Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications

Davide Balzarotti[§], Marco Cova[§], Vika Felmetsger[§], Nenad Jovanovic[*], Engin Kirda[¶], Christopher Kruegel[§], and Giovanni Vigna[§]

[§] University of California, Santa Barbara
{balzarot,marco,rusvika,chris,vigna}@cs.ucsb.edu

[*] Secure Systems Lab
Technical University Vienna
enji@seclab.tuwien.ac.at

[¶] Institute Eurecom
France
engin.kirda@eurecom.fr

Abstract

Web applications are ubiquitous, perform mission-critical tasks, and handle sensitive user data. Unfortunately, web applications are often implemented by developers with limited security skills, and, as a result, they contain vulnerabilities. Most of these vulnerabilities stem from the lack of input validation. That is, web applications use malicious input as part of a sensitive operation, without having properly checked or sanitized the input values prior to their use.

Past research on vulnerability analysis has mostly focused on identifying cases in which a web application directly uses external input in critical operations. However, little research has been performed to analyze the correctness of the sanitization process. Thus, whenever a web application applies some sanitization routine to potentially malicious input, the vulnerability analysis assumes that the result is innocuous. Unfortunately, this might not be the case, as the sanitization process itself could be incorrect or incomplete.

In this paper, we present a novel approach to the analysis of the sanitization process. More precisely, we combine static and dynamic analysis techniques to identify faulty sanitization procedures that can be bypassed by an attacker. We implemented our approach in a tool, called Saner, and we applied it to a number of real-world applications. Our results demonstrate that we were able to identify several novel vulnerabilities that stem from erroneous sanitization procedures.

1 Introduction

Web applications have evolved from simple CGI-based gateways that provide access to back-end databases into full-fledged, complex applications. Such applications (e.g., email readers, web portals, or e-commerce front-ends) are developed using a number of different technologies and frameworks, such as ASP.NET [21] or PHP [32]. Even though these technologies provide a number of mechanisms to protect an application from attacks, the security of web applications ultimately rests in the hands of the programmers. Unfortunately, these programmers are often under time-to-market pressure and not always aware of the available protection mechanisms and their correct usage. As a result, web applications are riddled with security flaws that can be exploited to circumvent authentication, bypass authorization checks, or access sensitive user information. A report published by Symantec in March 2007 states that, out of the 2,526 vulnerabilities that were documented in the second half of 2006, 66% affected web applications [42].

One of the most common sources of vulnerabilities is the lack of proper validation of the parameters that are passed by the client to the web application. In fact, OWASP's *Top Ten Project*, which lists the top ten sources of vulnerabilities in web applications, puts unvalidated input as the number one cause of vulnerabilities in web applications [30]. Input validation is a generic security procedure, where an application ensures that the input received from an external source (e.g., a user) is valid and meaningful. For example, an application might check that the number of items to purchase, sent as part of a form submission, is actually provided as an integer value and not as a non-numeric string or a float. As another example, an application might need to ensure that the message submitted to a bulletin board does not exceed a certain length or does not contain JavaScript code. Also, a program typically has to enforce that arguments to database queries do not contain elements that alter the intended meaning of these queries, leading to SQL injection attacks.

A particular type of input validation is *sanitization*. In general, sanitization is performed to remove possibly malicious elements from the input. Section 2 introduces more examples of how sanitization is performed in web applications. At this point, it suffices to say that sanitization is performed on external input parameters *before* they are used in critical operations. The lack of sanitization can introduce vulnerabilities (e.g., cross-site scripting (XSS) [20] and SQL injection [1,39] flaws) that can be exploited by attackers. A number of past research efforts [9, 13, 17, 18, 22, 45] have focused on the problem of identifying vulnerabilities in which external input is used without any prior sanitization. These vulnerability detectors are typically based on some form of data flow analysis that tracks the flow of information from the application's inputs (called *sources*) to points in the program that represent security-relevant operations (called *sinks*). The underlying assumption of these approaches is that if a sanitization operation is performed on all paths from sources to sinks, then the application is secure.

Interestingly, there has been little research to precisely model how effective the sanitization process actually is. In fact, most approaches assume that if a regular expression or a certain, built-in sanitization function is applied to an external input, then the result is safe to use. Unfortunately, this is not always the case. For example, a regular expression could be used by a programmer to check for the occurrence of certain values in the input without any intention to perform sanitization. Also, it is possible to apply a sanitization function to the input that protects from certain malicious values, but does not offer complete protection from all attacks. For example, in [41], the authors discuss the possibility of subtle SQL injection vulnerabilities that can be exploited even when the input has been processed by a built-in PHP sanitization routine. Sanitization is particularly dangerous when custom checking routines are used. In these cases, a programmer does not rely on built-in input validation functions, but, instead, manually specifies a list of unwanted characters or a regular expression that should remove malicious content.

In this paper, we introduce a novel approach to analyze the correctness of the sanitization process. The approach combines two complementary techniques to model the sanitization process and to verify its thoroughness. More precisely, a first technique based on static analysis models how an application modifies its inputs along the paths to a sink, using precise modeling of string manipulation routines. This approach uses a conservative model of string operations, which might lead to false positives. Therefore, we devised a second technique based on dynamic analysis. This approach works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process.

We implemented our techniques in a system called Saner, a prototype that analyzes PHP applications. The choice of PHP was driven by the fact that PHP is one of the most popular languages for web application development. According to the latest Netcraft survey [27], more than 20 million sites were using PHP in June 2007. In the monthly Security Space Reports [36], PHP has constantly been rated as the most popular Apache module over the last few years. To evaluate our system, we used Saner on a set of real-world applications. The results show that the sanitization process is faulty in a number of cases and that apparently effective sanitization routines can be bypassed to exploit the applications.

The contributions of this paper are the following:

- We describe a static analysis technique that characterizes the sanitization process by modeling the way in which an application processes input values. This allows us to identify cases where the sanitization is incorrect or incomplete.
- We introduce a dynamic analysis technique that is able to reconstruct the code that is responsible for the sanitization of application inputs, and then execute this code on malicious inputs to identify faulty sanitization procedures.
- We compose the two techniques to leverage their advantages and mitigate their disadvantages.
- We implemented our approach and evaluated the system on a set of real-world applications. During our experiments, we identified a number of previously unknown vulnerabilities in the sanitization routines of the analyzed programs.

The rest of the paper is structured as follows. In Section 2, we provide an example of the type of errors in the sanitization process that we are interested in identifying. In Section 3, we present our techniques for the analysis of the sanitization process in web applications. Then, in Section 4, we describe a prototype implementation of our approach and the results of its evaluation on real-world applications. Section 5 presents related work. Finally, Section 6 concludes and outlines future work.

2 Motivation

In this section, we discuss in more detail the ways in which web applications can perform input validation. This discussion also helps to establish a more precise notion of sanitization. Then, we provide an example of a custom sanitization routine that is typically not handled by static vulnerability detectors. This demonstrates the need for an improved analysis process and serves as an underlying motivation for our work.

2.1 Input Validation and Sanitization

Web applications typically work by first reading some input from the environment (either provided directly by

a user or by another program), then processing this data, and finally outputting the results. As previously stated, the program locations where input enters the application are referred to as sources. The locations where this input is used are called sinks. Of course, sources often take data directly from potentially malicious users, and the application can make little (or no) assumptions about the values that are supplied. Unfortunately, many types of sinks cannot process arbitrary values, and security problems may arise when specially crafted input is passed to these sinks. We refer to these sinks as *sensitive* sinks.

An example of a sensitive sink is a SQL function that accesses the database. When a malicious user is able to supply unrestricted input to this function, she might be able to modify the contents of the database in unintended ways or extract private information that she is not supposed to access. This security problem is usually referred to as a SQL injection vulnerability [1]. Another example of a sensitive sink is a function that sends some data back to the user. In this case, an attacker could leverage the possibility to send arbitrary data to a user to inject malicious JavaScript code, which is later executed by the browser that consumes the output. This problem is commonly known as an XSS vulnerability [20].

To avoid security problems, an application has to ensure that all sensitive sinks receive arguments that are well-formed, according to some specification that depends on the concrete type of the sink. Because input from potentially malicious users can assume arbitrary values, the program has to properly validate this input. Therefore, the application checks the input for values that violate the specification. When such invalid values are found, a programmer has two options. The first option is to abort further processing: the application stops to handle the request and returns an error code to signal incorrect input. The second option is to transform the input value such that the altered value conforms to the input specification and no longer poses a security threat when passed to a sensitive sink. We denote the process of transforming the input to a representation that is no longer dangerous as *sanitization*. Typically, sanitization involves the removal of (meta)-characters that have a special meaning in the context of the sink, escaping these characters, or truncating the length of the input.

2.2 Static Analysis and Proper Sanitization

Static analysis tools that check the security of (web) applications often employ data flow analysis to track the use of program inputs. The goal of these systems is to identify program paths between the location where an input enters the application and a location where this input is used. Once such a program path is identified, the tool checks whether the programmer has properly sanitized the input on its way from the source to the sensitive sink. When input is properly sanitized on all paths from an input source to a sensitive sink, the application is correct and does not contain a security vulnerability. However, it is unfortunately not immediately obvious when to declare input as properly sanitized.

The first problem is that the input sanitization depends on the type of sink that consumes the input. For example, when an attacker can inject SQL commands into the output that the application sends back to a user, this application is not vulnerable. A security problem arises only when the attacker can inject that same input into a function that accesses the database. As a result, static analysis tools typically require a policy that specifies for each type of sensitive sinks (such as database access or output functions) the set of operations that constitute proper sanitization.

The second problem that makes it hard to ascertain proper sanitization is the difficulty of specifying all sanitization operations *a priori*. Fortunately, many languages provide built-in functions that sanitize input. For example, the PHP function `htmlentities` converts characters that have special meaning in HTML into their corresponding HTML entities (e.g., the character '<' is converted into '`<`'). This ensures that all characters in a string preserve their meanings when interpreted as HTML. The PHP manual states that this function is useful "in preventing user-supplied text from containing HTML markup." Applying `htmlentities` to an input string ensures that the resulting string can be safely sent back to a user. The reason is that all script tags in the input (such as "`<script>`") are converted into tokens that are no longer interpreted by a browser as the start of JavaScript code, but simply displayed as the string "`<script>`." Of course, it is easy to recognize the use of such functions as proper sanitization.

Besides the use of built-in sanitization functions, a programmer can also write *custom code* that strips dangerous characters from an input string. For example, the programmer could apply the PHP function `str_replace` to the input string and remove all occurrences of the character '<' (more precisely, to replace all occurrences of the angle bracket character with the empty string). In this case, the result would also be safe with respect to XSS and could be sent back to the user. To see the difference between standard and custom sanitization, consider the example code in Figure 1. Of course, the mere fact that the programmer applies a string replacement operation on an input value does not ensure that the result is properly sanitized.

```
1 $input = _GET['x'];
2
3 $standard = htmlentities($input);
4 $standard = 'Hello ' . $standard;
5 echo $standard;
6
7 $custom = str_replace('<', '', $input);
8 $custom = 'Hello ' . $custom;
9 echo $custom;
```

Figure 1. Standard and custom sanitization.

When analyzing the code in Figure 1, most static analysis tools would correctly flag the use of variable

`$standard` in Line 5 as safe. The reason is that they are typically equipped with a policy that specifies that all values processed by `htmlentities` can be safely echoed back to a user. The situation is more complicated when analyzing the use of the function `str_replace`, which performs custom sanitization in this case. In principle, static analyzers could interpret the application of any function that modifies an input (e.g., through string replacement) as an indication that the programmer performed sanitization. If this strategy is used, the analysis would correctly consider the application of the `str_replace` function on Line 7 as a form of sanitization. Of course, this approach suffers from two drawbacks. First of all, the programmer might have simply applied this function to alter the string based on some requirement implied by the application logic, and changing the string does not imply that the result is safe to be used by a sensitive sink. Second, the programmer could have made a mistake. Even when the input is modified with the intention to make it safe, there is no guarantee that the result is correct (and our results demonstrate that programmers do make frequent mistakes when using custom sanitization routines). Also, the opposite strategy of assuming that all custom sanitization operations are incorrect is problematic, because it causes static analysis tools to report incorrect warnings in case a programmer has correctly applied custom sanitization.

Current static analysis systems (see Section 5 for a detailed discussion of related work) typically disregard the use of custom sanitization routines. The result is that whenever a programmer makes use of custom sanitization, these tools report an error. This requires a tedious, manual inspection of the false positives. Of course, once a sanitization routine has been manually examined and annotated as safe, more powerful static analysis tools will honor this annotation and no longer report false positives. Unfortunately, programmers are often unlikely to spot the application of incorrect custom sanitization. The reason for this is that programmers *expect* the static analysis tools to report an error in association with their custom sanitization routines, and, therefore, there is little need to double-check them. This is dangerous, as we have found several instances in a number of real-world programs in which custom sanitization was used incorrectly.

To address the shortcomings of current analysis tools, we propose a technique that can handle the use of custom sanitization routines and properly track the effect of functions that manipulate and modify program input. The goal is to model the effect of sanitization routines so that we can check, for every sensitive sink, whether the input that can reach these sinks might contain malicious values. This solves two problems. First, we can reduce the number of false positives produced by current static analysis tools by taking into account correct sanitization. Second, we can identify incorrect sanitization routines and alert the programmer when she has made a mistake.

3 Approach

The goal of Saner is to analyze the use of custom sanitization routines to identify possible XSS and SQL injection vulnerabilities in web applications. In the context of our work, any function that takes as input a (string) value and that can output a modified version of this input is considered a possible sanitization routine. In particular, this includes functions that replace or remove certain characters or substrings from their input (such as the PHP functions `str_replace` or `eregi_replace`). As mentioned previously, this requires our system to model the ways in which these functions can modify the application's input. To this end, we use a combination of static and dynamic program analysis techniques.

The core of the approach consists of a static analysis component that uses data flow techniques to identify the flows of input values from sources to sensitive sinks. This component is based on the open-source web vulnerability scanner called Pixy [17, 18]. In its current form, Pixy only provides information about the presence of data flows between sources and sinks. In addition, it can determine whether built-in sanitization operations (such as `htmlentities`) are applied on all paths between a source and a sink. To achieve this, it is sufficient to assign one of two types (or labels) to each program variable: *tainted* or *untainted*. Whenever input is read from a user and stored in a variable, the variable initially receives the label *tainted*. Once a variable is sanitized, its label is set to *untainted*. Whenever a *tainted* variable is used in a sensitive sink, an error is signaled. Unfortunately, this simple approach cannot model the effect of sanitization routines, as a variable can only be *tainted* or *untainted*, and the tool cannot capture the set of values that the variable can hold. To address this problem, we have extended Pixy to derive an over-approximation of the set of (string) values that each program variable can hold. This calculation is done for every point in the program. For each sensitive sink, we can then check whether this value set contains any element that poses a security risk when used at that sink.

The static analysis component is sound with respect to the supported language features[1]. That is, whenever the static analysis component declares a sanitization operation to be correct, we are certain that there exists no vulnerability. The drawback of this approach is that the system might produce false positives (i.e., not every reported problem is an actual vulnerability). Because the number of false positives can be large (depending on the application), we augment the static analysis with an additional dynamic analysis phase.

The goal of the dynamic phase is to examine all those program paths from input sources to sensitive sinks that the static analysis has identified as suspicious. More precisely, using dynamic analysis, we attempt to confirm the existence of a potential security vulnerability (reported by

[1]Most notably, our analysis does not support the `eval` function and certain cases of aliased array elements.

the static analysis phase) by finding program inputs that can bypass the sanitization routines and reach the sensitive sink. To this end, the dynamic analysis is used to simulate the effect of the program operations on the input while it is propagated to the sensitive sink (in particular, sanitization operations are of interest). Of course, the analysis is performed by exercising the code with a large set of different input values, which contain many different ways of encoding and hiding malicious characters. In some sense, the dynamic analysis phase automates the actions of a programmer when a static analysis tool reports a warning. Similar to our dynamic phase, the programmer would first identify the operations that are applied to an input on the path from the source to the sink. Then, using a number of test cases, she would attempt to understand whether one of these inputs could lead to a security violation.

Whenever the dynamic analysis phase determines that a malicious value can reach a sensitive sink, this input is reported as a concrete example that violates the security of the application. If no such input can be found, there are two options. The first option is to assume that the static analysis phase has incorrectly flagged a correct sanitization routine as suspicious. This sacrifices soundness because the dynamic analysis might miss a true vulnerability, but it is convenient as no further manual inspection is required. The second option is to report confirmed vulnerabilities with higher confidence, and to yield the remaining warnings to the programmer.

3.1 Sanitization-Aware Static Analysis

As mentioned in Section 2, existing static analysis approaches simply "guess" whether a custom sanitization routine is effective or not. A sound analysis would regard all types of custom sanitization as ineffective, which typically leads to many false positives. An unsound analysis would assume that custom sanitization is always correct, which may result in missed vulnerabilities. Pixy, the analysis tool that we build upon, follows a sound approach. Hence, our first goal is to improve the existing static analysis so that it is able to assess the effectiveness of custom sanitization routines. As a result, whenever our improved analysis verifies that a custom sanitization is correct, it has essentially suppressed a false positive that (the original) Pixy would have reported. To achieve this goal, we present a technique that leverages *transducer-based, implicit taint propagation*. Once the static analysis is finished, our second objective is to provide the subsequent dynamic analysis phase with appropriate information that allows for further inspection of all suspicious sanitization procedures.

3.1.1 Basic String Automata

As a first step for modeling sanitization routines, we require information about the *set of values* that different program variables may hold, not only the information whether they are tainted or not. To this end, we employ an analysis that can approximate the string values that certain variables might hold at certain program points, using finite automata. Commonly, automata are used as acceptors. That is, they are applied for deciding whether string values belong to a certain language. For our purposes, we make use of another property of automata (or, equivalently, regular expressions), namely the ability to describe an arbitrary set of strings.

In our automata representation, every edge denotes a single-character transition. Note that the label $\langle . \rangle$ stands for an arbitrary character. In addition to the characters that make up a string value, automata also need to be able to encode information about the taint status of these stings. That is, we want to be able to express that certain parts of a string value are tainted, whereas other parts are untainted. This allows us to reconstruct which parts of the strings are possibly derived from (malicious) user input, and which parts stem from static strings embedded in the program source code. This property is achieved by associating taint qualifiers to the transitions of the automata. An edge can either represent a tainted character (represented by a *dotted* line) or an untainted character (represented by a *solid* line).

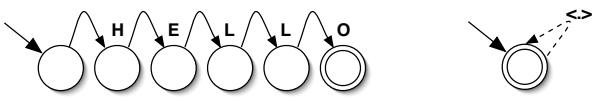

Figure 2. Automata for the string "Hello", and for an unknown, tainted string.

As example, two automata are shown in Figure 2. The left automaton represents the static string value "`Hello`", and hence, contains a series of transitions labeled with the individual characters of this string. Since the string derives from a static literal provided by the programmer, it is considered to be untainted, which is represented by solid transitions. In contrast, the automaton on the right side could represent the value set for variable `$_GET['x']`. Because this value is user-supplied and not known until runtime, the automaton describes the set of all possible strings. The dotted transition indicates that the value is tainted.

Dependence Graphs. To compute the set of string values that a variable at a certain program point can hold, we leverage Pixy's dependence analysis. Dependence analysis is a data flow analysis that computes a dependence graph for every program point (and each variable). Such dependence graphs provide reaching definitions for a particular program point. Intuitively, this means that a dependence graph provides a list of all variables (program points) that might directly influence (or reach) the current program point. As an example, consider the dependence graph in Figure 3. This graph reflects the dependencies for variable `$custom` on Line 9 in Figure 1.

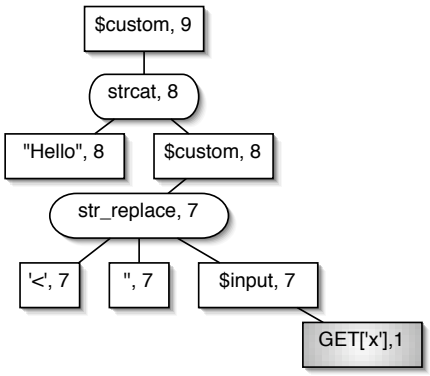

Figure 3. Example dependence graph.

```
1  decorate(Node n) {
2    decorate all successors of n;
3    if n is a string node:
4      decorate n with an automaton for this string
5    else if n is an <input> node:
6      decorate n depending on type of input
7    else if n is an operation node:
8      simulate the operation's semantics
9    else if n is a variable node:
10     decorate n with the union of n's successor
11     automata
12   else if n is a SCC node:
13     decorate n with a star automaton
14     (the taint value of its transition depends
15     on the successor nodes)
16 }
```

Figure 4. Dependence graph decoration algorithm.

Computing Automata. Assume for the moment that a dependence graph for a certain variable (at a particular program point) does not contain cycles. In this case, we can compute the automaton for this variable by applying the algorithm shown in Figure 4. This algorithm takes the root of the dependence graph as input, and recursively processes all nodes of the graph in a postorder traversal. During this traversal, each processed node is associated (*decorated*) with a separate automaton. Each of these automata describes the possible string values of the corresponding node. For computing such an automaton, the automata of all successor nodes are required as input, which explains the postorder traversal. Once all successors of a node have been successfully decorated, the way how the current node is decorated depends on the type of this node.

In the simplest case, the current node represents a string literal. Such nodes are simply decorated with an automaton that describes exactly this string. For a program ⟨`input`⟩ node (shown shaded in Figure 3), it is necessary to analyze the type of this input. If the node represents a variable whose value is taken from the user, such as `$_GET['x']`, the automaton shown on the right in Figure 2 is used for decoration. This automaton represents the set of all possible strings (we will refer to such an automaton as "star automaton" from now on). Its sole transition is tainted to reflect the fact that the user input can be malicious. If the input cannot be directly controlled by a remote user, the transition would be untainted.

If the node to be processed is an operation node (that is, a call to a built-in function), then the semantic of this operation has to be simulated. In case of a string concatenation operation (represented as `strcat`), this is simply done by concatenating the automata of the successor nodes. All other operations are divided into two groups (and our system is equipped with a list that assigns built-in functions to one of these categories). The first group contains functions that are precisely modeled. That is, our system is able to compute an automaton that describes all possible output strings, even when the input parameters to the function are not concrete string instances but automata as well. This is realized with the help of *transducers*, which are described in more detail in the following Section 3.1.2. We have developed a number of transducers for functions that manipulate strings (such as `str_replace`) as well as functions that are commonly used for input sanitization (such as `html_entities`). This is essential to be able to precisely capture the effect of sanitization routines.

The second group of operations contains functions that are not modeled. In this case, we resort to a conservative approximation and assume that each function returns the set of all possible strings (represented as the star automaton). The taint status of the automaton's transition depends on the taint status of the function's actual parameters. To be more precise, the taint status for such functions is the *least upper bound* over the taint status of their parameters. That is, if any parameter is tainted, then the return value is tainted as well. Otherwise, the return value is not tainted.

The next case in the algorithm of Figure 4 (Line 9) applies if the current node is a node representing a variable. In a dependence graph, the successor nodes of a variable node represent the values that this variable *may* possess. Different successor nodes correspond to different paths through the program. This fact can be translated into an automaton by creating the union of the successor nodes' automata. For example, if a variable `$a` depends on the two string literals "`b`" and "`c`", it means that `$a` can hold one of these two strings at runtime. An automaton that encodes this information is created by computing the union of the two automata that represent "`b`" and "`c`", respectively.

Cyclic Dependence Graphs. The previously described algorithm for transforming dependence graphs into automata is not directly applicable to graphs that contain cycles. In general, the precise modeling of cyclic string operations is a difficult problem [4, 24]. Our solution is to replace strongly connected components (SCCs) in the dependence graph with special SCC nodes, which results in a dependence graph without cycles. This explains the final Lines 12 to 15 of our decoration algorithm in Figure 4. Here, SCC nodes are treated analogously to built-in functions that are not modeled. That is, they are decorated with a star automaton, and the taint value of this automaton's

transition is given by the taint values of the SCC node's successors.

Discussion. In general, strings can be created or modified by one of the following three methods: the use of string literals (e.g., `$s = 'ab'`), the concatenation of two strings, or the use of a built-in function. The use of the first two methods, string literals and string concatenation, are always handled precisely by our algorithm. This is also true for built-in functions that are modeled by transducers. Built-in functions that are not explicitly modeled are conservatively approximated with a star automaton. As a result, our analysis either computes precise results, or a safe approximation of the actual result. The only exception is that we do not handle the manipulation of strings through indexing. For instance, it is possible to change the value of the third character of some string variable `$s` through an assignment to `$s{2}`. While this technique for modifying strings is common in C programs, it occurs rarely in PHP applications. In fact, all applications that we evaluated for this paper did not make use of index-based string modifications.

3.1.2 Precise Function Modeling

As mentioned previously, for the analysis of custom sanitization, it is necessary to introduce a precise modeling of string-modifying functions (such as `str_replace`) and replacement functions using regular expressions (`ereg_replace` and `preg_replace`). A suitable algorithm was presented in the natural language processing community by Mohri and Sproat [25]. This algorithm is based on the use of *finite state transducers*. A transducer is an automaton whose transitions are associated with output symbols. This way, it is not only able to accept (or reject) input strings, but it also produces output for each input string.

For example, when using Mohri and Sproat's algorithm to analyze the string operations on Lines 7 and 8 in Figure 1, we obtain the automaton shown in Figure 5. This automaton precisely captures the possible values of the variable `$custom`. That is, it describes the set of strings that start with the prefix "`Hello`", and end with a suffix that does not contain the '<' character. Unfortunately, the computed automaton does not distinguish between tainted and untainted transitions anymore. Instead, it simply assumes all transitions to be untainted. This is because Mohri and Sproat's algorithm is not designed to work on taint-aware automata. We will present a solution to this problem later in this section.

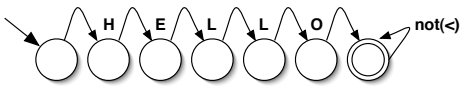

Figure 5. Automaton after replacement of '<'.

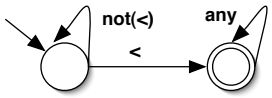

Figure 6. Example target automaton for XSS.

3.1.3 Vulnerability Detection Through Intersection

To check whether a program is vulnerable at some sensitive sink (even when sanitization routines are previously applied), it is necessary to determine whether it is possible that the input to this sensitive sink contains any malicious characters or strings. For instance, an XSS attack typically requires characters such as '<' to be present, as they are needed to construct JavaScript or HTML code. In our approach, we verify this requirement by intersecting the automaton that represents the sink's input with an automaton that encodes the set of undesired strings (the *target automaton*). If the automaton that results from this intersection is empty[2], it means that none of the undesired strings can be contained in the input, and that this sink is safe.

We are aware of the fact that certain XSS attacks do not require the attacker to inject a '<' character into the program output. For example, if an application lets a user modify HTML tags (such as CCS properties or fonts), an attacker could inject script handlers into tag attributes. In such cases, the automaton that captures such attacks will need to be more complex. This can be done without requiring any modifications to our basic technique. Moreover, by checking for the presence of the '<' character, our system already covers a significant fraction of existing XSS threats.

A simple example automaton that represents a conservative approximation of the undesired strings with respect to XSS is shown in Figure 6. This target automaton represents all strings that contain at least one '<' character. Intersecting this automaton with the automaton from Figure 5 yields an empty automaton, which means that this input cannot be used to successfully perform an attack. By doing this, we have successfully determined that the applied custom sanitization was effective. In contrast, the intersection of the target automaton with the automaton that represents the potentially dangerous value of variable `$_GET['x']` (right side of Figure 2) is non-empty, since the unknown value might contain an arbitrary number of '<' characters.

3.1.4 Implicit Taint Propagation

Unfortunately, the techniques for function modeling and vulnerability detection described above are still lacking an important ingredient for reaching a sufficient level of precision. The reason is that the algorithm of Mohri and Sproat

[2]To be precise: If the resulting automaton accepts only the empty language.

does not operate on taint-aware automata, but, instead, on traditional automata without taint qualifiers associated to their transitions. That is, the algorithm is not able to propagate taint values through the modeled functions. Without additional measures, this information loss would lead to false positives, as taint information is essential for vulnerability detection. For instance, the simple example code depicted in Figure 7 below would result in a false positive.

```
$s = "Hello\n";
$x = str_replace("\n", '<br/>', $s);
echo $x;
```

Figure 7. Code that causes a false positive.

```
$s = 'a';
$x = str_replace('a', $_GET['x'], $s);
echo $x;
```

Figure 8. Code that could cause a false negative.

In this code, all occurrences of the '\n' control character are replaced with an HTML line break. Intersecting the automaton that is computed for $x with the XSS target automaton would yield a non-empty result, since $x does contains a '<' character. As a consequence, our analysis would report a vulnerability for this example.

A possible approach to solve the problem of propagating taint values through custom sanitization functions would be to modify Mohri and Sproat's algorithm such that it becomes taint-aware. This modification would ensure that the algorithm accepts taint-aware automata as input (i.e., the arguments of the modeled sanitization function), and returns a taint-aware automaton as output. However, we propose an alternative solution that is sound, efficient, less complex, and less error-prone than a modification of the existing algorithm. Instead of explicitly keeping track of both tainted and untainted values, we concentrate our attention on the tainted parts of the automata. In this *implicit taint propagation*, strings that are statically embedded into the application by the programmer (and hence, untainted) are replaced by the empty string during the automata computation. This has the effect that only tainted strings are explicitly encoded in the automata, and that static, untainted strings can no longer lead to false positives.

If used without care, however, implicit taint propagation can cause false negatives in certain cases (i.e., vulnerabilities might be missed). Consider the (rather contrived) example in Figure 8. Here, the program replaces the character 'a' inside the string $s with a user-provided value (taken from $_GET['x']). If our analysis would propagate taint values implicitly (and replace the value of $s with the empty string), it would incorrectly deduce that the str_replace operation results in the empty string as well. Under the XSS target automaton defined above, an empty string is benign, and, therefore, the vulnerability would be missed. To ensure soundness, it is necessary to compensate the information loss due to the implicit taint propagation with a supplementary "safety net." This additional mechanism corresponds to checking whether the second parameter of str_replace (or, analogously, the replacement parameter of similar functions) is tainted. If the parameter is tainted, the result of the function invocation is conservatively approximated with the automaton that describes the set of all possible strings. This ensures that implicit taint propagation does not introduce false negatives.

3.1.5 Providing Information to Dynamic Analysis

The dynamic analysis that follows the static analysis phase is focused on the detection of routines that perform insufficient custom sanitization. Hence, it only requires information about those vulnerabilities reported by the static analysis process that actually involve the use of custom sanitization routines. For instance, consider the following example code:

```
1 $x = str_replace('<script>', '', $_GET['x']);
2 echo $x;
```

In this program, the variable $_GET['x'] is insufficiently sanitized. The programmer attempted to remove all script tags from the input. Unfortunately, this simple sanitization technique can be easily circumvented by embedding JavaScript code in HTML event handlers (such as onload), which do not require the use of script tags. When checking this code, the dynamic analysis should be informed that there is a possible vulnerability due to insufficient sanitization, and that this vulnerability involves the user-controlled variable $_GET['x'] as source and the echo statement on Line 2 as sink. That is, the dynamic analysis is provided with *source-sink pairs* that describe possible vulnerabilities due to insufficient custom sanitization. This information is extracted from the dependence graphs that static analysis uses internally by means of a simple reachability computation.

Recall that the focus of this paper is on the analysis of custom sanitization routines. As a consequence, we do not provide dynamic analysis with information about vulnerabilities that do not involve any custom sanitization. Instead, these vulnerabilities are immediately reported to the user.

3.2 Testing Sanitization Routines

The static analysis phase is conservative, and, therefore, it may generate false positives, which a developer needs to manually assess. This process, however, is tedious and error-prone. The goal of the dynamic analysis (or testing) phase is to automate this task, or at least, to automatically confirm vulnerabilities for which inputs can be found that bypass sanitization functions.

During the dynamic phase, we test the effectiveness of the sanitization routines applied along the paths between a source and the corresponding sink. This is done by directly

executing the corresponding sanitization routines, using as input a number of attack strings. Then, a decision function (typically called an " oracle" in a testing context) is used to evaluate whether the attack strings were successfully reduced to non-malicious values. If the testing process confirms that the sanitization is actually ineffective between a source and a sink, it also provides the path along which malicious input can reach the sink, as well as a sample attack string that successfully exploits the vulnerability. This information can then be leveraged by the developer to identify and fix the problem.

Ideally, one would run dynamic tests on a live installation of the application. However, it is often the case that a vulnerability may be exploited (and, therefore, discovered through testing procedures) only if the application is in a certain, well-defined state (e.g., after the administrator has logged in and the database contains specific values). Unfortunately, it is very difficult to test an application under these conditions in a completely automated fashion. Therefore, we take a different testing approach that focuses only on the sanitization process itself and abstracts away all other details of the application.

The dynamic analysis phase is composed of two different steps. First, we construct a *sanitization graph* for each pair of sources and sinks that are provided by the static analysis component. Conceptually, we model the sanitization process as the execution of a sequence of primitive operations, i.e., sanitization functions. The sanitization graph is the data structure that we use to efficiently store the sequences of sanitization operations that are applied to the input along all paths from a source to its sink. The second step of the dynamic analysis uses the sanitization graph to test the corresponding sanitization code using a number of predefined test cases.

3.2.1 Extracting the Sanitization Graph

For a given pair of source and sink nodes, the sanitization graph is a slice (subgraph) of the interprocedural data-flow graph of the application. This slice contains all nodes that correspond to sanitization instructions along the paths from the source to the sink. More precisely, we first compute the interprocedural data-flow graph of the program between the source and the sink nodes. By definition, each node of the resulting subgraph represents an operation that affects the contents of the variables used by the sink. This graph is then simplified by keeping only those nodes that correspond to statements relevant to the sanitization process. These include all calls to language-provided sanitization routines (such as `strip_tags` and `htmlspecialchars`), regular-expression-based substitution functions (`preg_replace`), string-based substitutions (e.g., `str_replace`, `strtoupper`), as well as other built-in string operations (e.g., concatenation).

As an example, consider the code snippet shown in Figure 9. The static analysis of the program identifies that a user-provided input on Line 11 (the source) can reach the

```
1  <?php
2  function sanitize($data){
3    $res = eregi_replace("<script", "", $data);
4    if(version_compare(phpversion(),"4.3.0")=="-1")
5      $res = mysql_escape_string($res);
6    else
7      $res = mysql_real_escape_string($res);
8    return $res;
9  }
10
11 $name = sanitize($_GET["username"]);
12 echo "Name: ".$name;
13 ?>
```

Figure 9. Customized sanitization function.

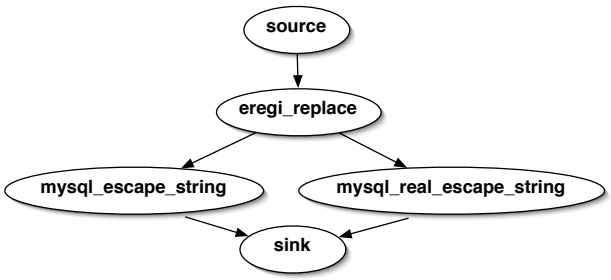

Figure 10. Sanitization graph.

`echo` statement at Line 12 (the sink) without proper sanitization. Following the data flow edges, we identify three operations that affect the content of the variable `$name` that is used by the sink: `mysql_real_escape_string` at Line 5, `mysql_escape_string` at Line 7, and the regular expression at Line 3. The nodes corresponding to these three functions are connected according to the edges in the data flow graph, to reflect the order in which the operations are performed in the original program. Figure 10 shows the resulting sanitization graph for our simple example.

3.2.2 Testing the Effectiveness of the Sanitization Routines

To analyze the effectiveness of the sanitization operations performed by the application, we use the sanitization graph to extract all possible paths P_i that lead from the source to the sink. In our experiments, we found that the sanitization graph is usually acyclic. However, in order to avoid possible paths of infinite length, we adopted the common solution of traversing each loop only once. For each path P_i, we generate a block of code C_i by concatenating the PHP instructions that correspond to each node that belongs to P_i. This operation may involve resolving values of variables to determine (or extract) parameters used in a function call. In our example, the call to `eregi_replace` is examined to fetch the constant values of the first and second parameter. The set of all the blocks C_i corresponds to all the possible combinations of sanitization operations that are applied by the program between the source and the sink.

Since the sanitization graph only considers data flow information, it is possible that our analysis generates code corresponding to infeasible paths, i.e., paths that cannot be executed at runtime. This can result from the fact that the sanitization graph does not model branch conditions, and, therefore, the dynamic analysis might consider branches that, in the original program, are infeasible. In case user input is not properly sanitized along an infeasible path, the tool might generate a false positive. This would occur if proper sanitization is performed on all other, feasible paths. However, we have not observed such a case in our experiments. Also, note that even though we remove all cycles from the sanitization graph, the number of paths can still grow exponentially. When the number of paths exceeds the capacity of our dynamic analysis to examine them all, we assume that the sanitization is incorrect. Again, this problem has not occurred during our experiments.

Depending on the type of the sink, we then select the appropriate test suite to be used in the experiment. For the prototype implementation of Saner, we implemented two different test suites: one for testing cross-site scripting attacks and one for testing SQL injection attacks. Both test suites contain a large number of test cases, each representing a particular value of user input that contains malicious data. We created these test suites using attack strings derived from both our own experience and from specialized web sites (such as [23, 35]). For instance, typical input values that can be used to test the effectiveness of sanitization code for XSS are:

```
<script>alert(1);</script>
<scr<scriptipt src=http://evil.com/attack.js>
<img onmouseover="alert(1);" src=""></img>
<SCRIPT src="http://..."?<B>
```

Some of these test cases are just straightforward examples of XSS attacks, while others represent more complex cases that are likely to evade poorly-written sanitization routines. Note that we do not claim that the set of test cases that we adopt in our test runs is complete. Instead, the goal of the dynamic analysis part is to examine automatically and in detail those cases in which the static analysis phase has reported a potential vulnerability.

Finally, we invoke the PHP interpreter to evaluate the result of executing each block of code C_i on the list of malicious inputs contained in the selected test suite. The results of the executions are collected and analyzed by a special oracle function associated with the test case. This function is responsible for deciding whether the input string was successfully sanitized by the code under test. To do that, it is usually enough to check for the occurrence of particular substrings in the result. However, more advanced techniques can be used to implement oracle functions, such as analyzing the structure of a SQL query to verify the result of a SQL injection.

In our example of Figure 10, we can identify two different paths through the sanitization graph. During the execution of the first test case (the one that uses the input string `<script>alert(1);</script>`), we invoke the PHP interpreter to evaluate the two following pieces of code:

```
Code 1:
  $tmp = "<script>alert(1);</script>";
  $tmp = eregi_replace("<script", "", $tmp);
  $tmp = mysql_escape_string($tmp);

Code 2:
  $tmp = "<script>alert(1);</script>";
  $tmp = eregi_replace("<script", "", $tmp);
  $tmp = mysql_real_escape_string($tmp);
```

At the end of the two executions, the value of the `$tmp` variable is analyzed by the oracle function associated with the test case. In this simple case, the oracle is realized as a check to verify that the resulting string still contains the unescaped `script` tag. Note that the execution of this test does not spot any vulnerability, since the `eregi_replace` function does remove the opening tag from the input string. However, the following test case, which is based on the attack string `<scr<scriptipt src=http://evil.com/attack.js>`, will immediately reveal the weakness of the sanitization routine.

4 Evaluation

We implemented our approach in a prototype tool called Saner, and we evaluated this system on five popular, publicly-available PHP applications that contain custom sanitization routines.

The results of our analysis are shown in Table 1. The table shows the total number of security-sensitive sinks that are present in each analyzed program (*sinks total*). The next column (*sinks with sanitization*) shows the subset of sinks that have as input at least one value that depends on the output of a sanitization routine. That is, for each sink, there exists at least one program path such that the output of a sanitization routine flows into this sink. This number is important, as it represents those cases where incorrect sanitization could have a potential impact on the security of the program. It serves as a baseline for those sinks that need to be further analyzed for incorrect sanitization. Note that the number of sinks with sanitization is significantly smaller than the total number of sinks. The reason is that many of the sensitive sinks do not receive any tainted input. Also, in a few cases, tainted input reaches sensitive sinks directly, without any sanitization. In such cases, Pixy would report a vulnerability. However, for this paper, we are only interested in paths on which sanitization operations are performed.

The column *eliminated (basic)* shows the number of sensitive sinks with sanitization for which the basic analysis of Pixy determines that there is no security problem. This could be, for example, when the sanitization routine is not processing any malicious input, and, as a result, its output is guaranteed to be benign. The column *eliminated (advanced)* counts those cases in which the sensitive sinks do process potentially malicious input, but our improved static analysis is able to verify that the sanitization process works as intended. Finally, all sinks for which the static

Application	Sinks		Static Analysis		Dynamic Analysis		
	Total	With Sanitization	Sinks Eliminated (Basic)	Sinks Eliminated (Advanced)	Sinks Analyzed	Sinks Vulnerable	Not Vulnerable
Jetbox 2.1	311	7	5	0	2	1	1
MyEasyMarket 4.1	737	50	0	45	5	5	0
PBLGuestbook 1.32	41	5	2	0	3	3	0
PHP-Fusion 6.01	1,015	67	19	0	48	4	44
Sendcard 3.4.1	84	10	2	0	8	1	7
Totals	2,188	139	28	45	66	14	52

Table 1. Detection results.

analysis process cannot verify the correctness of the preceding sanitization routines are forwarded to the dynamic analysis (*sinks analyzed*).

The column *sinks confirmed* reports the total number of sinks that are confirmed to be vulnerable after running the dynamic analysis phase. The last column (*not vulnerable*) shows those sinks for which the dynamic analysis could not find input to bypass the sanitization routines. We manually verified that all these cases are indeed harmless and represent false positives produced by the static analysis phase. This demonstrates that the dynamic analysis phase was able to correctly produce input values to bypass the sanitization routines in all vulnerable cases. Of course, in general, dynamic analysis suffers from false negatives. Thus, to remain sound, a human analyst would have to verify all cases manually in which no malicious input is found. However, our experiments show that the dynamic analysis phase is very accurate in practice. Therefore, one could choose to trust its results. In this case, the system would only report sanitization routines that are very likely incorrect and also suggest input values that can be used to exploit the identified vulnerabilities.

The outcome of the experiments confirms our original hypothesis that sanitization mechanisms used in real-world applications are not always effective and can often be circumvented by determined attackers. To the best of our knowledge, all the ineffective sanitization routines discovered during the experiments correspond to novel, previously unknown vulnerabilities (with the exception of one of those found in PBLGuestbook, which was previously described in CVE-2006-3617). Therefore, we identified 13 novel vulnerabilities in the five applications we analyzed. We have working exploits for each vulnerability and notified the appropriate application developers.

4.1 Discussion of Sanitization Errors

A detailed analysis of the vulnerabilities detected in our experiments reveals that the sanitization process performed by a program can be ineffective for several reasons, which we classify based on the most common cases that we have encountered.

First, the code that performs the sanitization can contain programming errors. That is especially true if the sanitization is based on regular expressions, whose complex syntax can lead inexperienced developers to introduce subtle bugs in their specification. For example, MyEasyMarket attempts to sanitize the user-provided parameter `www` by using the following regular expression-based substitution:

```
ereg_replace("[^A-Za-z0-9 .-@://]","",$www);
```

Clearly, the parameter is used to store a URL and the developer intended to allow it to include the '−' (dash) character. Unfortunately, the dash character, when used inside a character class definition (marked by the square brackets), is interpreted as the character range separator. Therefore, the regular expression leaves unaffected all characters included in the range between '`.`' (dot) and '`@`' (at), which includes the open and close tag characters (< and >) and the equal symbol (=). Thus, an attacker can inject the string `<script src=http://evil.com/attack.js/>` and successfully perform a cross-site scripting attack.

Second, the sanitization process can be bug-free but insufficient. This is usually due to two reasons: the developer is not aware of all possible attack vectors (e.g., does not remove all HTML elements that can cause the execution of JavaScript code), or, even if she is, she ignores the fact that browsers accept and interpret malformed, non-standard documents. As an example of the first problem, consider the following sanitization performed in Jetbox (slightly simplified for readability):

```
function removeEvilTags($source){
  $allowedTags = "<h1><b><i><a>";
  $source = strip_tags($source, $allowedTags);
  return preg_replace('/<(.*?)>/ie',
     "'<'.removeEvilAttributes('\\1').'>'",
     $source);
}


function removeEvilAttributes($tagSource){
   $stripAttrib = 'javascript:|onclick|".
     "ondblclick|onmousedown|onmouseup|".
     "onmouseover|onmousemove|onmouseout|".
     "onkeypress|onkeydown|onkeyup|style|".
     "onload|onchange';

   return preg_replace("/$stripAttrib/i",
     'forbidden', $tagSource);
}
```

The developer intended to only permit the use of a limited number of HTML tags cleaned of attributes that allow for the execution of JavaScript code. However, the list of insecure attributes is not complete: for example, the input string `<a onfocus="malicious code" href="url">dummy</a>` remains unaltered when it is processed by the sanitization routines, and thus, it can be used to execute malicious code. As an example of the second problem, consider the sanitization performed by PBLGuestbook:

```
preg_replace(
    "/\<SCRIPT(.*?)\>(.*?)\<\/SCRIPT(.*?)\>/i",
    "SCRIPT BLOCKED", $value);
```

Note that the specified pattern looks for a closing `script` tag. Unfortunately, most browsers accept malformed documents where an open tag is not followed by a corresponding close tag, and automatically insert the missing close tag. Therefore, an attacker can provide the input string `<script>malicious code<` to circumvent this sanitization.

Finally, the sanitization process implemented by a developer may correctly take into account all attack vectors but still be evadable. For example, consider the following sanitization:

```
str_replace("script","", $input)
```

This sanitization routine is intended to completely remove all occurrences of the string "`script`" from the user input. Unfortunately, an attacker can bypass this sanitization by providing the string `<scrscriptipt> code </scrscriptipt>` as input. The reason is that this string is transformed by the sanitization procedure into `<script>code <script>`, which invokes the embedded JavaScript code.

4.2 Discussion of Effectiveness and Efficiency

The combination of static and dynamic techniques proved to be effective. In fact, for the smaller applications, the dynamic testing phase was always able to automatically confirm all the alerts provided by the static analysis part. The advantage of using the dynamic analysis is more evident when analyzing larger applications. For example, in PHP-Fusion, the static analysis component generates a large number of alerts, which, in all benign cases, were correctly identified as false positives by the dynamic analysis phase.

Our results also indicate that current state-of-the-art vulnerability analysis tools would benefit from our approach, especially when analyzing applications that use a non-trivial amount of custom sanitization code. In fact, our approach provides a method to reduce the false positives that are generated when a tool conservatively considers all custom sanitization routines to be ineffective, and false negatives if the tool takes the opposite approach of considering all sanitization routines to be secure. The reason is that the sanitization functions are precisely modeled, and not *a priori* assumed to be either entirely correct or faulty.

Table 2 presents the runtime performance of Saner. For each application, we report the total number of lines of code (*lines of code*), the time required to perform the static analysis phase (*static analysis time*), the time required to perform the dynamic analysis phase (*dynamic analysis time*), and the total time (*total time*). Note that, even though in this prototype implementation performance was not a primary consideration, the time required to perform our analysis is in the order of a few minutes for almost all applications, and, in all cases, well under 20 minutes. Through careful engineering, the performance of our research prototype could be further enhanced. However, we believe that the current system operates well in practice and can be successfully used with large, real-world applications.

Interestingly, during the dynamic analysis phase, most of the time was spent to compute the inter-procedural data flow graph and extract the sanitization graphs. In our experiments, sanitization graphs were generally small, both in terms of number of nodes (i.e., sanitization primitives used) and of number of paths. Therefore, the time spent running the test attacks had a limited impact on the total time.

5 Related Work

The approach described in this paper is a composition of static and dynamic analysis techniques. Therefore, in the following two sections, we review the research work that is related to these two types of analysis.

5.1 Static Analysis

Type-Based Analysis. For typed programming languages, information about the taint status of variables can be propagated through the program by extending the type system of the language. For example, CQual [7] is a tool that allows one to extend the type system of the C language with user-defined qualifiers. After defining the new type system, the programmer manually introduces the additional qualifiers at a few key points in the application. CQual's qualifier inference then determines whether the program contains a type error under the extended system. This technique was used by Shankar et al. [38] for the detection of format string vulnerabilities, and by Johnson and Wagner [16] to identify user/kernel pointer bugs in the Linux kernel. Analogously, Zhang et al. [46] discovered security problems regarding the placement of authorization hooks in the Linux Security Modules framework.

JFlow [26] is an extension to the Java programming language that adds a type system for tracking information flow. In this system, the user is provided with annotations (labels) that define restrictions on the way in which the information may be used in the program, permitting the verification of information confidentiality and integrity. JFlow

Application	Lines of Code (#)	Static Analysis Time (s)	Dynamic Analysis Time (s)	Total time (s)
Jetbox 2.1	69,177	62	5	67
MyEasyMarket 4.1	2,544	202	26	228
PBLGuestbook 1.32	1,595	40	180	220
PHP-Fusion 6.01	56,339	723	386	1,109
Sendcard 3.4.1	8,504	130	38	168

Table 2. Performance results.

supports a wide range of language features (such as objects and exceptions), and is implemented in the Jif [15] tool.

Rule-Based Bug Finding. In [5], Engler et al. present meta-level compilation, a technique for the translation of simple user-defined rules (such as "never use floating point in the kernel") into extensions for the C compiler. During the compilation of a program, these extensions are able to determine whether the program violates the specified rules. An automated extraction of such program rules from a given application is described in [6]. In [2], the authors use the system to detect potentially dangerous accesses to user-supplied, unchecked values in Linux and OpenBSD.

Web Application Analysis. There exist several approaches that are focused on the detection of "taint-style" vulnerabilities (such as XSS or SQL injections), which frequently occur in web applications. Huang et al. [13] adapted parts of the techniques used in CQual to develop an intraprocedural analysis for PHP programs. In [14], the same authors present an alternative approach that is based on bounded model checking. Whaley and Lam [44] described an interprocedural, flow-insensitive alias analysis for Java applications. Their analysis is based on binary decision diagrams, and was used by Livshits and Lam [22] for the detection of taint-style vulnerabilities. As already mentioned, our approach is based on Pixy [17,18], an open source static PHP analyzer that uses taint analysis for detecting XSS vulnerabilities.

In [9], the authors applied the Java String Analyzer by Christensen et al. [4] to extract models of a program's database queries, and used these models as the basis for a runtime monitoring and protection component for SQL injection attacks. The main difference compared to our approach is that the extracted models do not contain information about the taint status of embedded variables. As a result, it is not possible to detect vulnerabilities using static analysis only. In our system, we can identify vulnerabilities using static analysis alone. In addition, we use a dynamic phase to automatically determine inputs that can exploit a vulnerability.

Xie and Aiken [45] presented an interprocedural and flow-sensitive system for the discovery of SQL injection vulnerabilities through a bottom-up analysis of basic blocks, procedures, and the whole program. In their work, the authors take into account the effect of applying one of a number of regular expressions to an input value. In principle, the authors manually specify a list of regular expressions that simply extends the list of built-in sanitization routines (such as `htmlentities`). In contrast, our technique automatically decides whether an arbitrary regular expression is suitable for sanitization, and requires no manual extensions when scanning new applications.

The system that is probably closest to ours was developed by Wassermann and Su [43]. In their paper, the authors present a static analysis technique for finding subtle SQL injection flaws. For this, they independently and concurrently developed a mechanism to determine the possible string values of variables in PHP programs. This allows them to take into account the effect of sanitization routines. The differences to our work are twofold. First, our focus is different. We attempt to verify the correctness of the sanitization process and do not limit our analysis to the detection of SQL injection vulnerabilities. Second, our system employs an additional dynamic analysis phase to find automatically input values that can exploit a vulnerability.

5.2 Dynamic Analysis

The dynamic techniques described in this paper are related to the research in the area of applying dynamic taint propagation analysis to web applications. Perl's *Taint mode* [31] is one of the best-known examples of such approaches. Similar approaches have been applied to other languages as well: Nguyen-Tuong et al. [28] propose modifications of the PHP interpreter to dynamically track tainted data in PHP programs, and Haldar et al. [8] have instrumented the Java Virtual Machine. Pietraszek and Vanden Berghe [33] present a unifying view of injection vulnerabilities and describe a general approach to the detection and prevention of injection attacks through the dynamic tracking of the flow of untrusted data inside an application. None of these approaches, however, provides a precise modeling of sanitization routines used to untaint data, and, thus, these approaches do not offer an effective protection against web attacks.

The dynamic part of our work is also related to a number of research results and tools in the areas of application security testing and fault injection [3,12,19,29,37,40]. All these systems inject malicious (or sometimes random) input into the applications to identify security problems. In our tool, we inject strings corresponding to possible XSS and SQL injection attacks to dynamically execute parts of the analyzed applications. Our work is also related to the

data flow testing of applications, such as [10, 11, 34]. In this type of testing, data flow graphs are used to identify the test case requirements for a program. In our approach, we use data flow graphs to find program statements related to the sanitization process.

6 Conclusions

Web applications perform mission-critical tasks and handle sensitive information. Even though there have been a number of research efforts to identify the use of unvalidated input in web applications, little has been done to characterize how sanitization is actually performed and how effective it is in blocking web-based attacks.

In this paper, we have presented Saner, a novel approach to the evaluation of the sanitization process in web applications. The approach relies on two complementary analysis techniques to identify faulty sanitization procedures. We implemented our approach, and by applying it to real-world applications, we identified novel vulnerabilities that stem from incorrect or incomplete sanitization. Future work will focus on the analysis of type-based validation procedures, as scripting languages often allow the programmer to interpret the values of variables in different ways, depending on the application context. This might lead to vulnerabilities that are difficult to detect.

Acknowledgements

This work has been supported by the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157, and P-18368, and by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

References

[1] C. Anley. Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software, Ltd, 2002.

[2] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy*, 2002.

[3] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *9th Information Security Conference (ISC)*, Samos Island, Greece, September 2006.

[4] A. Christensen, A. Møller, and M. Schwartzbach. Precise Analysis of String Expressions. In *International Static Analysis Symposium (SAS)*, 2003.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating System Principles (SOSP)*, 2001.

[7] J. Foster, M. Faehndrich, and A. Aiken. A Theory of Type Qualifiers. In *Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *21st Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, December 2005.

[9] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *International Conference on Automated Software Engineering (ASE)*, pages 174–183, November 2005.

[10] M. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, 1994.

[11] M. Harrold and M. Soffa. Interprocedural Data Flow Testing. In *ACM SIGSOFT 3rd Symposium on Software Testing, Analysis, and Verifications*, pages 158–167, 1989.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *11th International Conference on World Wide Web (WWW)*, 2003.

[13] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *12th International World Wide Web Conference (WWW)*, 2004.

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

[15] Jif: Java + Information Flow. `http://www.cs.cornell.edu/jif/`, 2007.

[16] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *13th USENIX Security Symposium*, 2004.

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

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

[19] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *15th International World Wide Web Conference (WWW)*, United Kingdom, May 2006.

[20] A. Klein. Cross Site Scripting Explained. Technical report, Sanctum Inc., 2002.

[21] J. Liberty and D. Hurwitz. *Programming ASP.NET*. O'REILLY, February 2002.

[22] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *14th USENIX Security Symposium*, pages 271–286, August 2005.

[23] F. Mavituna. SQL Injection Cheat Sheet, Version 1.4. `http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/`, May 2007.

[24] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *14th International Conference on World Wide Web (WWW)*, 2005.

[25] M. Mohri and R. Sproat. An Efficient Compiler for Weighted Rewrite Rules. In *34th Annual Meeting on Association for Computational Linguistics*, 1996.

[26] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages (POPL)*, 1999.

[27] Netcraft. PHP Usage Stats. `http://www.php.net/usage.php`, June 2007.

[28] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *20th International Information Security Conference (SEC)*, pages 372–382, May 2005.

[29] Nikto. Web Server Scanner. `http://www.cirt.net/code/nikto.shtml/`.

[30] OWASP. Top ten project. `http://www.owasp.org/`, May 2007.

[31] Perl. Perl security. `http://perldoc.perl.org/perlsec.html`.

[32] PHP: Hypertext Preprocessor. `http://www.php.net`, 2005.

[33] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, pages 372–382, 2005.

[34] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[35] RSnake. XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`, May 2007.

[36] Security Space. Apache Module Report. `http://www.securityspace.com/s_survey/ data/man.200603/apachemods.html`, April 2006.

[37] T. N. SecurityTM. Nessus Vulnerability Scanner. `http://www.nessus.org/`.

[38] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *10th USENIX Security Symposium*, 2001.

[39] K. Spett. Blind SQL Injection. Technical report, SPI Dynamics, 2003.

[40] Spike. `http://www.immunitysec.com/resources-freesoftware.shtml`.

[41] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages (POPL)*, 2006.

[42] Symantec. Symantec internet security threat report, March 2007.

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

[44] J. Whaley and M. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[45] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, August 2006.

[46] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *11th USENIX Security Symposium*, 2002.