2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 978-1-6654-1658-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/TRUSTCOM53373.2021.00112

VulChecker: Achieving More Effective Taint Analysis by Identifying Sanitizers Automatically

Xiarun Chen
School of Software and Microelectronics
Peking University
Beijing, China
xiar_c@pku.edu.cn

Qien Li
Digital Star Technology Co.
Sichuan, China
liqien@pku.edu.cn

Zhou Yang
School of Software and Microelectronics
Peking University
Beijing, China
yzss2019@pku.edu.cn

Yongzhi Liu
School of Software and Microelectronics
Peking University
Beijing, China
lyz_cs@pku.edu.cn

Shaosen Shi
School of Software and Microelectronics
Peking University
Beijing, China
deadpoo3@pku.edu.cn

Chenglin Xie
School of Software and Microelectronics
Peking University
Beijing, China
cony1996@pku.edu.cn

Weiping Wen
School of Software and Microelectronics
Peking University
Beijing, China
weipingwen@pku.edu.cn

***Abstract*—The automatic detection of vulnerabilities in Web applications using taint analysis is a hot topic. However, existing taint analysis methods for sanitizers identification are too simple to find available taint transmission chains effectively. These methods generally use pre-constructed dictionaries or simple keywords to identify, which usually suffer from large false positives and false negatives. No doubt, it will have a greater impact on the final result of the taint analysis. To solve that, we summarise and classify the commonly used sanitizers in Web applications and propose an identification method based on semantic analysis. Our method can accurately and completely identify the sanitizers in the target Web applications through static analysis. Specifically, we analyse the natural semantics and program semantics of existing sanitizers, use semantic analysis to find more in Web applications. Besides, we implemented the method prototype in PHP and achieved a vulnerability detection tool called VulChecker. Then, we experimented with some popular open-source CMS frameworks. The results show that Vulchecker can accurately identify more sanitizers. In terms of vulnerability detection, VulChecker also has a lower false positive rate and a higher detection rate than existing methods. Finally, we used VulChecker to analyse the latest PHP applications. We identified several new suspicious taint data propagation chains. Before the paper was completed, we have identified four unreported vulnerabilities. In general, these results show that our approach is highly effective in improving vulnerability detection based on taint analysis.**

Keywords—vulnerability detection, taint analysis, sanitizers identification，security check

I. INTRODUCTION

Web applications play an extremely important role in the Internet ecosystem, and their security issues are equally far-reaching [1]. Among the many security risks, taint-type vulnerabilities are one of the most prevalent and threatening types. This category of vulnerabilities usually refers to security risks caused by malicious external inputs, such as SQL injection, XSS, etc. In recent years, many methods have been proposed by researchers to analyse Web applications to detect taint-type [2] vulnerabilities. Among them, static code analysis has been widely studied because of its efficiency benefits [3-5]. This method can get the structure and characteristics of the applications without running and thus analysing potential security risks. Among the many static analysis methods, the method based on taint analysis [6, 7] is a hot research topic. It tracks the flow of data in a program and analyses the source and propagation of data to determine if there is a security risk. This method models the source and use of data, which is similar to the approach used when manually auditing code to find vulnerabilities. However, the static taint analysis methods also suffer from a high rate of false positives [8]. In the current research on taint analysis, researchers focused more on solving the analysis problem of alias propagation [9-11]. These taint analysis methods analyse whether data can be transmitted directly from the taint source to the taint aggregation point without going through sanitizers, where sanitizers are used to process the tainted data in order to remove sensitive information or dangerous data. As an important component of taint analysis, sanitizer is also an important influence of taint analysis results [12]. Among these existing methods of sanitizers identification, library function dictionaries [4, 6] and keyword matching [13] are commonly used. However, the development of Web applications is complex and changeable. It is difficult to identify a valid set of keywords for the effective identification of sanitizers. In other words, the existing identification methods may have many misses and false positives, and this can further affect the accuracy of the vulnerability detection results.

It is worth noting that the identification of sanitizers is not only relevant for taint analysis. A more accurate identification method can be applied to many aspects of program analysis and security management. One example is the identification of irregularities in data inspection. Combining the identification

2324-9013/21/$31.00 ©2021 IEEE
DOI 10.1109/TrustCom53373.2021.00112

of sanitizers with statistical methods, it is possible to find out which security checks are imperfect [14]. In addition to this, by identifying sanitizers we can guide fuzzing to better trigger critical questions [15,16]. Identifying sanitizers can focus the program analyst or security inspector, selectively concentrating on a certain set of targets that are more likely to be threatened, thus improving the efficiency and accuracy of each task [17].

In this paper, we focus on the identification of sanitizers in taint analysis, and propose more effective methods for vulnerability analysis. To do this, we summarise and categorise several types of commonly used sanitizers by analysing a large number of open-source Web applications. And we also carry out a semantic modeling analysis of these sanitizers. In addition, we design a sanitizers identification method based on semantic analysis. Firstly, we use natural semantic analysis to obtain the set of suspected sanitizers. Secondly, we combine data flow analysis and control flow analysis to obtain the program semantics, filter and further confirm the elements in the suspicious set. Finally, we get a more accurate set of sanitizers. We implemented method prototypes in PHP code and analysed popular CMS frameworks. By the time the paper was completed, we had identified four unreported vulnerabilities.

To summarize, we make the following contributions:

1. We analyzed the top 30 Web applications with the most stars in Github [18] and compiled the defenses against taint-type vulnerabilities in them. We defined, categorised and classified these sanitizers and described them using a semantic model. We hope that this work will inform the design of subsequent sanitizers identification methods and provide inspiration for developers to build secure defences.

2. We propose a method to identify sanitizers based on semantic analysis. Through this method, we can perform fast and accurate identification of sanitizers in Web applications, thus providing a basis for vulnerability detection.

3. We implemented a prototype of the method on PHP code and implemented a vulnerability detection tool (VulChecker). By compared Vulchecker with popular vulnerability detection methods, we can see that our method can identify more sanitizers. In addition, VulChecker has a lower false positive rate and a higher detection rate for vulnerability detection.

II. Background and Related Work

A. Vulnerability detection technology based on taint analysis

Among program analysis techniques, taint analysis is an important tool for analysing code vulnerabilities and detecting attack methods [19-22]. It has a very wide range of applications in automated vulnerability detection. In vulnerability detection, we mark the data (usually external input from the program) as tainted data and then track the flow of it [6]. By doing this, we can see if the tainted data affects critical program operations and detect program vulnerabilities.

Taint analysis can be abstracted into a triad of *<sources, sinks, sanitizers>*.

Source: The source of the taint, which represents the direct introduction of untrusted data or confidential data into the system [23].

Sink: A taint aggregation point, which represents an instruction that can directly generate dangerous operations or compromise private data to the outside world [24].

Sanitizers: They sanitise data by encrypting or removing compromising operations so that the data is no longer compromising to the application [25].

Using the abstract definition above, we can generalise the process of vulnerability detection based on taint analysis. It analyses whether data introduced by a taint source in a program can be propagated directly to the taint aggregation point without sanitizers. If it cannot, it indicates that the system has a high probability of being secure. Otherwise, it indicates that the system may have security problems.

As shown in Fig. 1, taint variable 1 reaches the taint aggregation point via taint propagation. Which indicates that there may be a problem here. Taint variable 2, on the other hand, undergoes sanitizers in taint propagation and cannot reach the taint aggregation point directly, so this data stream is safe.

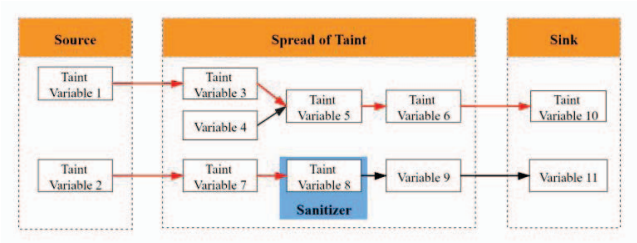

Fig. 1. The process of taint analysis to detect vulnerabilities.

From Figure 1 we can see that the key to taint analysis-based vulnerability detection methods is the analysis of data propagation. In this part, the analysis for direct assignment propagation and function call propagation is more mature. Researchers are currently focusing on the analysis of alias propagation. However, as we can see from Figure 1, the impact of sanitizers on the results of the entire propagation chain cannot be ignored. Therefore, the identification of sanitizers is equally important.

B. Sanitizers and recognition technology in taint analysis

Sanitizer is an application protection mechanism used by developers [26]. It processes data so that the data no longer carries sensitive data or is no longer harmful to the applications. In taint analysis, taint marks are removed when the tainted data passes through the sanitizers. Effective sanitizers identification during vulnerability detection can reduce the amount of tainted data and increase efficiency. Also, it can avoid the problem of inaccurate analysis results due to taint proliferation.

Among the existing taint analysis methods, the sanitizers generally used include two categories [25].

Data encryption functions [27]: In Web applications, developers often encrypt important data in order to prevent sensitive data. Encrypted data is usually difficult to extrapolate and no longer threatening, so researchers often identify this class of functions as sanitizers.

Input validation functions: For Web applications, external input data can be harmful to the application by carrying dangerous operations. To defend against such hazards, developers often use some validation functions. In line 9 of the code in Fig. 2, the developer uses the "*htmlspecialchars()*" function to validate the input of "*name*", This function encodes the HTML tags in the data, thus defending against XSS vulnerabilities. In taint analysis, these functions are also identified as sanitizers. In this section, in addition to the input validation library functions that come with the programming language, some systems provide additional input validation tools, such as ScriptGard [28], CSAS [29], XSS Auditor [30], BEK [31]. These tools are also regarded as sanitizers.

In existing studies on sanitizers, researchers have focused on the effectiveness of sanitizers [2,33,34] and how to automate their placement [35]. But the identification for sanitizers has been less studied.

```
1  <?php
2
3  // Is there any input?
4  if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
5      // Check Anti-CSRF token
6      checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );
7
8      // Get input
9      $name = htmlspecialchars( $_GET[ 'name' ] );
10
11     // Feedback for end user
12     $html .= "<pre>Hello ${name}</pre>";
13 }
14
15 // Generate Anti-CSRF token
16 generateSessionToken();
17
18 ?>
```

Fig. 2. Example of using a library function as a sanitizer in DVWA[32].

Through our research, we found that the sanitizers in current taint analysis are generally functions. And in terms of identification methods for sanitizers, researchers often use pre-constructed dictionaries of library functions [4, 6] or collections of keywords [13] for identification. For example, in paper [13], when using taint analysis for vulnerability detection in OpenMRS, the researchers construct dictionaries by collecting the filter functions provided by Hibernate.

As can be seen, both in terms of the semantic level of sanitizers(generally functions) used today and in terms of identification methods, the sanitizers identification methods used in current taint analysis methods are relatively simple. Although these methods can be useful, there are still major problems in terms of accuracy. In addition to using the library functions as sanitizers, developers are increasingly custom sanitizers. Existing identification methods struggle to cope with the increasing complexity of Web application development. There is no way to ever guarantee developer habits, so these sanitizers are often difficult to match using a particular character rule.

Furthermore, in addition to function-level sanitizers, security check statements in Web applications should not be ignored. Such checks may be present within any function in the code, such as the security check branch statement on line 10 of the code in Fig. 3. Security checks can also have the same effect as input validation functions. However, in existing taint analysis, this type of security check is often difficult to identify accurately and even ignored.

```
1  <?php
2
3  // Security check function
4  function safecheck($strstr){
5      if(preg_match("/select|union|prepare|set|update|delete|drop|insert|where|\./i",$strstr)) {
6          die('no hacker!');
7      }
8  }
9
10 // The function to excute sql
11 function excutesql($name){
12     $sql = "select * from `users` where name = '$name';";
13     $res = $db->mysqli_query($sql);
14     return $res;
15 }
16
17 $name = $_GET['name'];
18
19 // Sanitize name input
20 safecheck($name);
21 var_dump(excutesql($name));
22
23 if(isset($_GET['inject'])) {
24
25     $id = $_GET['inject'];
26
27     // Sanitize inject input
28     if(preg_match("/select|union|prepare|set|update|delete|drop|insert|where|\./i",$id)) {
29         die('no hacker!');
30     }
31     // Splice and excute sql
32     $sql = "select * from `words` where id = '$id';";
33     $res = $db->mysqli_query($sql);
34     var_dump($res);
35 }
36
37 ?>
```

Fig. 3. Example of using a security check as a sanitizer.

In summary, we found the following problems with existing sanitizers and identification methods:

(1) Most of the sanitizers collections currently in use are library functions. This collection contains the encryption and input validation functions that come with the programming language library. However, such collections cannot include custom functions, which can lead to certain false positives.

(2) To extend the sanitizers functions, researchers use keywords to identify more sanitizers functions. Bug such a simple approach tends to introduce larger false positives and misses. On the other hand, the keyword matching approach is difficult to identify security checks.

To deal with challenge (1), we analysed the top 30 Web applications on Github with most stars. By summarising the taint-type vulnerabilities and sanitizers in these open source applications, we describe the sanitizers with natural and program semantics. This will serve as the basis for designing an automatic identification method. To address challenge (2), we designed a semantic analysis-based approach to identify sanitizers. It combines natural semantic analysis with program semantic analysis to identify sanitizers in Web applications. By building a semantic model, we can identify sanitizers not only at the function level but also at the statement level, such as security checking branch statements.

III. Summary and Analysis of Sanitizers

To design the automatic identification method, we analysed the source code of the top 30 Web applications with the most stars on Github. We summarised the sanitizers in these applications. By determining whether the instructions directly modify the data, we divided them into two categories: data transformations and security checks:

A. Data transformation

Web application developers often process data when it is difficult to confirm the security of the user's input method. The program allows some of the input data to contain illegitimate characters, and the data is transformed in the subsequent process to ensure security. We have summarised the data transformations into four types of operations.

a) *Substitution*: As data flows through a program, it is necessary to ensure that the data is separated from the code. Therefore, developers often use substitution operations to replace sensitive characters in the data. This operation involves replacing a character with another string of characters and replacing a character with a null (*delete*). For example, when preventing directory traversal, we can replace the string "../" string with a null character.

b) *Splicing*: To ensure data is secure and controllable, developers often splice other data at the front or end of the data. For example, to prevent file inclusion vulnerabilities, we could splice a path before the original file name or splice a type suffix.

c) *Escape*: In Web application development languages, some characters play a special role. Some characters can truncate statements, close forward data, or even execute commands. For example, the backquote '`' in PHP has the same effect as the '*exec()*' function. Developers often use escape operations for strings. It removes the special meaning of a character and escapes it to a harmless character.

d) *Decoding*: This operation also targets sensitive characters, but it has a wider application than escaping. By encoding, we can convert characters into a more secure form. For example, to prevent XSS vulnerabilities, we can encode "<>" to "*<>*".

For all four of these data transformations, Web applications often implement them by calling functions. For example, the "*htmlspecialchars()*" function encodes the data into HTML format. Therefore, it may seem reasonable to use the library function dictionary for sanitizers recognition. In many cases, however, developers prefer to implement more targeted data transformation functions. It is difficult to match these custom functions with a particular set of rules. In other words, existing recognition methods are not accurate for data transformations. Simple character matching methods are no longer adequate, It is difficult to analyse custom data transformation functions in new Web applications.

The naming rules for custom functions are not uniform across different Web applications. However, we have found that developer naming in a particular Web application often follows a certain specification. This situation is due to the internal specification requirements of the development team. Therefore, it is necessary to use natural semantic analysis for identification. On the other hand, we also found that these data transformation functions have similar semantics. These functions transform the data and return the transformed data. And there are fewer statements in such functions other than data transformations. So we can construct a semantic model of these functions and identify them in relation to the program semantics.

B. Security Check

In addition to transforming the data directly, it is also important to check the data. The developer needs to check the input data to determine whether to continue with the program process or to perform transformations on the data. By analysing these existing Web application, we have divided the security checking semantics into two layers. The first layer is the check statement (conditional branching statement) and the second layer is the security handling operation:

a) Check statement: This consists mainly of conditional branch statements. The data characteristics are analysed in order to select the subsequent branches to be executed. For example, it analyses the data for dangerous strings to determine whether continue to execute the code.

b) Security handling statement: By analysing the data, the program selects a different branch for execution. If it is judged to be normal, the program continues to be executed. If an exception is judged to be present, then the safe handling statement is executed.

Security handling in Web applications can be divided into three categories: data transformations, exception handling functions and exception return codes. Among them, exception handling refers to the function that handles the abnormal situation when the program runs. It can keep the program running normally or protect the system from damage. For example, it can interrupt a program or report an error. Exception return codes are those that return certain characters that mark the program as abnormal.

We have found that the purpose of the security check is to identify whether the input data is legitimate. Once a data exception is caught, it is processed safely and if no exception exists it is executed normally. Therefore, we construct a semantic model of security checking. Assuming that the branch statement is M, and using $\mathrm{M}(\mathrm{C_i})$ to denote a particular one of the branches, the possible cases of the branch statement are:

$$\mathrm{M}(\mathrm{C_i}) = \begin{cases} SH \text{ Security handing instructions} \\ NH \text{ Regular instructions} \end{cases} \quad (1)$$

Then , M is a Security Check if :

$$\begin{gathered} \exists \mathrm{C_i} \text{ Such That } \mathrm{M}(\mathrm{C_i}) = \mathrm{SH} \\ \text{and} \\ \exists \mathrm{C_j}, \text{ Where } \mathrm{i} \neq \mathrm{j}, \text{ Such That } \mathrm{M}(\mathrm{C_j}) = \mathrm{NH} \end{gathered} \quad (2)$$

That is, a conditional statement is a security check if there is at least one branch of the security handling statement and a normal program branch in the branch.

Existing taint analysis is difficult to identify security checks accurately. These methods only recognise security checks with a specific string in the function name, but not at the statement level. As an example in Fig. 3, existing methods can identify the security check function on line 20 using keywords, but can't identify the security check statements on lines 23 to 35. The fact that such security checks are common in applications and also means that the existing identification of sanitizers is heavily underreported. It can be seen that such statements have common semantic features, which means we can combine both

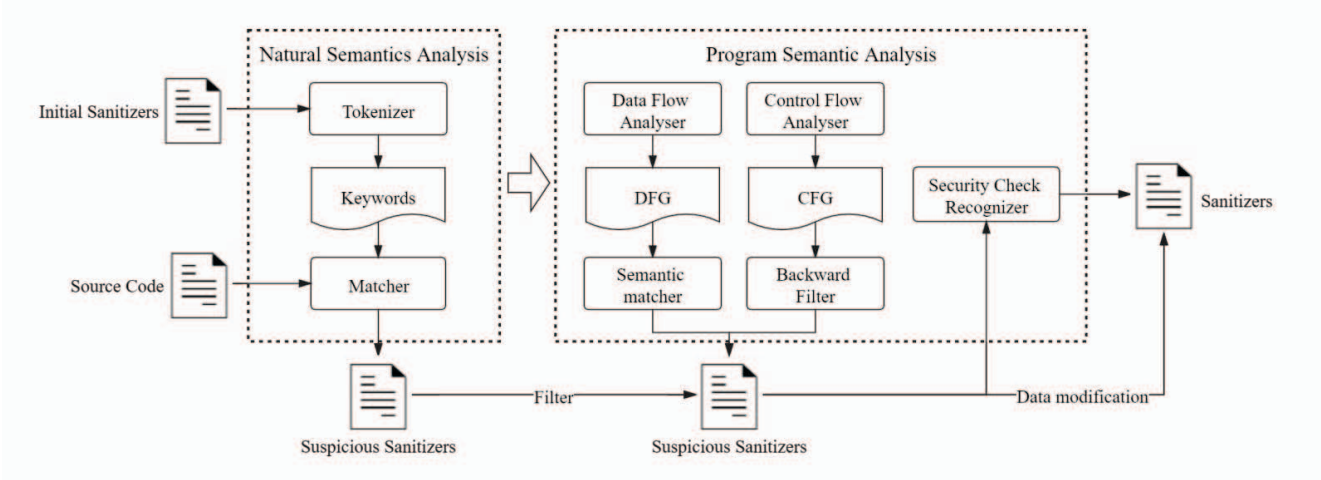

Fig. 4. Process for the automatic identification of sanitizers.

natural semantic and program semantic to identify security checks, thus reducing the rate of misses.

IV. Approach to Identify Sanitizers

In the previous section, we have summarised the sanitizers in Web applications and analysed the shortcomings of existing identification methods. In this section, we describe our approach to identify sanitizers.

A. Approach Overview

The Fig. 4 shows the workflow of our approach. The entire analysis process is divided into two parts: natural semantic analysis and program semantic analysis. It is important to note that, as seen in Section 4, there may be data transforms, exception handling functions and exception return codes nested within the security check. Therefore, for the identification of security checks, we need to identify these three kinds of instructions first, and then identify the security checks through semantic analysis.

Natural semantic analysis: In this step, we perform a preliminary identification of data transformations, exception handling and exception return codes. The functions that come with the programming language can be easily found in the official documentation. In addition, for programming specification reasons, custom functions often use the same strings, such as "*filter*", "*safe*", etc. Therefore, we propose automatic recognition based on natural semantic analysis. We split the function names in the collected base function set and obtain a new set of suspicious keywords based on negative word splitting. This step aims to achieve an intelligent construction of the suspicious keyword set. We can then construct a more complete set of functions and statements for the analysis target.

Program semantic analysis: Based on the semantic model constructed in Section 4, we filter the data transformation functions derived from the natural semantic analysis. We then use the data transformations, exception handling and exception return codes to identify the security checks in the program.

B. Natural Semantics Analysis

Suspicious keyword sets are difficult to define humanly, so identifying suspicious functions by keywords alone will result in a large number of false positives and misses. In addition, the use of prefabricated keywords does not allow for automated construction of keyword sets for emerging Web applications.

To solve this problem, we devised a method to generate keyword sets automatically for specific Web applications. First, we construct an initial set of keywords by splitting the functions and statements that can be identified. Then we use the keywords to fetch suspicious instructions in the application, select feature fields for the suspicious set and generate a new set of keywords.

A major challenge in building keyword sets is that some of the most frequent segments may not be meaningful, such as generic words like "*to*" and "*get*", which are commonly used by developers. To solve this, we analysed the semantics of sanitizers and found that sanitizers often carry some negative semantic fragments (e.g. "*error*", "*fail*", etc.). Therefore, we select high-frequency words with negative semantics as the set of keywords. The natural semantic analysis detection process is divided into the following steps.

(1) We split the initial sanitizer set and using high-frequency words with negative semantics as the keyword set.

(2) Analysing target Web application using the keyword set. In this way, we can get the sanitizers set with the Web application developers' naming feature. Next, we perform a new round of subdivisions of this set to construct the suspicious keyword set.

(3) Finally, we analysis the Web application using the suspicious keyword set. In this step, we will construct the set of suspicious sanitizers.

C. Program Semantics Analysis

There are often some false positives in the results of natural semantic-based recognition. Natural semantic analysis is also unable to identify Security checks. Therefore, we need to combine program semantic analysis to further filter the results,

identify the Security checks and add them to the final sanitizers set.

1) Filtering of data transformations

We find that the data transformation functions used for sanitizers often have similar semantic patterns. These functions have fewer redundant statements other than the transformation operations on the data, and the input values are strongly correlated with the output values. We have defined the semantic model for data transformation functions that fit the characteristics of sanitizers:

- The input to the function, after propagation through the statements within the function, must reach the output of the function (the return value).
- The statements within the function should be directly or indirectly related to the input value, and there is no chain of data propagation within the function that is unrelated to the input value.

Based on the above semantic model, we filter the data transformation operations. The filtering method mainly uses data flow analysis to obtain the semantic characteristics of the suspect function. If the function is judged to satisfy the above two conditions it is retained, otherwise it is removed. The analysis process is as follows:

(1) Traversing the suspect function and analysing the data flow graph of the function.

(2) Determine whether the function has a return value. If so, the return value data propagation chain will be analysed, and if not, the function is removed from the suspect set.

(3) Use backward data flow analysis to determine if the return value is related to the input value. Delete if not relevant, otherwise analyse other data propagation chains.

(4) Determine if there are other data propagation chains that are unrelated to the data propagation chain of the input data. If present then the function is judged not to be sanitizer, if not then it is recorded as a sanitizer.

(5) Continue the analysis for the next suspect function.

2) Filtering of exception handling functions and exception return codes

Both of these are mainly used for the subsequent identification of security checks. By analysing the semantic model of security checks, we can deduce their semantic model in reverse. In a program, the exception handling function and the exception return code used for the security check are located in a branch of the conditional statement. So we can filter them by determining whether they satisfy the semantic model of security checks.

In this step, since we do not have the final set of security checks at this point, we define a semantic model named CSM which is similar to security check model in formula (2).

We assume that the branch statement is N, $N(R_j)$ denotes a particular one of the branch statements, and $\vartheta\left(N(R_j)\right)$ denotes the functions and instructions used in this one branch. In addition, we define the set containing these two classes of code as D, D_i as an element of D, and ND_i as any instruction other than D_i. N satisfies CSM model if :

$$\exists D_i, R_j \text{ Such That } D_i \in \vartheta\left(N(R_j)\right) \quad \text{and} \quad \exists R_k, \text{Where } j \neq k, \text{Such That } ND_i \in \vartheta\left(N(R_k)\right) \tag{3}$$

That is, if there is at least one branch contains D_i and at least one branch does not contain D_i in a branch statement N, then N obeys CSM model. If the semantic environment of a suspect function does not satisfy CSM model, it must not be an exception handling function. It is worth clarifying that this definition cannot be used to determine an exception-handling function or exception-returning code, but it can determine that a code fragment is not either of these.

By using this definition, we can filter out false positives from suspicious collections. To implement this, we need to obtain the calling relationships of the suspicious elements in the program. First, we need to analyse the control flow graph of the obtained program and then analyse each suspicious element in the collection. The process is as follows :

(1) Constructing a control flow graph of the program and traversing the control flow graph.

(2) If a function or instruction in the suspicious set is found to be called in a program, backtrack to find the parent instruction.

(3) Determine if the parent instruction is a conditional statement such as "*IF*" or "*SWITCH*". If so, continue to check the branch.

(4) Check each branch of the conditional statement to determine if it matches the semantic model. If it does, the suspect element will be retained; otherwise it will be deleted.

(5) Continue to analyse the next suspicious instruction call

3) Identifying Security Checks

Once the above identification has been completed, we can identify the security check statements in the program. This step is similar to identifying the exception handling functions. In this step, we use the security check semantic pattern for matching. The analysis flow is as follows:

(1) Construction of a control flow graph and traversal of the control flow graph.

(2) If a data transform, exception handling or exception return code is found to be used in a program, backtrack to find the parent instruction.

(3) Determine if the parent instruction is a conditional statement such as "*IF*" or "*SWITCH*". If so, continue to check the branch.

(4) Check each branch of the conditional statement to determine if it conforms to the security check pattern. If it does, the suspect element will be retained; otherwise delete it.

(5) Continue with the analysis of the next instruction.

V. IMPLEMENTATION

We have implemented a prototype of the above sanitizers identification method in PHP and built a vulnerability detection tool called VulChecker, with reference to the method proposed in [6]. Since the work in this paper focuses on sanitizers identification, the specific implementation of the vulnerability detection tool is not be presented. We now present some interesting implementation details

A. Collecting the initial collection of sanitizers

We are implementing this on PHP, so the sanitizers we need to collect are also PHP-related. The innocuous treatments we collected consisted of three categories.

a) *PHP library functions:* we obtain the officially provided sanitizers functions by reading the official PHP documentation.

b) *Sanitizers in previous work:* the work of previous authors is very informative. We analysed the source code of existing open source tools, including Rips [36], Pixy [4]. We will validate these sanitizers and place the more plausible ones into our initial collection.

c) *Sanitizers in Web applications*: In our analysis of popular PHP Web applications, we have also obtained several new custom sanitizers. For this part, we prefer to select those that are universal rather than unique to a particular Web application.

B. Grading the results

In previous work, we can see that researchers have conducted a large number of studies on the security of sanitizers. If sanitizer is ineffective, there may be security problems in the data dissemination chain that we have ignored. In other words, this will result in a high level of leakage. This cannot be ignored in our detection methods either. In particular, custom sanitizers undergo less actual validation than library functions. Therefore, these sanitizers are more likely to be problematic. To counter this, we rewrote the taint propagation analysis results output and divide the results into three levels.

R1 *Less likely problematic data propagation chains*: We classified data propagation chains that used library functions or other elements from the initial sanitizers set as *R1*.

R2 *Potentially problematic data propagation chains*: Data propagation chains without sanitizers from the initial set, but use sanitizers that we subsequently identify automatically. This category we classify as level *R2*.

R3 *Data propagation chain without sanitizers:* If there is a taint propagation chain in the program that does not use any sanitizers (both in the initial set and in the new set), this chain has a higher probability of being a security problem. We classify this as level *R3*.

The classification of the detection results is not arbitrary but takes into account the different security of the sanitizers in the different collections. In addition, this classification is more user-friendly. For developers, who often require a lower false positive rate, the analysis can be done mainly for *R3*. For security researchers or testers, who often have more time to audit code, auditing both *R2* and *R3* is necessary.

VI. EVALUATION

In this section, we evaluate the effectiveness of VulChecker. and compare the effectiveness of our sanitizers identification with other tools. We use them to identify sanitizers and validate vulnerabilities in popular CMS frameworks. Finally, we give four unreported vulnerabilities that we found.

A. Evaluating sanitizers identification

In this experiment, we analyse SeaCMS [37]. We selected popular open-source PHP taint analysis tools for comparison and analysed the number of sanitizers in the various methods. In addition, we counted the accuracy of the identification by manually verifying the identified sanitizers.

The results are shown in the Table. I, Pixy and Rips use library functions as sanitizers, so we only record the number of sanitizers they use. The column sanitizers in the table indicate the number of sanitizers identified in the results, and we can see that our approach has a clear advantage in identifying the number of sanitizers. By using both natural and procedural semantic recognition, VulChecker is able to recognize both sanitizer functions and statement-level safety checks, so it is able to recognize more sanitizers than other methods.

TABLE I. RESULTS OF COMPARING WITH OTHER TOOLS IN SANITIZERS IDENTIFICATIONS

Tools	Approach	Type	Sanitizers	Accuracy (%)
Pixy [4]	Dictionary	Function	67	/
Rips [36]	Dictionary	Function	91	/
Phos [13]	Dictionary Keyword matching	Function	138	63.4
VulChecker	Dictionary Semantic analysis	Function Statement	153	89.5

Furthermore, it is inaccurate to measure effectiveness by numbers alone, these sanitizers may not be effective. Therefore, we used a manual audit to confirm and calculate the accuracy of the identified methods. The base of the calculation is the total number of sanitizers identified by each type of method, and the numerator is the number of valid sanitizers. As we can see, VulChecker does not just do simple recognition but also makes judgments about the validity of sanitizers through semantic analysis, which leads to higher accuracy. The traditional keyword matching method tends to identify some other functions containing keywords as sanitizers, so it has a lower accuracy rate.

B. Vulnerability detection

To compare the effectiveness of vulnerability detection, we compared VulChecker with existing vulnerability detection tools. The targets are the popular PHP CMS frameworks: SeaCMS, DedeCMS [38], CMSMS(CMS Made Simple) [39] and YoudianCMS [40]. We collect historical vulnerabilities from these four CMSs and use the detection and false positive rates to compare their effectiveness. In addition, to test the effectiveness of VulChecker's rating function, we analysed the results at the *R3* level only and the results at *R2 and R3* together.

We define the vulnerability detection rate as R, the false positive rate as F, the number of vulnerabilities in the method

detection result as CA , the number of successful method verification as CV , and the total number of historical vulnerabilities as HV, then:

$$R = CV / HV \quad (4)$$

$$F = (CA - CV) / CA \quad (5)$$

The comparison of the vulnerability detection results is shown in Table. II.

TABLE II. RESULTS OF COMPARING WITH OTHER TOOLS IN VULNERABILITY DETECTION

Approaches or tools	R (%)	F (%)
Pixy [4]	90.25	47.13
Rips [36]	75.78	55.59
Phos [13]	63.12	40.92
VulChecker (R3)	86.39	17.67
VulChecker (R2&R3)	93.67	34.59

We can see from Table. II that Pixy and Rips, which simply use dictionaries for sanitizers identification, have a high false positive rate of vulnerability identification. Although these two tools identify more results, the overly simple sanitizers identification affects the final analysis results. The Phos method, which uses dictionaries and keyword matching, also suffers from high false positives because the keyword matching is too simple. In addition, Pixy focuses on accurate analysis of alias propagation, so its vulnerability detection rate is relatively high. VulChecker, on the other hand, extends and checks the sanitizers dictionary based on Pixy, combines it with semantic automation to identify sanitizers. Not only can it ensure better vulnerability identification, but also reduce the false positives. Finally, the hierarchy of the results in VulChecker is also effective. The results prove that the hierarchy results can make manual auditing and vulnerability exploitation more efficient for different users.

C. Vulnerabilities we found

To validate the effectiveness of VulChecker in practice, we tested the latest versions of SeaCMS, DedeCMS, CMSMS and YoudianCMS. Through the testing, we found several new suspicious tainted data propagation chains. We validated these propagation chains to find unreported vulnerabilities.

Unfortunately, although we have taken as much time as possible to analyse and submit vulnerabilities, as of the paper submission, we have only four vulnerabilities that have received a public disclosure notice with a vulnerability number. Therefore, we are only presenting these four vulnerabilities at this time and will add further vulnerabilities when they are confirmed. Below is information of the vulnerabilities we found in Table. III, the vulnerability IDs are from CNVD [41] (China National Information Security Vulnerability Sharing Platform) .

TABLE III. THE VULNERABILITIES WE FOUND

Vulnerability ID	Software	Version	Type
CNVD-2021-17446	SeaCMS	V210202	SQL Injection
CNVD-2021-17447	SeaCMS	V210202	SQL Injection
CNVD-2021-26060	SeaCMS	V210202	XSS
CNVD-2021-67961	YoudianCMS	9.2	XSS

VII. CONCLUSION AND FUTURE WORK

In this paper, unlike most researchers who have improved taint analysis by constructing accurate alias propagation analysis, we focus on sanitizers identification. We improve the analysis of taint propagation chains by automatically identifying sanitizers. Our approach enables the construction of more complete and accurate sets of sanitizers, thus improving vulnerability detection. By comparing it with existing vulnerability detection tools, we demonstrate that VulChecker has a lower false positive rate and better detection results. In addition, in our practice, we have found some new suspicious taint propagation chains in some popular PHP CMS frameworks and have successfully identified four 0day vulnerabilities.

However, there is still room for improvement in our work. These will also be the direction of our future research.

(1) *Try to identify more fine-grained data transformations*: In the approach we have built, we identify data transformation functions at the function level. However, there are more fine-grained data transformations in Web applications, such as splices, and character substitution by developers through "for" loops. These statements also play a role as sanitizer in some cases. We will therefore investigate this type of data transformation in our subsequent work to identify this type of sanitizers.

(2) *Determination of the validity of sanitizers*: This paper only discusses the identification of sanitizers, and the analysis of the validity is more lacking. An invalid sanitizer can lead to more false positives and missed positives in vulnerability detection. Therefore, we will conduct a study on the determination of the effectiveness of sanitizers in our subsequent work, and then combine it with taint analysis to achieve more accurate vulnerability detection.

(3) *Extending the application of sanitizer identification method:* As mentioned at the beginning of the paper, the significance of sanitizer identification lies more in its application, so we will subsequently consider its further application in program analysis and security management as well.

ACKNOWLEDGMENT

Thanks to every member of the team who put in a lot of effort into the research. In addition, our mentor, Professor Weiping Wen has provided us with a lot of valuable guidance. This work is supported in part by the National Natural Science Foundation of China (NSFC No. 61872011).

REFERENCES

[1] H. Yuan, L. Zheng, L. Dong, X. Peng, Y. Zhuang, and G. Deng, "Research and Implementation of Security Vulnerability Detection in Application System of WEB Static Source Code Analysis Based on JAVA." pp. 444-452.

[2] G. Wassermann, and Z. Su, "Sound and precise analysis of Web applications for injection vulnerabilities." pp. 32-41.

[3] N. Antunes, and M. J. I. A. o. t. H. o. C. Vieira, "Defending against web application vulnerabilities," vol. 45, no. 02, pp. 66-72, 2012.

[4] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities." pp. 6 pp.-263.

[5] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of JavaScript Web applications in the wild." pp. 61-70.

[6] N. Jovanovic, C. Kruegel, and E. J. J. o. C. S. Kirda, "Static analysis for detecting taint-style vulnerabilities in Web applications," vol. 18, no. 5, pp. 861-907, 2010.

[7] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4F: taint analysis of framework-based Web applications." pp. 1053-1068.

[8] J. Zhao, J. Qi, L. Zhou, and B. Cui, "Dynamic taint tracking of web application based on static code analysis." pp. 96-101.

[9] J. Zhang, C. Tian, Z. J. C. Duan, and Security, "An efficient approach for taint analysis of android applications," vol. 104, pp. 102161, 2021.

[10] J. Galea, and D. Kroening, "The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation." pp. 622-636.

[11] N. Allen, F. Gauthier, and A. J. a. p. a. Jordan, "IFDS Taint Analysis with Access Paths," 2021.

[12] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial SOA applications: a case study at Alibaba." pp. 1477-1486.

[13] C. Skalka, S. Amir-Mohammadian, and S. J. J. o. C. S. Clark, "Maybe tainted data: Theory and a case study," no. Preprint, pp. 1-41, 2020.

[14] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in 28th {USENIX} Security Symposium ({USENIX} Security 19), 2019, pp. 1769-1786.

[15] S. Gan et al., "Collafl: Path sensitive fuzzing," in 2018 IEEE Symposium on Security and Privacy (SP), 2018, pp. 679-696: IEEE.

[16] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 2123-2138.

[17] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in NDSS, 2016.

[18] C. W. Thomas Preston-Werner, P. J. Hyett "Github." http://github.com/

[19] A. Youssef, and A. F. Shosha, "Quantitave dynamic taint analysis of privacy leakage in android arabic apps." pp. 1-9.

[20] B. Xiong, G. Xiang, T. Du, J. S. He, and S. Ji, "Static taint analysis method for intent injection vulnerability in android applications." pp. 16-31.

[21] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller, "Heaps'n leaks: how heap snapshots improve Android taint analysis." pp. 1061-1072.

[22] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for Java Web applications." pp. 140-154.

[23] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." p. 12.

[24] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." p. 110.

[25] L. Wang, F. LI, and L. J. J. o. S. LI, "Principle and practice of taint analysis," vol. 28, no. 4, pp. 860-882, 2017.

[26] 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)." pp. 317-331.

[27] A. Askarov, and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies." pp. 207-221.

[28] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy Web applications." pp. 601-614.

[29] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers." pp. 587-600.

[30] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters." pp. 91-100.

[31] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and Precise Sanitizer Analysis with BEK."

[32] R. Wood. "DVWA: Damn Vulnerable Web Application." https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_r/source/impossible.php

[33] B. Livshits, M. Martin, and M. S. J. T. R. Lam, "SecuriFly: Runtime protection and recovery from Web application vulnerabilities," 2006.

[34] S. Amir-Mohammadian, and C. Skalka, "In-depth enforcement of dynamic integrity taint analysis." pp. 43-56.

[35] B. Livshits, and S. J. A. S. N. Chong, "Towards fully automatic placement of security sanitizers and declassifiers," vol. 48, no. 1, pp. 385-398, 2013.

[36] "RIPS." http://rips-scanner.sourceforge.net/

[37] "SeaCMS." https://www.seacms.net/

[38] "DesDev. "DedeCMS." [28]http://www.dedecms.com/

[39] "CMSMS: CMS Made Simple." http://www.cmsmadesimple.org/

[40] Y. S. T. Co. "YoudianCMS." http://www.youdiancms.com/

[41] Cert. "CNVD: China National Information Security Vulnerability Sharing Platform." https://www.cnvd.org.cn/