Preventing Web Application Injections with Complementary Character Coding

Raymond Mui and Phyllis Frankl

Polytechnic Institute of NYU
6 Metrotech Center
Brooklyn, NY, 11201, USA
wmui01@students.poly.edu,pfrankl@poly.edu

Abstract. Web application injection attacks, such as SQL injection and cross-site scripting (XSS) are major threats to the security of the Internet. Several recent research efforts have investigated the use of dynamic tainting to mitigate these threats. This paper presents complementary character coding, a new approach to character level dynamic tainting which allows efficient and precise taint propagation across the boundaries of server components, and also between servers and clients over HTTP. In this approach, each character has two encodings, which can be used to distinguish trusted and untrusted data. Small modifications to the lexical analyzers in components, such as the application code interpreter, the database management system, and (optionally) the web browser, allow them to become complement aware components, capable of using this alternative character coding scheme to enforce security policies aimed at preventing injection attacks, while continuing to function normally in other respects. This approach overcomes some weaknesses of previous dynamic tainting approaches. Notably, it offers a precise protection against persistent cross-site scripting attacks, as taint information is maintained when data is passed to a database and later retrieved by the application program. A prototype implementation with LAMP and Firefox is described. An empirical evaluation shows that the technique is effective on a group of vulnerable benchmarks and has low overhead.

1 Introduction

Web applications have become an essential part of our every day lives. As web applications become more complex, the number of programming errors and security holes in them increases, putting users at increasing risk. Injection vulnerabilities, such as cross site scripting and SQL injection, rank as the top two of the most critical web application security flaws in the OWASP (Open Web Application Security Project) top ten list [25].

Web applications typically involve interaction of several components, each of which processes a language. For example, an application may generate SQL queries that are sent to a database management system and generate HTML code with embedded Javascript that is sent to a browser, from which the scripts are sent to a Javascript interpreter. Throughout this paper we will use the term

V. Atluri and C. Diaz (Eds.): ESORICS 2011, LNCS 6879, pp. 80–99, 2011.

component languages to refer to the languages of various web application technologies such as PHP, SQL, HTML, Javascript, etc. We will also use the term *components* to denote the software dealing with the parsing and execution of code written in these languages from both server side and client side, such as a PHP interpreter, a database management system, a web browser, etc.

Web application injection attacks occur when user inputs are crafted to cause execution of some component language code that is not intended by the application developer. There are different classes of injection attacks depending on which component language is targeted. For example, SQL injection targets the application's SQL statements, while cross site scripting targets the application's HTML and Javascript code. These vulnerabilities exist because web applications construct statements in these component languages by mixing untrusted user inputs and trusted developer code. Best application development practice demands the inclusion of proper input validation code to remove these vulnerabilities. However, it is hard to do this because proper input validation is context sensitive. That is, the input validation routine required is different depending on the component language for which the user input is used to construct statements. For example, the input validation required for the construction of SQL statements is different from the one required for the construction of HTML, and that is different from the one required for the construction of Javascript statements inside HTML. Because of this and the increasing complexity of web applications, manual applications of input validation are becoming impractical. Just a single mistake could lead to dire consequences.

Researchers have proposed many techniques to guard against injection vulnerabilities. Several approaches use dynamic tainting techniques [6,10,12,23,24,26,27,37]. Current implementations of dynamic tainting involve instrumenting application code or modifying the application language interpreter to keep track of which memory locations contain values that are affected by user inputs. Such values are considered "tainted", or untrusted. At runtime, locations storing user inputs are marked as tainted, the taint markings are propagated so that variables that are affected by inputs (through data flow and/or control flow) can be identified and the taint status of variables is checked at "sinks" where sensitive operations are performed.

Dynamic tainting techniques are effective at preventing many classes of injection attacks, but there are a number of drawbacks to current approaches to implementing dynamic tainting. Perhaps the most limiting of these arises when applications store and/or retrieve persistent data (e.g. using a database). Current approaches to dynamic tainting do not provide a clean way to preserve the taint status of such data. Viewing the entire database as tainted, when retrieving data, is overly conservative. But viewing it as untainted leaves applications vulnerable to persistent attacks, such as stored XSS attacks.

This paper presents a new approach to dynamic tainting, in which taint marks are seamlessly carried with the data as it crosses boundaries between components. In particular, data stored in a database carries its taint status with it, allowing it to be treated appropriately when it is subsequently processed by

other application code. The approach is based on *complementary character coding*, in which each character has two encodings, one used to represent untainted data and the other used to represent tainted data. Characters can be compared with *full comparison*, in which the two representations are treated differently, or *value comparison*, in which they are treated as equivalent. With fairly small modifications, components (e.g. the application language interpreter, DBMS, and optionally client-side components) can become *complement aware components (CACs)*, which use full comparison for recognizing (most) tokens of their component language, while using value comparison in other contexts. When component language code entered by a user (an attempted injection attack) is processed by the CAC under attack, the component does not recognize the component language tokens, therefore does not execute the attack. Meanwhile, trusted component language code is treated normally. This allows secure execution of code without the need of input sanitization. Ideally, the approach will be deployed with complement aware components on both the server side and the client side, but we also demonstrate a server side only approach that still protects current web browsers against XSS attacks. This allows for a gradual migration strategy through the use of server side HTTP content negotiation, supporting both current web browsers and complement aware browsers at once.

In addition to offering protection against stored attacks, our approach has several other attractive features. Existing dynamic tainting approaches require the processing at sinks to embody detailed knowledge of the component language with which the application is interacting at the sink (e.g. SQL dialect, HTML) and to parse the strings accordingly. Our technique delegates this checking to the components, which need to parse the strings the application is passing to them anyway. This provides increased efficiency and, potentially, increased accuracy. Taint propagation is also very efficient with complementary character coding, because taint propagation via data flow occurs automatically, without the need for auxilliary data structures. The main contributions of this work are:

- The concept of *complementary character coding*, a character encoding scheme where each character is encoded with two code points instead of one. Two forms of complementary character coding, *Complementary ASCII* and *complementary Unicode*, are presented.
- A new approach to dynamic tainting with complementary character coding, which allows transparent taint propagation across component boundaries.
- The concept of *complement aware components (CAC)*, which use complementary character coding to prevent a number of web application input injection attacks, including SQL injection and cross site scripting.
- A proof of concept implementation of LAMP (Linux Apache MySQL PHP) and Firefox using the technique in complementary ASCII. The prototype LAMP implementation is also backwards compatible with current web browsers.
- An experimental evaluation of the prototype, demonstrating that the approach effectively prevents SQL injection, reflected, and stored XSS attacks without causing defects for legitimate requests, and has low overhead.

The remainder of this section presents a motivating example. Section 2 introduces complementary character coding. Section 3 describes how complementary character coding is used to implement dynamic tainting and how complement aware components prevent injection attacks, without the need for sanitization. Section 4 illustrates how the example attacks are prevented with our technique. Section 5 describes the prototype implementation, Section 6 shows the results of the experimental evaluation, Section 7 discusses related work, and Section 8 concludes.

```
 1. <?php
 2.
 3. //connect to database
 4. connectdb();
 5.
 6. //unsanitized user inputs
 7. $message = $_POST['message'];
 8. $username = $_POST['username'];
 9.
10. //html header
11. echo '<html>
12.   <head> <title>Blog</title> </head>
13.   <body>';
14.
15. //welcome the user
16. if(isset($username)) {
17.   echo "Welcome $username <br />";
18. }
19.
20. //insert new message
21. if(isset($message)) {
22. $query = "insert into messages values ('$username', '$message')";
23.   $result = mysql_query($query);
24. }
25.
26. //display messages except those from this user or admin
27. $query = "select * from messages where username != '" + $username + "'";
28. $result = mysql_query($query);
29. echo '<br /><b>Your messages:</b>';
30. while($row=mysql_fetch_assoc($result)){
31.   if($row['username'] != "admin") {
32.     echo "<br />{$row['username']} wrote: <br />{$row['message']}<br />";
33.   }
34. }
35.
36. //display the rest of html...
```

Fig. 1. Motivating Example

Figure 1 contains the code of an example web application written in PHP. The database contains a single table, called *messages* with attributes *username* and *message*, both stored as strings. Four input cases are shown in Figure 2. Case one is an example of a normal execution. Case two is a SQL injection attack. Case three is a reflected cross site scripting attack. Case four is a persistent cross site scripting attack. Please refer to appendix A for detailed descriptions of each case. In Section 4 below, we will show how our technique prevents these attacks.

```
Case 1:
username = user          message = hello

Case 2:
username = user          message = hello');drop table messages;--

Case 3:
username = <script>document.location="http://poly.edu"</script>          message = hello

Case 4:
username = user          message = <script>document.location="http://poly.edu"</script>
```

Fig. 2. Input Cases for Example in Fig. 1

2 Complementary Character Coding

In complementary character coding, each character is encoded with two code points instead of one. That is, we have two versions of every character. This section introduces *complementary ASCII* and *complementary Unicode*, two forms of complementary character coding, as well as the concepts of *value comparison* and *full comparison* which are used to compare characters in complementary character coding.

2.1 Complementary ASCII

In complementary character coding, there are two versions of each character. Standard ASCII uses 7 bits per character (with values 0–127), while each byte is 8 bits (with values 0–256). Complementary ASCII is encoded as follows: The lowest seven bits are called the *data bits*, which corresponds to standard ASCII characters 0–127. The eighth bit is called the *sign bit*, a sign bit of 0 corresponds to a *standard character* and a sign bit of 1 corresponds to a *complement character*. In other words, for every standard character c in $\{0...127\}$ from standard ASCII, there exists a complement character $c' = c + 128$ that is its complement. The conversion between standard and complement characters in complementary ASCII can be done in a single instruction by flipping the sign bit.

2.2 Value Comparison and Full Comparison

Since there are two versions of every character in complementary character coding, there must be certain rules to establish how characters are compared. In complementary character coding there are two different ways to compare characters, *value comparison* and *full comparison*. Under value comparison, a standard character is equivalent to its complement version. A simple way to implement value comparison is to compute the standard forms of the characters and compare them. Full comparison, however, compares all bits of a character. Therefore under full comparison the standard and complement versions of the same character are not equal. Note that all complement characters will be evaluated as greater than all standard characters under full comparison regardless of the value of their data bits. This is not a problem because our technique only uses full comparison for *equals* and *not equals* comparisons.

2.3 Complementary Unicode

With the internationalization of the web, Unicode [32] schemes are becoming the standard character formats for displaying web content. Currently Unicode contains over a million code points and as of the current version of Unicode 6.0 less than 25 percent of this space is used or reserved. Due to the vast amount of available space, complementary Unicode can be implemented in different ways. One possible implementation of complementary Unicode can be done just like complementary ASCII through the use of the high order bit as the sign bit. Under this representation the operations of character conversion, value comparison and full comparison are implemented in nearly the same way as their counterparts in complementary ASCII. The extra space also allows the possibility of for having more than two versions of every character, which will be explored in future work.

3 Preventing Injections with Complementary Coding

This section describes how complementary character coding is used to implement dynamic tainting and how complement aware components prevent injection attacks, without the need for sanitization. The use of HTTP content negotiation to ensure backwards compatibility with non-complement aware web browsers is also discussed. Assumptions about the application code and environment are noted in section 3.4.

3.1 Dynamic Tainting with Complementary Coding

Prior dynamic tainting techniques maintain a data structure indicating which variables or memory locations are tainted. This data structure is initialized and updated either by instrumenting the application code or by modifying the application language interpreter. Points where the application sends code to other components (e.g. SQL statements to the DBMS, HTML to the browser, etc.) are called "sinks". Custom checks are performed at sinks to check whether tainted data is used inappropriately.

In contrast, our approach does not require an additional data structure to track taint status. Trusted application code is encoded in standard characters as not tainted. When character strings from an untrusted source enter the system, they are tainted as the web server converts them into complement characters. Value comparison is used to compare characters during execution of the application code, thus the program continues to function normally in spite of the fact that extra information (taint status) is carried along with each character. Since a character and its taint status reside in the same piece of data, taint propagation via dataflow occurs automatically.

If the component C to which a string is being sent is complement aware, checking of whether tainted data is being used appropriately is delegated to C, as discussed in section 3.2, so no code instrumentation is needed at the taint sink. If C is a legacy component that is not complement aware, taint sink processing similar to that of existing dynamic tainting techniques can be used.

Complementary character coding has the following advantages over prior dynamic tainting techniques: First it allows for free taint storage and implicit taint propagation through normal execution, removing the need for code instrumentation and the resulting overhead of existing dynamic tainting techniques. Second, under the guise of a character encoding, our technique allows for seamless taint propagation between different server-side components, and also between servers and clients over HTTP. This approach is particularly useful against persistent cross site scripting attacks, as taint status of every character is automatically stored in the database, along with the character. Data read in from the database carries detailed information about taint status. Thus, when such data becomes the web application output, it can be handled appropriately (either through a complement aware browser or through server-side filtering.)

3.2 Complement Aware Components

We now describe how a component can leverage complementary character coding to allow safe execution against injection attacks. A web application constructs statements of a component language by mixing trusted strings provided by the developers and untrusted user input data and sends these to other components.

Each component C takes inputs in a formal language $\mathcal{L}_\mathcal{C}$ with a well-defined lexical and grammatical structure (SQL, HTML, etc.). As in reference [30] each component language can have a security policy that stipulates where untrusted user inputs are permitted within elements of $\mathcal{L}_\mathcal{C}$. In general, a security policy could be expressed at the level of $\mathcal{L}_\mathcal{C}$'s context free grammar, but our technique focuses on security policies defined at the level of $\mathcal{L}_\mathcal{C}$'s lexical structure.

In our approach, complementary character coding is used to distinguish trusted (developer-generated) characters from untrusted (user-generated) characters throughout the system. Trusted characters are represented by standard characters while untrusted characters are converted to complement characters by the web server. By making small modifications to their parsers, components can be made *complement aware*, capable of safe execution against input injection attacks without need of sanitization through the enforcement of a default security policy, or other optional policies if the default policy is deemed too restrictive.

More formally, the security policy of a complement aware component C is defined in terms of the tokens of $\mathcal{L}_\mathcal{C}$. The *allowed tokens* are tokens which can include untrusted characters; all other tokens are designated as *sensitive tokens* where untrusted characters are not allowed.

We define a *Default Policy* for each component language as follows: *All tokens except literal strings and numbers are sensitive.* The Default Policy defines the allowed token set as numbers and literal strings, all other tokens are defined as sensitive tokens. For example, the Default Policy applied to SQL states that tokens representing numbers and literal strings are allowed tokens, while all other tokens representing SQL keywords, operators, attribute names, delimiters, etc. are sensitive tokens.

A component C with input language $\mathcal{L}_\mathcal{C}$ is *complement aware* with respect to a security policy P with allowed token set A_P if

- The character set includes all relevant standard and complement characters (e.g. complementary ASCII or complementary Unicode).
- Sensitive tokens, i.e., tokens that are not in A_P, only contain standard characters.
- $\mathcal{L}_\mathcal{C}$ has a default token d which is in A_P. Strings that do not match any other token match d. (Typically this would be the string literal token).
- During lexical analysis C uses value comparison while attempting to recognize tokens in A_P and uses full comparison for all other tokens.
- Aside from parsing, C uses value comparison (e.g. during execution).

The first four elements assure that complement aware components enforce their security policies and the last element allow the component to function normally after checking the security policy, so data values are compared as usual, preserving normal functionality.

Assume trusted developer code is encoded in standard characters and user inputs are translated into complement characters on entry to the system (e.g. by the web server). When the application sends a string s to component C, no substring of s that contains complement characters can match any sensitive token under full comparison. Therefore the following **Safety Property** is satisfied: If component C is complement aware with respect to security policy P then C enforces P, i.e., for any string s, consisting of trusted (standard) and untrusted (complement) characters that is input to C, parsing s with $\mathcal{L}_\mathcal{C}$'s grammar yields a parse tree in which every token (terminal symbol) that contains untrusted characters is in A_P. Consequently, when the parsed token stream is further interpreted (e.g. during execution of the input), no sensitive tokens will come from untrusted inputs.

Note that if C is complement aware with respect to the Default Policy and if s is an attempted injection attack in which characters that come from the user are encoded with complement characters, then C's lexical analyzer will treat any keywords, operators, delimiters, etc. in s that contain complement characters (i.e. that were entered by the user) as parts of the default token (string literal), and the attack string will be safely executed like normal inputs.

The Default Policy is a strong policy that is restrictive. It is designed to be a safe default that is applicable to a wide number of languages against both malicious and non-malicious types of injections. For example, the Default Policy would define the use of HTML boldface tags (<*b*> and </*b*>) from user inputs as a form of HTML injection. Other less restrictive policies can be defined through the addition of more tokens to the allowed token set A_P. For example, if the developers of a web browser wish to allow users to enter boldface tags, they can modify the Default Policy by adding boldface tags to A_P, creating a less restrictive policy which allows the browser to interpret untrusted boldface tags.

3.3 Architecture, Backwards Compatibility and Migration Strategy

Figure 3 provides an architectural overview of our technique. User inputs are converted into complement characters by the server upon entry. We can ensure backwards compatibility between the complement aware server and legacy web

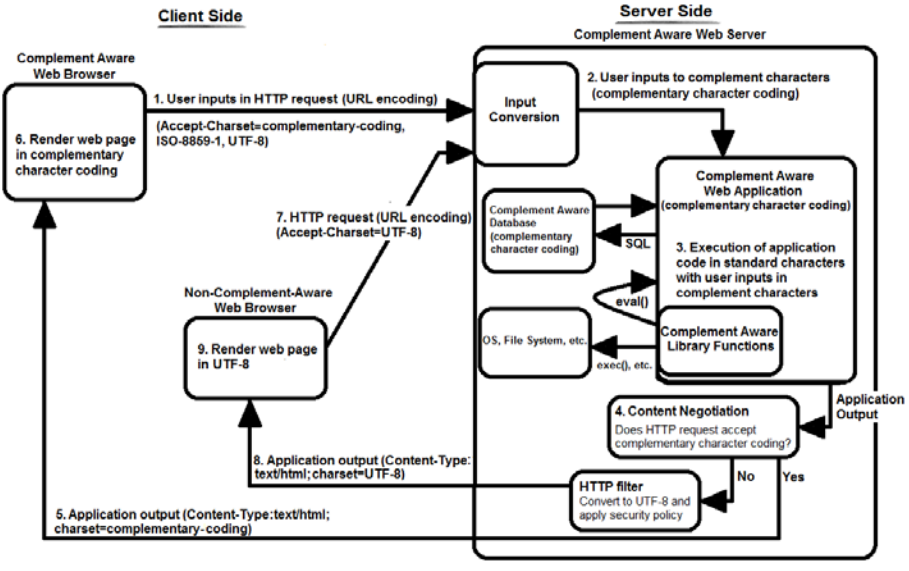

Fig. 3. Architecture of Our Technique

browsers with the use of HTTP content negotiation [36] with the Accept-Charset header. A content negotiation module, shown in step 4 of Figure 3, routes the application output in two ways. For a complement aware browser which specifies itself as complement aware in the Accept-Charset header, the content negotiation module sends the application output in complementary character coding over HTTP unchanged. For a web browser that does not support complementary character coding, the negotiation module routes the output to an HTTP filter. The filter performs the function of a complement aware web browser on the server side at the expense of server side overhead. It does so by applying the Default Policy for HTML and converting its character encoding to one that is readable by the client web browser, specified by the Accept-Charset header in the request. This modified output is then sent back to the client web browser.[1]

This architecture allows for a gradual migration strategy. Initially, deployment of complement aware servers would result in the usage of the HTTP filter for nearly all requests, resulting in extra server overhead. This extra server overhead would gradually decrease, as more and more users upgrade to complement aware web browsers, which do not require the filtering. Please refer to Appendix B for a step by step walk-through of the architecture.

3.4 Limitations

Complement aware components are mostly compatible with existing application code, with some exceptions. Since our technique is designed to execute code

[1] Server administrators who do not want clients to see which parts of the HTML come from user inputs might opt to apply server side filtering as well.

without the use of sanitization functions, existing sanitization in application code would have to be removed to avoid possible conflicts. While library sanitization functions (such as *mysql_real_escape_string* in PHP) can be made to do nothing by default, custom sanitization code would have to be changed manually. Also, application code that involves bit level operations on characters (e.g. shifting left) would not work with the technique. However in the context of web applications, we expect that developer code involving direct bit manipulations are rare. We did not encounter these during our experiments. The technique is also circumvented by applications that produce statements in component languages that are control-dependent, but not data dependent on inputs. The same problem occurs with other dynamic tainting techniques unless taint propagation via control dependence is implemented [7]. We also assume that the technique is not being used in an environment that is already compromised.

4 Example Revisited with CAC

We demonstrate how the four example cases from Section 1.1 will execute as complement aware components enforcing the Default Policy with complementary ASCII. Assume we are using a complement aware web browser. First, according to steps 1 and 2 on Figure 3, all user inputs are converted into complement characters by the server upon arrival. Developer code is encoded in standard characters. We describe each case as we begin step 3 on Figure 3, as the application begins to execute. In the example cases, we will show all complement characters with underlines.

In case one, first the application generates *Welcome user* as HTML at lines 16 to 18. At line 24, the application constructs the SQL query *insert into messages values ('user', 'hello')* and sends it to the DBMS to be executed. During parsing of the SQL query, the complement aware DBMS enforces the Default Policy by using full comparison to match all sensitive tokens in SQL. The tokens *user* and *hello* are recognized as literal strings (albeit with a non-standard character set). The values *user* and *hello* are stored in the database. When lines 27 to 34 are executed, the application generates HTML to display the contents of the database. A SQL query *select * from messages where username != 'user'* is generated at line 27 and the query is passed to the DBMS at line 28. Aside from the literal *user*, the SQL tokens in this query are encoded entirely in standard characters; each string representing a token matches the intended token using full comparison, so the query is executed. During the execution of the SQL query, value comparison is used to evaluate the WHERE clause. Since under value comparison, *user*, *user*, *user*, etc. are equal to *user*, rows containing any of those entries are not selected. Other rows with usernames not equal to *user* under value comparison are selected as desired. Similarly, when the PHP interpreter performs the comparison at line 31, it uses value comparison, which works correctly – the values *admin*, *admin*, *admin*, *admin*, etc. are all equivalent to each other under value comparison so messages posted by any of these variants are excluded from the output. The generated HTML is then sent to the complement aware web

browser, which parses the HTML (Steps 4, 5 and 6 on Figure 3). To enforce the Default Policy, full comparison is used during parsing to match HTML tags. Since *user* and *hello* are in complement characters while HTML tags are in standard characters, they cannot be matched as any tag under full comparison during parsing and the Default Policy is enforced. After parsing, the characters are then rendered by the web browser, at this point value comparison is used in principle, so complement characters are made to look the same as their standard counterparts on the user's screen.

The executions of cases two, three, four or any other input proceed similarly as case one. We will briefly discuss how the attacks from each case are prevented.

In case two, the SQL query *insert into messages values ('user', 'hello');drop table messages;−−')* is constructed and sent to the database parser at line 24. Full comparison is used during parsing. No substring of *hello');drop table messages;−−* matches any sensitive tokens in SQL because under full comparison, ′ is not equal to ′,) is not equal to), etc. Therefore the entire input string is recognized as the default token (string literal) and is stored in the database just like any other string the user provides. The injection attack fails.

In case three, value *Welcome <script>document.location="http://poly.edu" </script>* is generated as HTML at lines 16-18. When the page is parsed by the complement aware web browser, the HTML parser uses full comparison. No tags are matched by the parser because *<script>* is not equal to *<script>* under full comparison. So the browser does not interpret the injected tag as the beginning of a script. Instead, this string is rendered literally on the screen and the reflected XSS attack fails.

Case four is the same as case three except that the attack string is stored in the database as well. Like before, the input does not match any tokens in SQL or any HTML tags under full comparison during parsing. The string is stored literally in the database and is displayed literally in the client's web browser.

This example only shows the prevention of SQL injection and cross-site scripting, however it's important to note that our technique is designed to be general and it can be used against other types of web application injections as well. With complementary character coding, wherever user input is being used to construct statements in a language that is interpreted by other components (XML interpreters, eval, etc), security policies for those components can be defined and complement aware versions of the component can be implemented to prevent injection attacks.

5 Implementation

We describe our implementation of a complement aware web server with LAMP (Linux Apache MySQL PHP) and a complement aware web browser with Firefox. The implementation is done in complementary ASCII. Our implementation of LAMP enforces the Default Policy, while our implementation of Firefox is able to enforce customized security policies through specified allowed token sets. The complement aware LAMP server incorporates the use of HTTP content

negotiation to be backwards compatible with unmodified browsers as well, as discussed in section 3.3. The key effort is implementing value comparison at the right places, since full comparison is already done by default.

We begin with an installation of LAMP with an 8 bit character encoding. For simplicity, we used the *Latin-1* character set. Latin-1's first 128 characters are exactly the same as the standard characters in complementary ASCII. We use the other 128 characters to represent complement characters and modify the way they are displayed. We modified PHP to encode the contents of GET and POST input arrays into complement characters at the point they are initialized. We modified the PHP interpreter so that the bytecode instructions for comparison used value comparison. The parser continues to use full comparison. PHP string functions are modified to support complementary ASCII. For MySQL, the query execution engine was modified to use value comparison, while the parser continued to use full comparison. The content negotiation module and HTTP filter are implemented with an Apache output filter. Since we are using the Default Policy, the filter simply converts all complement characters to a safe representation by encoding them using HTML numeric character references.

Our implementation of the server is sufficient for performing our experiments. A complete implementation would for example, convert everything that comes from the user (such as *$_SERVER['PATH_INFO']*) into complement characters.

We have modified Firefox 3.5 to be complement aware and enforce a customized security policy specified from an allowed token set, as described in section 3.2. If a tag name contains only standard characters, it is interpreted normally. If a tag name contains complement characters and it exists in the allowed token set, then the corresponding tag is interpreted normally. If the tag name contains complement characters and it does not exist in the allowed token set, then the tag is not interpreted and is rendered literally.

6 Evaluation

Our experimental evaluation has two objectives: 1) to confirm that the technique is effective against attacks without causing defects, and 2) to measure the runtime overhead resulting from using our implementation. Two sets of test data were used. The SQL Injection Application Testbed [29] has been used for evaluating various techniques developed by other researchers [2,11,12,28,30]. It consists of a large number of test cases on a series of open source applications. It contains two types of test cases: the ATTACK set which contains SQL injection attacks, and the LEGIT set which contains legitimate inputs. The second benchmark is from ARDILLA [17], a technique which generates cases of SQL injection and XSS attacks automatically. This test set contains cases of SQL injections, and both reflected and persistent cross site scripting attacks on a set of open source applications. The programs are LAMP applications. Links to the benchmark programs can be found in [17,29]. The experiments were performed on a dual core 2 GHz laptop with 3 GB of RAM running our LAMP implementation based on Ubuntu 9.04, Apache 2.2.13, MySQL 5.1.39, and PHP 5.2.11.

Table 1. Summary of SQL Injection Application Testbed and ARDILLA test set

	LOC	SQL Injection	Reflected XSS	Persistent XSS	Legit
SQLIA Testbed					
bookstore	16,959	5474	–	–	608
classifieds	10,949	5590	–	–	576
empldir	5658	6388	–	–	660
events	7,242	5606	–	–	900
portal	16,453	5686	–	–	1080
ARDILLA					
schoolmate	8,181	6	10	2	–
webchess	4,722	12	13	0	–
faqforge	1,712	1	4	0	–
geccbblite	326	2	0	4	–

Table 1 summarizes the benchmarks. The first column contains the names of the applications. The second column contains the number of lines of code (LOC) from each application. Columns 3–5 show the numbers of test cases targeting each type of attack (SQL injection, reflected cross-site scripting, persistent cross-site scripting.) Column 6 shows the number of test cases representing legitimate inputs (not attempted attacks.)

For the much larger SQL Injection Application Testbed, we manually identified the queries that the developer intended to execute for each web page. For example on a Login page the only query is of the form *SELECT * FROM members WHERE username = '?'AND password = '?'*. (Here question marks denote values dependent on user inputs.) A successful SQL injection attack would result in execution of a query with a different structure. We used scripts to execute the ATTACK test cases targeting each page using the CAC prototype, and to check that queries in the database logs matched the intended forms. All the queries were of the intended forms, which shows that the prototype prevented all of the attempted SQL injection attacks from the test suite.

We executed the LEGIT test cases on both the CAC prototype and a standard configuration with identical initial environments, and compared the generated HTML using value comparison. The results of the two implementations were identical by value comparison, except for current time-stamps generated by one page; this shows that the prototype handled these non-attack inputs normally without functionality defects.

We ran the ARDILLA test cases manually. For each test case we checked the database states, database logs and the output HTML for any signs of SQL injection or XSS. We again found no signs of injection attacks. We also found no functionality defects caused by our implementation.

We then measured the response times to determine server side overhead. We expected the overhead of the technique to be small, since the only sources of overhead are from the encoding of user inputs into complement characters and the use of value comparison, each of which was implemented in a few instructions. Our evaluation is done by comparing the difference in runtime between the original LAMP installation that our implementation is based on, and our CAC implementation both with and without the use of the HTTP filter to measure

Table 2. Result of Timing Evaluation

	Default LAMP (seconds)	CAC without filter (seconds)	Percent Overhead (without filter)	CAC with filter (seconds)	Percent Overhead (with filter)
bookstore	6.816 ± 0.055	6.867 ± 0.058	0.74%	6.935 ± 0.061	1.74%
classifieds	6.852 ± 0.057	6.873 ± 0.095	0.32%	6.915 ± 0.069	0.93%
empldir	10.166 ± 0.075	10.149 ± 0.066	-0.17%	10.183 ± 0.081	0.17%
events	17.745 ± 0.186	17.723 ± 0.181	-0.12%	17.760 ± 0.183	0.09%
portal	45.581 ± 0.202	45.905 ± 0.196	0.71%	45.794 ± 0.228	0.47%

the overhead of our content negotiation technique. We only use the LEGIT set from the SQL Injection Application Testbed for this, since successful attacks from the ATTACK set on the original installation would cause different paths of execution, and produce irrelevant timing results.

We ran this test set on each setup 100 times and computed the average run time and the 95% confidence interval. The results are shown in Table 2. The first column contains the names of the applications. The second column contains the average time of the original LAMP installation over 100 runs along with its 95% confidence interval. The third column contains the average time of our complement aware server implementation without passing through the HTTP filter (interacting with a complement aware web browser). The fourth column contains the percentage difference between columns two and three. The fifth column contains the average time of our complement aware server through the HTTP filter (interacting with a legacy web browser) to show the overhead of our backwards compatibility technique.

These results show a performance improvement of complementary character coding compared to existing dynamic tainting techniques. For example, the average overhead of WASP [12] over the same benchmark is listed as 6%, while the worst case overhead of our technique is no more than 2%. Since overhead were on the order of milliseconds per request, other factors such as database operations, network delay, etc. will easily dominate it when our technique is deployed for real world applications.

7 Related Work

Researchers have proposed many other techniques against web injection attacks. Dynamic tainting techniques [6,10,12,23,24,26,27,37] have the most similarity to our technique. Dynamic tainting techniques are runtime analyses that involve the marking of every string within a program with taint variables and propagating them across execution. Attacks are detected when a tainted string is used as a sensitive value. As discussed in section 3, the difference between our technique and traditional dynamic tainting techniques is that complementary character coding provides character level taint propagation across component boundaries of web applications without the need of code instrumentation and the overhead of maintaining extra data structures to propagate taint information. Another difference is that while previous dynamic tainting techniques implement taint

sinks using code instrumentation to detect attacks, our technique delegates enforcement of the security policy to the parser of each component.

Sekar proposed a technique of black-box taint inference to address some of the limitations with dynamic tainting [28], where the input/output relations of components are observed and maintained to prevent attacks. Su and Wassermann provided a formal definition of input injection attacks and developed a technique to prevent them involving comparing parse trees with an augmented grammar [30]. Bandhakavi, Bisht, Madhusudan, Venkatakrishnan developed CANDID [2], a dynamic approach to detect SQL injection attacks where candidate clones of a SQL query, one with user inputs and one with benign values, are compared during parsing. Louw and Venkatakrishnan proposed a technique to prevent cross site scripting [20] where the application sends two copies of output HTML to a web browser for comparison, one with user inputs and one with benign values. Bisht and Venkatakrishnan proposed a technique called XSS-GUARD [3], in which shadow pages and their parse trees are being compared at the server. Buehrer, Weide, and Sivilotti developed a technique involved with comparing parse trees [5] to prevent SQL injection attacks.

Static techniques [1,11,14,16,19,31,34,35] employ the use of various static code analysis techniques to locate sources of injection vulnerabilities in code. The results are either reported as output or instrumented with monitors for runtime protection, others employ the use of machine learning [13,33]. Martin, Livshits, and Lam developed PQL [21], a program query language that developers can use to find answers about injection flaws in their applications and suggested that static and dynamic techniques can be developed to solve these queries.

Boyd and Keromytis developed a technique called SQLrand [4] to prevent SQL injection attacks based on instruction set randomization. SQL keywords are randomized at the database level so attacks from user input become syntactically incorrect SQL statements. A proxy is set up between the web server and the database to perform randomization of these keywords using a key. Van Gundy and Chen proposed a technique based on instruction set randomization called Noncespaces against cross site scripting [9]. Nadji, Saxena and Song developed a technique against cross site scripting called Document Structure Integrity [22] by incorporating dynamic tainting at the application and instruction set randomization at the web browser. Kirda, Kruegel, Vigna and Jovanovic developed Noxes [18], a client side firewall based approach to detect possibilities of a cross site scripting attack using special rules. Jim, Swamy, and Hicks proposed a cross site scripting prevention technique called browser-enforced embedded policies [15] where a web browser receives instructions from the server over what scripts it should or should not run.

Interpolique [8] is a framework for sanitizing sensitive inputs by replacing their values with Base64 encodings, along with calls to components' Base64 decoding functions. Like our technique, Interpolique converts sensitive inputs into forms that do not match any tokens of component languages. However, unlike our technique, Interpolique requires developers to identify variable uses that should be transformed.

8 Conclusion and Future Work

In this paper, we have presented complementary character coding and complement aware components, a new approach to dynamic tainting for preventing a wide variety of web application injection attacks. In our approach, two encodings are used for each character, standard characters and complement characters. Untrusted data coming from users is encoded with complement characters, while trusted developer code is encoded with standard characters. Complementary character coding allows taint information about each character to be propagated across component boundaries seamlessly. Components are modified to enforce security policies, which are characterized by sets of allowed tokens, for which user input characters should not be permitted. Each complement aware component enforces its policy by using full comparison to match sensitive tokens during parsing. Elsewhere they use value comparison to preserve functionality. This allows them to safely execute attempted injection attacks as normal inputs. While ideally, the technique would be used with complement aware components on both the server side and the client side, it is backward compatible with existing browsers through HTTP content negotiation and server-side filtering. Whether deployed with complement aware browser or with a legacy browser, it provides protection against stored XSS attacks.

We have implemented a prototype for LAMP and Firefox. An experimental evaluation on the prototype prevented all SQL injection, reflected and stored cross-site scripting injection attacks and executed legitimate inputs normally in the benchmarks studied with only small overhead. Directions of future work include extending the prototype to use complementary Unicode, incorporating techniques to deal with taint propagation via control flow, and exploring other applications of complementary character coding and its extended version through the use of multiple taint bits.

Acknowledgments. This research was partially supported by the US Department of Education GAANN grant P200A090157, National Science Foundation grant CCF 0541087, and the Center for Advanced Technology in Telecommunications sponsored by NYSTAR. We thank the reviewers for their helpful comments.

References

1. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: SP 2008: Proceedings of the 2008 IEEE Symposium on Security and Privacy, pp. 387–401. IEEE Computer Society, Washington, DC, USA (2008)
2. Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: Candid: preventing SQL injection attacks using dynamic candidate evaluations. In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 12–24. ACM, New York (2007)

3. Bisht, P., Venkatakrishnan, V.N.: XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 23–43. Springer, Heidelberg (2008)
4. Boyd, S.W., Keromytis, A.D.: SQLrand: Preventing SQL injection attacks. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 292–302. Springer, Heidelberg (2004)
5. Buehrer, G., Weide, B.W., Sivilotti, P.A.G.: Using parse tree validation to prevent SQL injection attacks. In: SEM 2005: Proceedings of the 5th International Workshop on Software Engineering and Middleware, pp. 106–113. ACM, New York (2005)
6. Chin, E., Wagner, D.: Efficient character-level taint tracking for Java. In: Proceedings of the 2009 ACM Workshop on Secure Web Services, SWS 2009, pp. 3–12. ACM, New York (2009)
7. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA 2007: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 196–206. ACM, New York (2007)
8. Kaminsky, D.: Interpolique, `http://dankaminsky.com/interpolique/`
9. Gundy, M.V., Chen, H.: Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: NDSS (2009)
10. Haldar, V., Chandra, D., Franz, M.: Dynamic taint propagation for Java. In: ACSAC 2005: Proceedings of the 21st Annual Computer Security Applications Conference, pp. 303–311. IEEE Computer Society, Washington, DC, USA (2005)
11. Halfond, W.G.J., Orso, A.: Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In: ASE 2005: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183. ACM, New York (2005)
12. Halfond, W.G.J., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In: SIGSOFT 2006/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 175–185. ACM, New York (2006)
13. Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web application security assessment by fault injection and behavior monitoring. In: WWW 2003: Proceedings of the 12th International Conference on World Wide Web, pp. 148–159. ACM, New York (2003)
14. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: WWW 2004: Proceedings of the 13th International Conference on World Wide Web, pp. 40–52. ACM, New York (2004)
15. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: WWW 2007: Proceedings of the 16th International Conference on World Wide Web, pp. 601–610. ACM, New York (2007)
16. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: SP 2006: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 258–263. IEEE Computer Society, Washington, DC, USA (2006)
17. Kieyzun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: ICSE 2009: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pp. 199–209. IEEE Computer Society, Washington, DC, USA (2009)

18. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: a client-side solution for mitigating cross-site scripting attacks. In: SAC 2006: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 330–337. ACM, New York (2006)
19. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: SSYM 2005: Proceedings of the 14th Conference on USENIX Security Symposium, pp. 18–18. USENIX Association, Berkeley (2005)
20. Louw, M.T., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: SP 2009: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, pp. 331–346. IEEE Computer Society, Washington, DC, USA (2009)
21. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. SIGPLAN Not. 40(10), 365–383 (2005)
22. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: NDSS (2009)
23. Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-site scripting prevention with dynamic data tainting and static analysis. In: Proceeding of the Network and Distributed System Security Symposium, NDSS 2007 (2007)
24. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: Sasaki, R., Qing, S., Okamoto, E., Yoshiura, H. (eds.) SEC, pp. 295–308. Springer, Heidelberg (2005)
25. OWASP Top Ten Project, `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`
26. Perl security: Taint mode, `http://perldoc.perl.org/perlsec.html#Taint-mode`
27. Pietraszek, T., Berghe, C.V., Chris, V., Berghe, E.: Defending against injection attacks through context-sensitive string evaluation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 124–145. Springer, Heidelberg (2006)
28. Sekar, R.: An efficient black-box technique for defeating web application attacks. In: NDSS (2009)
29. SQL Injection Application Testbed, `http://www.cc.gatech.edu/~whalfond/testbed.html`
30. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 372–382. ACM, New York (2006)
31. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 87–97. ACM, New York (2009)
32. Unicode Consortium, `http://unicode.org/`
33. Valeur, F., Mutz, D., Vigna, G.: A learning-based approach to the detection of SQL attacks. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 123–140. Springer, Heidelberg (2005)
34. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI 2007: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 32–41. ACM, New York (2007)
35. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 171–180. ACM, New York (2008)

36. World Wide Web Consortium: RFC 2616 Section 12: Content Negotiation, http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html
37. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: USENIX-SS 2006: Proceedings of the 15th Conference on USENIX Security Symposium. USENIX Association, Berkeley (2006)

Appendix A: Detailed Description of Figures 1 and 2. Case one is an example of a normal execution. Lines 7 and 8 get the user's inputs from the HTTP request for this page. Lines 10 to 13 begin generation of an HTML page that will eventually be sent to the user's browser. A greeting is generated as HTML at lines 16 to 18. At lines 21 to 24, an SQL insert statement is generated then sent to MySQL, which inserts data provided by the user into the database. Lines 27 to 34 generate a SQL query, send it to MySQL, then iterate through the result set, generating HTML to display the contents of the database (excluding messages from the admin and the user). The web server sends the generated HTML to the user's browser, which parses it and displays the welcome message and the table on the user's screen. We will assume the database is not compromised initially, so no attacks occurred.

Case two is an example of a SQL injection attack. The SQL code being executed at line 23 becomes *insert into messages values ('user', 'hello');drop table messages;−−')*, since there is no input validation. This results in the deletion of the table *messages* from the database. By modifying the attack string an attacker can construct and execute other malicious SQL code as well.

Case three is an example of a reflected cross site scripting attack. The unsanitized user input (a script) is included in the HTML at line 17. When the HTML is parsed by the browser, it will recognize the script tags and send the enclosed script to its Javascript engine, which will parse it and execute it. In this case the script redirects the user to another website. An attacker can exploit this by inducing users to provide inputs like case three, causing redirection to another malicious web page which steals personal information.

Case four is an example of a persistent cross site scripting attack. At line 23, the attack script is stored in the database. It is sent to any user visiting the application when lines 27 to 34 are executed. This is a more severe form of cross site scripting because it affects everyone visiting the web page.

Appendix B: Illustration of Architecture from Figure 3. We present two scenarios to the architecture from Figure 3 in detail. Scenario (1) uses a complement aware web browser. Scenario (2) uses a non-complement aware web browser to demonstrate our content negotiation mechanism for backwards compatibility. For both scenarios, we assume the complement aware components implement the Default Policy as their security policies.

Scenario 1: In step 1, a HTTP request along with standard URL encoded user inputs are sent to the server by a complement aware web browser. The request is URL encoded as specified by the HTTP protocol, identifying itself as complement aware with the Accept-Charset header. In step 2, the server converts the

user input into complementary ASCII/Unicode as complement characters. The input conversion module returns complement characters for all possible inputs. In step 3, web application executes with user inputs in complement characters, while developer code is in standard characters. Value comparison is used within the application, so it functions normally. The application constructs the HTML output by mixing developer code, user inputs, and values obtained from the database. In step 4, this output is sent to the content negotiation module, which checks the Accept-Charset header of the HTTP request to see if the client browser is complement aware. Since the browser is complement aware in scenario (1), the application output is sent back to the client browser as the HTTP response, labeling the output character set as complementary ASCII/Unicode. In step 5, the complement aware browser receives the HTML output, recognizes the output character set as complementary ASCII/Unicode and parses the output accordingly. During parsing the browser's security policy is enforced. Because the Default Policy is used, full comparison is used to match all HTML tags, comments, etc. Consequently, any such tokens that are tainted, whether they came directly from this user's input or whether they'd been stored previously then retrieved from the database, are treated as default tokens, i.e. string literals. After parsing, the page is then rendered on the screen where value comparison is used in principle; this means that complement characters are made to look like their default counterparts on the screen.

Scenario 2: The browser does not support complementary character coding. Beginning in step 7, the browser sends an URL encoded HTTP request to the server, similar to step 1. However, the request does not identify itself as complement aware at the Accept-Charset header; it accepts UTF-8 instead. The input conversion in step 2 and execution of application code in step 3 are the same as in scenario (1). In step 4, the application output is sent to the content negotiation module, which checks the Accept-Charset header of the HTTP request to see if the client web browser is complement aware. Since the web browser in this scenario does not identify itself as complement aware, the output is sent to an HTTP filter, which applies the Default Policy for HTML, while converting its character encoding to UTF-8. For example, the filter can escape tainted characters occurring in HTML tags using HTML numeric character references. Finally, the new output is sent to the browser in step 8 and rendered normally in step 9.