QUOTES` flag.

The URL-encoding functions can prevent XSS in all situations because they encode all non-alphanumeric characters. This means that an attacker is unable to write meaningful Javascript if the input goes through one of these functions. However, the use of HTML-encoding is usually preferred by developers in most situations, as the output of URL-encoding is less appealing when exhibited to the users.

2.2.3 SQL Injection Sanitization

Sanitization. SQLi attacks can be addressed with functions of the family `*_escape_string`. These functions escape SQL's metacharacters to make them safe to be included inside a SQL statement. For the MySQL database, the appropriate functions are `mysql_real_escape_string` and `mysqli_real_escape_string`. Both functions escape the same characters in a similar manner. The difference lies in the PHP extension they use — the former uses the MySQL extension[3] while

[2]Note that the execution of Javascript inside this tag is only possible in older versions of browsers.

[3]MySQL extension was deprecated in PHP 5.5 and removed in PHP 7.

the latter resorts to the MySQL Improved extension.

Often, the recommended way to block SQLi is to employ prepared statements. Prepared statements consist of two phases: (i) in the preparation phase, the statement is sent to the database, which then performs a syntax check and initializes resources for later use; (ii) in the execution phase, the client binds parameter values and sends them to the database. Afterwards, the database executes the statement with the bound values using the previously initialized resources. This ensures that user-supplied values are never treated as SQL commands. For the first task, PHP has function `mysqli_prepare` while for the second phase there are functions `mysqli_stmt_bind_param` and `mysqli_stmt_execute`.

Pitfalls. Functions of the `*_escape_string` family can only prevent SQLi if their output is placed in a SQL string. This happens because they only sanitize characters that can influence a string's limits, such as quotes and line breaks. If their result is, for example, included in a comparison with an integer, the attack will be able to proceed.

It is also important to note that prepared statements do not work in all situations. They do not allow the binding of parameters to table or column identifiers or SQL keywords, meaning that, in this situation, developers should resort to white lists to check the inputs against a set of valid values (however, the study in (Anderson and Hills, 2017) suggests that this situation is uncommon, which means that bugs may remain). Also, prepared statements can be utilized unsafely, which is likely to occur given that they are more complex than simple sanitization functions.

2.2.4 Filters

PHP filters can be employed as a sanitization method by calling the `filter_var` function, and by providing as constant that identifies the filter to be used. This sanitization method can operate both as a generic approach or as a XSS specific solution, depending on the selected filter. Default sanitization filters are all named `FILTER_SANITIZE_*`, and they range from number sanitization (like `intval`) to HTML-encoding (similar to `htmlspecialchars`).

3 Automated Code Correction

This section describes our approach in detail, including a discussion of the main challenges and the decisions that we took to deal with them.

3.1 Code Correction Challenges

A Static Analysis Tool (SAT) must solve three main challenges to be able to correct a web application automatically:

Where to insert the correction? As stated in Section 2.2, some variants of XSS and SQLi bugs can be fixed by inserting a properly configured sanitization function. However, there are usually various locations where to place the call, but some of them might end up breaking the application logic. For example, applying the correction always on the entry point is inappropriate if the application uses that (attacker) input in multiple sensitive sinks, which suffer from different classes of vulnerability (e.g., this would result in multiple amendments put in for the same entry point). However, adding the correction closer to the sensitive sink may also be difficult because there are no guarantees that the code that receives the input will be easy to analyze by the tool (e.g., a SQL query can be quite complex and span over multiple lines).

In order to resolve this challenge, we decided that all corrections should consist solely of adding new lines of code, instead of modifying existing ones. This will help to minimize the chances of causing a program to become syntactically invalid. In addition, if possible, the sanitization of a variable is to be located in a line of code immediately before the sensitive sink where the variable is used. When a tainted variable `var` can not be rectified in this manner, we will sanitize the variable(s) that caused `var` to become tainted, in the line(s) of code immediately before that happened. To do so, we simulate statically the execution of operations that act on the variable. This allows us to know the approximate value of a variable when it reaches a sensitive sink, thus revealing whether the variable can be entirely amended.

What correction to insert? The kind of correction to build is very closely related to the class of vulnerability and the context in which the input is used. Carrying this task thus requires the tool to be capable of reasoning about where the input data is inserted and what is it's expected

type. This asks for an understanding on how a SQL query is setup or the HTML is constructed.

To solve this challenge, we select the patch to apply based firstly on the class of vulnerability that was identified. If it is SQLi, the tool inserts a string escaping function accordingly with the sensitive sink. If it is XSS, the tool adds URL-encoding functions whenever possible and HTML-encoding functions in all other cases.

How to deal with existing sanitization? Existing sanitizations in the code pose another challenge because they might be insufficient to prevent all attacks. In such cases an automated tool has to decide between making some modifications to the existing sanitization or adding it's own fix to the program.

To tackle this difficulty, we decided that our approach would need to have the capability of reasoning about diverse sanitization methods. If the sanitization method in use is safe, the variable is marked as untainted, meaning that no correction is introduced. Otherwise, the remedy will be applied following the ideas presented for the first challenge. Note however that the problem we are tackling is undecidable in general. Therefore, in some cases, the application might contain a safe sanitization method that is regarded as unsafe by our solution. Here, the repair principles explained for the first challenge should help us prevent our correction from breaking the application's logic.

3.2 Overview of the Solution

Our approach aims to correct PHP web applications by inserting new lines of code that sanitize or validate inputs arriving at the entry points, which are later used in sensitive sinks in an unsafe manner. It is our intention to avoid possible syntactic errors or breaking the application logic, but it is out of scope to patch the functional behaviour of the application.

The tool starts by receiving as input a *slice* of code, containing a data flow beginning at an (or more) entry point and ending at a sensitive sink, and information about the class of vulnerability that the slice may suffer. The slice is then analysed by a solution inspired on taint tracking to discover which variables have to be sanitized or validated, and where rectify the slice. The outcome of the tool is a safe slice. Notice, however, that the analysis may deem a slice as non-vulnerable, and in this case no changes are made. Figure 1 provides an overview of the architecture

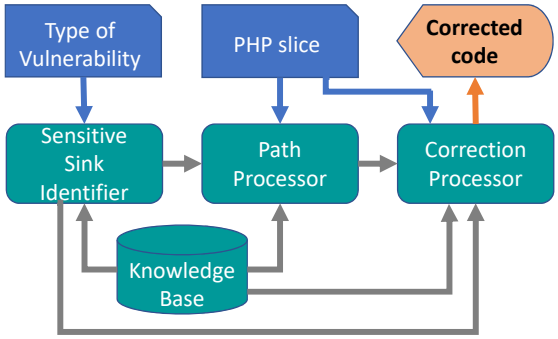

Figure 1: Automated code correction approach architecture.

of the tool. Next, we detail each component.

PHP slice. It is a PHP file containing a potentially vulnerable slice of code (typically produced by a vulnerability detection tool). The slice contains the instructions of a program corresponding to a data flow path that takes some input from an entry point and carries it to a sensitive sink. The slice can include more instructions than the strictly relevant for the vulnerability exploitation as long as it contains a single control flow path. Also, it can have multiple vulnerabilities if they are all of the same class.

Vulnerability type. It is a file with information about the class of vulnerability that the slice may have. Currently, only XSS and SQLi are supported. Our solution requires this as input because it allows the taint analysis to be more efficient and narrows down the sanitization functions that should be considered.

Knowledge Base. Contains the names of all PHP entry points, sensitive sinks and sanitization methods to be considered by our approach for each class of vulnerability. It is used by the Path Processor to get the entry points and sanitization methods, and by the Sensitive Sink Identifier to collect the sensitive sinks.

Sensitive Sink Identifier. Discovers the sensitive sinks in the PHP slice for all classes of vulnerability supported by our approach. Its output is used by the Path Processor and the Correction Processor to check whether a given slice instruction is a sensitive sink.

Path Processor. Performs the taint analysis and simulates the execution of operations with

the variable. It tracks the slice's input as it is processed and passed along the instructions, from the entry points to the sensitive sinks, and maintains the taint status of the program's variables. To perform these tasks, it consumes information from the Knowledge Base and the Sensitive Sink Identifier.

Correction Processor. Determines which variable(s) need to be sanitized, what corrections they require and the line(s) of code where those fixes should be applied. This is done using the information produced by the Sensitive Sink Identifier and Path Processor. The component is also responsible for generating the actual instructions to be inserted in the output file, including the appropriate parameters.

Corrected Code. Consists of a PHP file containing the patched version of the slice of code provided as input. No file is returned if no repair is necessary, as the original slice was not changed.

Algorithm 1 presents the high level steps of our approach. It starts by generating an Abstract Syntax Tree (AST) of the PHP slice. Then, it calls the Path Processor to perform the taint analysis and simulate the variable operations. This is done to confirm the vulnerabilities in the code and to compute the state of the program's variables. Next, it calls the Correction Processor to analyze the sensitive sinks, determine the required corrections and where they should be applied. Lastly, it inserts all corrections in the slice to generate the output file.

3.3 Variable Operation Simulation

Our approach simulates statically the operations involving PHP variables. This is done while the taint analysis is being performed and consists of simulating the execution instructions, like string concatenations and arithmetic operations with numbers. When an input value is involved in one of these operations, a special marker is put in it's place, thus indicating that part of the result might be influence by an attacker. Such approach allows for example to keep track where a user input might be inside of a string. This is useful to ascertain if a string might contain HTML tags, in the case of XSS, or to determine if the input is being included in a query, in the case of SQLi. However, it is important to note that this simulation might not always obtain the exact value of a variable, such as with function calls and arrays. In any case, the result is still useful for the majority of situations, as an approximate value for a variable is normally sufficient to make the right selection of the remedy.

Algorithm 1: Approach algorithm.

Input: PHP slice; Type of vulnerability
Output: Corrected PHP slice

```
1 Function Main(slice, vulnerability):
2     ast ← GenerateAST (slice);
3     state ← ProcessPath (ast, vulnerability);
4     corrections ← ProcessCorrections (ast, state,
        vulnerability);
5     for cor in corrections do
6         InsertLine (slice, cor.code, cor.line);
7 End Function
```

```
1  $in = $_GET["a"];
2  $html = "Input user data -" . $in;
3  $i = 10;
4  $j = $i + 1;
5  echo $html;
```

Listing 2: PHP program vulnerable to XSS.

Listing 2 shows an example of a simple program with operations involving strings and integers. The simulation of variable operations assigns the `::input` special marker to `$in` in line 1, to state that it contains some external value. Next, in line 2, it concatenates the string `"Input user data -"` with the value of variable `$in` to form the simulated value of `$html`. In line 3, the integer `10` is assigned to `$i`, and then, in line 4, the value of `$i` is summed with the integer `1` to obtain the simulated value of `$j`. Lastly, in line 5, the value of `$html` is used in the sensitive sink `echo`. Note that the simulation takes place statically, without ever executing the code. The result of the simulation is a list of $< key : value >$ pairs, in which the keys are the names of the variables and the values are their respective simulated values. An example of such list is shown in Listing 3, taking the code of Listing 2. In this list, the string `::input` corresponds to the special marker inserted in the place where the input was added.

```
1  $in    : '::input'
2  $html  : 'Input user data -::input'
3  $i     : 10
4  $j     : 11
```

Listing 3: Result of simulating the variable operations for the program in Listing 2.

3.4 Code Correction

The Correction Processor uses the results from the simulations and taint tracking, plus the line numbers of relevant sensitive sinks for the specific class of vulnerability, to prescribe a correction and where it should be applied. Taking as example the program in Listing 2 and the simulation of Listing 3, the Correction Processor is activated when the Path Processor identifies that the tainted variable `$html` reaches the `echo` function. It observes that the variable contains the special marker `::input` and backtracks the variable until its last assignment. Next, it uses the data provided by taint analysis to get the name of the variable that carries the special marker. In addition, based on the sensitive sink, it decides on the most appropriate sanitization function that should be added. This way, the Correction Processor generates the instruction `$in = htmlentities($in, ENT_QUOTES);` and places it right before the line where the assignment to `$html` was done, i.e., before line 2 on Listing 2.

4 Experimental Evaluation

The approach was implemented in a tool that we call PHPCorrector[4]. The tool was developed in Python and uses as parser a modified version of the PHPly[5], which supports both PHP 5 and PHP 7. The taint analysis, simulations, and corrections were implemented from scratch.

PHPCorrector is evaluated by correcting SQLi and XSS vulnerabilities in two sets of web applications. The first set is based on the programs of the NIST benchmark SARD - Software Assurance Reference Dataset[6] (Section 4.1), and the other test set utilizes real vulnerable applications that were selected from Exploit-DB[7] (Section 4.2). The research questions that are answered by the experiments are: (1) Is PHPCorrector able to validate the vulnerabilities in SARD and real web applications? (2) Is PHPCorrector able to fix XSS / SQLi vulnerabilities? (3) Are the corrections done appropriately?

[4]PHPCorrector is available at https://phpcorrector.sourceforge.io

[5]https://github.com/viraptor/phply

[6]https://samate.nist.gov/SRD/

[7]https://www.exploit-db.com/

4.1 SARD Test Cases

4.1.1 Dataset Characterization

We gathered a total of 1864 test cases from SARD, namely 1764 XSS and 100 SQLi. All of these test cases contain a single data flow path, meaning that they fit our definition of a slice of code. They have different types of entry points (e.g., $_GET, $_POST), vary from no sanitization, type casts or distinct forms of sanitization, and hold a single kind of sensitive sink, either `mysql_query` (for SQLi) or `echo` (for XSS).

While analyzing slices manually, we discovered that some of SARD's test cases are mislabelled. There are safe (not-vulnerable) test cases that are considered as unsafe (vulnerable) and vice-versa. For this reason, we ran a more thorough analysis to determine the actual label that the test cases should have. Our dataset contains 1494 test cases that kept their original SARD labels and 370 test cases whose labels had to be adjusted. Summarizing, the dataset is composed of 420 unsafe XSS, 1344 safe XSS, 13 unsafe SQLi, and 87 safe SQLi.

4.1.2 XSS and SQLi Evaluation

PHPCorrector was able to process all test cases. It is important to note that each unsafe test case contains a single vulnerability that requires one correction. This means that the number of vulnerabilities detected by our tool is equal to the number of patches it applied. For XSS, the tool always chose one of two sanitizations: i) a call to the `htmlentities` function with the `ENT_QUOTES` flag, or ii) a call to the `rawurlencode` function. For SQLi, the tool also always applied the same correction: a call to the `mysql_real_escape_string` function.

		Observed XSS		Total
		Vul	N-Vul	
Tool XSS	Vul	308	172	480
	N-Vul	112	1172	1284
Total		420	1344	1764

Table 1: Summary of results for XSS.

		Observed SQLi		Total
		Vul	N-Vul	
Tool SQLi	Vul	10	13	23
	N-Vul	3	74	77
Total		13	87	100

Table 2: Summary of results for SQLi.

Reason		Test Cases	Total
XSS-FNRe1	**Inclusion of input inside unquoted attributes:** These false negatives occurred for test cases that include the input inside of an unquoted HTML attribute. This makes the test cases vulnerable because it is possible to write nonexistent Javascript event handlers without using HTML's special characters.	16	112
XSS-FNRe2	**Inclusion of input inside CSS:** These false negatives occurred for test cases that include their input inside of a `<style>` tag. This makes them vulnerable because certain versions of some browsers allow the execution of some Javascript statements inside of CSS.	32	
XSS-FNRe3	**Inclusion of input inside of a script tag:** False negatives also occurred for test cases that include their input inside of a `<script>` tag. This makes them vulnerable because it is possible to write meaningful Javascript without using HTML's special characters.	32	
XSS-FNRe4	**Inclusion of input in a HTML tag name:** These false negatives occurred for test cases that include their input in the place of a HTML tag name. Similarly to the first reason, an attacker can craft a malicious input that adds nonexistent Javascript event handlers without using HTML's special characters.	16	
XSS-FNRe5	**Inclusion of input in a HTML attribute name:** This reason is very similar to the previous one, except that the test case's input is included in the place of a HTML attribute name.	16	
XSS-FPRe1	**Unsafe sanitization used in a context that makes it safe:** These false positives occurred for test cases that sanitize quotes. The inclusion of input inside of a Javascript string or a quoted CSS property value is safe in these cases because any quotes present in the input are sanitized by preceding them with backslashes. This means that an attacker can not execute meaningful code.	72	172
XSS-FPRe2	**Use of a sanitization method involving a regular expression:** All the false positives that occurred for this reason were caused by calls to `preg_replace` with a safe regular expression. Our tool does not currently handle regular expressions, meaning that any calls to `preg_replace` are considered to return tainted data, regardless of the regular expression used.	100	

Table 3: FN and FP explanations and numbers for XSS.

A summary of the results is presented in Tables 1 and 2, respectively for XSS and SQLi. Out of the 1764 XSS test cases, 1480 were correctly identified by the tool as being vulnerable (308) and not-vulnerable (1172). The remaining 284 test cases, there were cases where unnecessarily amendments were made (172 false positives, FP) and where no fix was done because the slice was deemed secure (112 false negatives, FN). For SQLi, out of the 100 test cases, the tool only undetected 3 vulnerabilities and generated 13 FP, while the remaining 74 cases were correctly processed. We investigated the key reasons that could explain the FN and FP. The main conclusions are discussed next and presented in Tables 3 and 4. Although we think our results are very promising, we intend in the future propose complementary solutions to address the mislabeling causes.

XSS FN and FP. XSS FN were always observed in test cases that resort to improper sanitization methods while encoding HTML's special characters, which were regarded as safe by our tool. Their detailed causes are described in lines 2–6 of Table 3 (identified as XSS-FNRex) and the number instances is showed in column 3. We believe that FNs related with XSS-FNRe2 and XSS-FNRe3 could be avoided if the tool's taint analysis was able to track the type of sanitization function that was being applied to a variable. This would allow the tool to reason that, for example, a variable is unsafe to be include in a `<script>` tag if it was sanitized by a HTML-encoding function. As for the remaining XSS-FNRes, averting them would require a detailed analysis of the context in which a tainted variable is added to the output. The XSS FP appeared in test cases that employ unsafe sanitization methods but that fortunately are used in a secure context, and in test cases that utilize a sanitization method involving a regular expression. Lines 7–8 elaborate on these reasons (identified by XSS-FPRex).

SQLi FN and FP. Regarding the SQLi FN, all 3 cases are explained by the same reason – sanitization of numeric data (line 2 of Table 4). The test cases had an unsafe usage of this op-

Reason		Test Cases	Total
SQLi-FPNe1	**Use of a sanitization method to sanitize numeric data:** These FN were caused by the use of the `mysql_real_escape_string` function to sanitize data that is later included in a comparison with an integer.	3	3
SQLi-FPRe1	**Use of a sanitization method that escapes quotes:** The FP occurred due to the use of addslashes (or an equivalent filter) to sanitize the input before it is included in the query. This type of sanitization is regarded as unsafe by our tool but it is safe in these test cases because any quotes present in the input are sanitized by preceding them with backslashes.	2	13
SQLi-FPRe2	**Use of a XSS sanitization method:** Occured due to the use of a XSS sanitization method to sanitize the input. This is safe in these test cases. However, it is not considered safe by our tool because XSS sanitization functions should never be used to prevent SQLi.	7	
SQLi-FPRe3	**Use of a sanitization method involving a regular expression:** These FP occurred due to a call to `preg_replace` with a safe regular expression. As mentioned before, our tool considers calls to `preg_replace` to return tainted data.	2	
SQLi-FPRe4	**Use of a numeric format specifier:** The FP occurred due to the use of a numeric format specifier in a call to `sprintf`. This effectively consists of casting the input to a numeric type, which was described in Section 2.2.1.	2	

Table 4: FN and FP reasons and numbers for SQLi.

eration, as explained in Section 2.2. Avoiding these FN would require knowledge about the data types of the database tables and a more detailed analysis of the query's structure. We believe that the FP associated to reasons SQLi-FPRe1 and SQLi-FPRe2 can not be avoided because these two methods of sanitization should not be considered safe for SQLi. As for the remaining FP, stopping them would require a better analysis of regular expressions or the simulation of calls to `sprintf`.

4.1.3 Applied Corrections

We manually analyzed all corrections that were applied by PHPCorrector to assess how many of them actually prevent attacks. It is important to note that none of the repaired programs became syntactically or semantically invalid. In total there were 503 test cases amended (480 XSS and 23 SQLi). To complete this task, we looked at the location where the potentially malicious input was included in the program's output to verify the safety of the fix. Corrections were organized in the following three groups: (i) *Safe:* all attacks are prevented, making the programs safe; (ii) *Unsafe:* some forms of attack remain active, leaving the programs still vulnerable; (iii) *Unneeded:* changes were applied to non-vulnerable test cases indicating that they were unnecessary.

Table 5 shows the number of test cases in each group. Most of the cases correspond to the Safe class (237), and so the tool performed well. With regard to the 185 Unneeded, the repair did not spoil the program and therefore it is innocuous. These cases correspond to the FP discussed in the previous section. The remaining 81 Unsafe cases are part of the 318 test cases (see Tables 1 and 2) that the tool properly detected as being vulnerable but that the correction is not sufficient to completely prevent the attacks.

Group	XSS	SQLi	Total
Unneeded	172	13	185
Safe	228	9	237
Unsafe	80	1	81
Total	480	23	503

Table 5: Number of XSS and SQLi corrections applied by PHPCorrector.

4.2 Real Web Applications

We used six web applications that were vulnerable to XSS from Exploit-DB to validate our tool with real programs. Table 6, on the first 5 columns, characterizes the applications that were assessed. Each application has a type, a vulnerable version, the number of PHP files and the number of PHP lines of code (LoC). Considering XSS, these applications contain a mixture of sensitive sinks, such as `echo`, `print`, `die` and `exit`. This shows that our tool is capable of detecting sensitive sinks other than `echo`, which is the only XSS sensitive sink in the test cases of SARD.

Among the six applications, the tool found 38 PHP files to be vulnerable with a total of 79 variables needing protection. All the identified bugs were automatically corrected, and the re-

Application	Vuln. Version	PHP Files	PHP LoC	Type of Application	Vuln. Files	Correction		
						Applied	Safe	Unsafe
Site@School	2.4.10	567	64 k	Content Management System for Primary Schools	13	16	16	0
Integria IMS	5.0.83	974	198 k	IT Service Support Management Tool	5	5	5	0
Electricks eCommerce	1.0	45	7 k	E-Commerce Website	5	27	23	4
userSpice	4.3	474	114 k	User Management Application	1	1	1	0
AShop Shopping Cart	6.0.2	628	113 k	Shopping Cart Software	8	24	24	0
I, Librarian	4.6	114	26 k	PDF File Manager	6	6	6	0
	Total	2,802	522 k	–	38	79	75	4

Table 6: Characterization of the real applications and results of our evaluation over them.

sulting repaired programs were all syntactically valid (see columns 6 – 8 of the table). There were 72 amendments using HTML-encoding functions and 7 using URL-encoding functions. On 77 occasions, the correction was applied to the tainted variable itself and, on 2 situations, on a variable's taint causes because the variable itself contained HTML tags in it's simulated value.

As shown in the table, 75 of the 79 patches are safe, preventing all attacks. Four fixes performed on Electricks eCommerce reduce the attack surface, partially protecting the application, but leave a few attacks vectors active. Therefore, they were considered unsafe. To better understand these cases, Listing 4 provides an example. The inserted sanitization is the call to `htmlentities` in line 2. Notice that the input is then used in the `value` attribute in line 3, without being surrounded by any quotes. Thus, it allows the addition of new HTML attributes, such as `onmouseover`. Therefore, an example input for `$_GET['prod_id']` that could still trigger the vulnerability is `1 onmouseover=alert(1)`.

```
1  <div class="form group">
2  <?php $_GET['prod_id'] = htmlentities($_GET[
       'prod_id'], ENT_QUOTES); ?>
3  <input type="hidden" class="form-control" id
       ="prod_id" name="prod_id" value=<?php
       echo $_GET['prod_id'];?>>
```

Listing 4: Correction applied to Electricks eCommerce (simplified for readability).

5 Related Work

Static analysis has the objective of analyzing the source code of an application to find vulnerabilities but without executing it. SATs compare the code to a set of patterns that indicate a vulnerability. If the tools knowledge base does not contain a pattern for a given type of vulnerability, the tool will not report it, leading to a false negative. On the other hand, they may report false positives, which they are a concern to developers since they spend time looking for unexistent problems. Most SATs still require human intervention to verify that the bugs reported are in fact vulnerabilities, and fix them.

One of the forms of static analysis is taint analysis. It consists in track input variables (entry points), stating them as tainted, and verify if they are used as arguments of functions that expect to receive untainted data (sensitive sinks). Taintedness is propagate along the program analysis, but if a tainted variable is passed through a sanitization function, it is stated as untainted.

Halfond et al. (Halfond et al., 2008) developed a novel form of taint analysis that they referred to as positive tainting for detection of SQLi. Their technique is based on the marking and tracking of trusted data, instead of untrusted data.

Dashe et al. (Dahse and Holz, 2014) proposed an approach that detects second-order vulnerabilities in web applications, such as stored XSS and second-order SQLi. The approach identifies taintable data stores and checks if a symbol originating from such a data store reaches a sensitive sink without being sanitized.

Livshits and Lam (Livshits and Lam, 2005) developed a static analysis approach that is based on context sensitive pointer alias analysis and introduced extensions to the handling of strings and containers to improve the precision.

AMNESIA (Halfond and Orso, 2005) is a tool that protects web applications from SQLi attacks, resorting from static and dynamic analysis. In the static phase, the tool inspects the applications code to get a model of all queries. In the dynamic

phase, the tool monitors the application to detect SQLi attacks based on the extracted models.

Flynn et al. (Flynn et al., 2018) developed and tested classification models that predict if static analysis alerts are true or false positives, using a combination of multiple SATs. Other works study the problem of combining SATs (Nunes et al., 2017) (Algaith et al., 2018).

In recent years, there have also been some research efforts focused on applying machine learning (ML) approaches to the detection of vulnerabilities in source code (Grieco et al., 2016) (Shar and Tan, 2012) (Shar et al., 2013) (Yamaguchi et al., 2011) (Medeiros et al., 2016).

There are a few SATs that employ code correction. WebSSARI (Huang et al., 2004) is a tool that statically finds XSS and SQLi vulnerabilities and then remove them by inserting guards to secure it. The authors, however, do not explain how the guards are inserted or what they consist of. Medeiros et al. (Medeiros et al., 2014) developed WAP, a SAT for PHP web applications. The most novel aspects of the tool are the use of data mining to predict false positives and automatic code correction. WAP uses taint analysis to find several types of vulnerabilities, and then to be processed by data mining to classify each one as a false positive or not. Lastly, the vulnerabilities that were not classified as false positives are corrected automatically.

6 Conclusion

In this work, we proposed an approach for automated code correction of PHP web application, visioning removing XSS and SQLi vulnerabilities from their code and improving their security. For that, firstly, we analyzed a multitude of sanitization methods available in PHP for both of these vulnerabilities and the situations when they should be applied and they do not work as expected. Also, we verified that the existent SATs do not apply code correction, and the very few ones that apply they often produce new programs that are syntactically invalid and can not be executed. We proposed an approach based on taint analysis to find and correct vulnerabilities in simplified PHP programs (i.e., slices of code) by adding new lines of code containing secure corrections, taking into account the defects of sanitization functions. We implemented the approach in the PHPCorrector static analysis tool, written in Python. The developed tool was evaluated using both test cases retrieved from SARD and real web applications obtained from Exploit-DB. The results showed that all corrected programs were syntactically valid and preserved their original behavior for both types of vulnerabilities.

Acknowledgments

This work was partially supported by the national funds through FCT with reference to SEAL project (PTDC/CCI-INF/29058/2017), and LASIGE Research Unit (UIDB/50021/2020).

REFERENCES

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L., and Stransky, C. (2016). You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Algaith, A., Nunes, P., Fonseca, J., Gashi, I., and Viera, M. (2018). Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools. In *Proceedings of the European Dependable Computing Conference*.

Anderson, D. and Hills, M. (2017). Query Construction Patterns in PHP. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*.

Dahse, J. and Holz, T. (2014). Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*.

Fischer, F., Bttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. (2017). Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Flynn, L., Snavely, W., Svoboda, D., VanHoudnos, N., Qin, R., Burns, J., Zubrow, D., Stoddard, R., and Marce-Santurio, G. (2018). Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. In *Proceedings of the International Workshop on Software Qualities and Their Dependencies*.

Grieco, G., Grinblat, G. L., Uzal, L., Rawat, S., Feist, J., and Mounier, L. (2016). Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*.

Halfond, W. G. J. and Orso, A. (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*.

Halfond, W. G. J., Orso, A., and Manolios, P. (2008). WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering*.

Huang, J., Li, Y., Zhang, J., and Dai, R. (2019). UChecker: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

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

Livshits, V. B. and Lam, M. S. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium*.

Medeiros, I., Neves, N. F., and Correia, M. (2014). Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives. In *Proceedings of the International World Wide Web Conference*.

Medeiros, I., Neves, N. F., and Correia, M. (2016). DEKANT: a static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

Nunes, P., Medeiros, I., Fonseca, J., Neves, N. F., Correia, M., and Vieira, M. (2017). On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study. In *Proceedings of the European Dependable Computing Conference*.

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

Shar, L. K. and Tan, H. B. K. (2012). Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities. In *Proceedings of the International Conference on Software Engineering*.

Shar, L. K., Tan, H. B. K., and Briand, L. C. (2013). Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis. In *Proceedings of the International Conference on Software Engineering*.

Steffens, M., Rossow, C., Johns, M., and Stock, B. (2019). Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *Proceedings of the Network and Distributed System Security Symposium*.

van der Stock, A., Glas, B., Smithline, N., and Gigler, T. (2017). Owasp Top 10 2017 The Ten Most Critical Web Application Security Risks. Technical report, OWASP.

WhiteHat Security (2019). Technical report, WhiteHat Security.

Yamaguchi, F., Lindner, F., and Rieck, K. (2011). Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the USENIX Conference on Offensive Technologies*.

Zheng, Y., Zhang, X., and Ganesh, V. (2013). Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*.