Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

Adam Kieżun
MIT
akiezun@csail.mit.edu

Philip J. Guo
Stanford University
pg@cs.stanford.edu

Karthick Jayaraman
Syracuse University
kjayaram@syr.edu

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

Abstract

We present a technique for finding security vulnerabilities in Web applications. SQL Injection (SQLI) and cross-site scripting (XSS) attacks are widespread forms of attack in which the attacker crafts the input to the application to access or modify user data and execute malicious code. In the most serious attacks (called second-order, or persistent, XSS), an attacker can corrupt a database so as to cause subsequent users to execute malicious code.

This paper presents an automatic technique for creating inputs that expose SQLI and XSS vulnerabilities. The technique generates sample inputs, symbolically tracks taints through execution (including through database accesses), and mutates the inputs to produce concrete exploits. Ours is the first analysis of which we are aware that precisely addresses second-order XSS attacks.

Our technique creates real attack vectors, has few false positives, incurs no runtime overhead for the deployed application, works without requiring modification of application code, and handles dynamic programming-language constructs. We implemented the technique for PHP, in a tool ARDILLA. We evaluated ARDILLA on five PHP applications and found 68 previously unknown vulnerabilities (23 SQLI, 33 first-order XSS, and 12 second-order XSS).

1 Introduction

This paper presents a technique and an automated tool for finding security vulnerabilities in Web applications. Multi-user Web applications are responsible for handling much of the business on today's Internet. Such applications often manage sensitive data for many users, and that makes them attractive targets for attackers: up to 70% of recently reported vulnerabilities affected Web applications [5]. Therefore, security and privacy are of great importance for Web applications.

Two classes of attacks are particularly common and damaging [5]. In SQL injection (SQLI), the attacker executes malicious database statements by exploiting inadequate validation of data flowing from the user to the database. In cross-site scripting (XSS), the attacker executes malicious code on the victim's machine by exploiting inadequate validation of data flowing to statements that output HTML.

Previous approaches to identifying SQLI and XSS vulnerabilities and preventing exploits include defensive coding, static analysis, dynamic monitoring, and test generation. Each of these approaches has its own merits, but also offers opportunities for improvement. Defensive coding [6] is error-prone and requires rewriting existing software to use safe libraries. Static analysis tools [19, 29] can produce false warnings and do not create concrete examples of inputs that exploit the vulnerabilities. Dynamic monitoring tools [13, 25, 27] incur runtime overhead on the running application and do not detect vulnerabilities until the code has been deployed. Black-box test generation does not take advantage of the application's internals, while previous white-box techniques have not been shown to discover unknown vulnerabilities [30].

We have created a new technique for identifying SQLI and XSS vulnerabilities. Unlike previous approaches, our technique works on unmodified existing code, creates concrete inputs that expose vulnerabilities, operates before software is deployed, has no overhead for the released software, and analyzes application internals to discover vulnerable code. As an implementation of our technique, we created ARDILLA, an automated tool for creating SQLI and XSS attacks in PHP/MySQL applications. ARDILLA is a white-box testing tool, i.e., it requires the source code of the application. ARDILLA is designed for testing PHP applications before deployment. Security vulnerabilities that ARDILLA identifies can be fixed before the software reaches the users because ARDILLA creates concrete attacks that exploit the vulnerability. In our experiments, ARDILLA discovered 68 previously unknown vulnerabilities in five applications.

ARDILLA is based on input generation, taint propagation, and input mutation to find variants of an execution that exploit a vulnerability. We now discuss these components.

ARDILLA can use any **input generator**. Our current implementation uses combined concrete and symbolic execution [1, 11, 30]. During each execution, this input generator monitors the program to record path constraints that capture the outcome of control-flow predicates. The input generator automatically and iteratively generates new inputs by negating one of the observed constraints and solving the modified constraint system. Each newly-created input explores at least one additional control-flow path.

ARDILLA's **vulnerability detection** is based on dynamic *taint analysis* [13, 29]. ARDILLA marks data coming from the

199

user as potentially unsafe (tainted), tracks the flow of tainted data in the application, and checks whether tainted data can reach *sensitive sinks*. An example of a sensitive sink is the PHP `mysql_query` function, which executes a string argument as a MySQL statement. If a string derived from tainted data is passed into this function, then an attacker can potentially perform an SQL injection if the tainted string affects the structure of the SQL query as opposed to just being used as data within the query. Similarly, passing tainted data into functions that output HTML can lead to XSS attacks.

ARDILLA's **taint propagation** is unique in that it tracks the flow of tainted data through the database. When tainted data is stored in the database, the taint information is stored with it. When the data is later retrieved from the database, it is marked with the stored taint. Thus, only data that was tainted upon storing is tainted upon retrieval. This precision makes ARDILLA able to accurately detect second-order (persistent) XSS attacks. By contrast, previous techniques either treat *all* data retrieved from the database as tainted [28, 29] (which may lead to false warnings) or treat all such data as untainted [19] (which may lead to missing real vulnerabilities).

To convincingly demonstrate a vulnerability, a tester or a tool must create concrete example attack vectors [4, 8, 10]. ARDILLA creates concrete attack vectors by systematically **mutating inputs** that propagate taints to sensitive sinks, using a library of strings that can induce SQLI and XSS attacks. This step is necessary because not every flow of tainted data to a sensitive sink indicates a vulnerability because the data may flow through routines that check or sanitize it. ARDILLA then analyzes the difference between the parse trees of application outputs (SQL and HTML) to **check** whether the attack may subvert the behavior of a database or a Web browser, respectively. This step enables ARDILLA to reduce the number of false warnings and to precisely identify real vulnerabilities.

This paper makes the following **contributions**:

- A fully-automatic technique for creating SQLI and XSS attack vectors, including those for second-order (persistent) XSS attacks. (Section 3)
- A novel technique that determines whether a propagated taint is a vulnerability, using input mutation and output comparison. (Section 4.3)
- A novel approach to symbolically tracking the flow of tainted data through a database. (Section 4.4)
- An implementation of the technique for PHP in a tool ARDILLA (Section 4), and evaluation of ARDILLA on real PHP applications (Section 5).

2 SQL Injection and Cross-Site Scripting

This section describes SQLI and XSS Web-application vulnerabilities and illustrates attacks that exploit them.

SQL Injection. A SQLI vulnerability results from the application's use of user input in constructing database statements. The attacker invokes the application, passing as an input a (partial) SQL statement, which the application executes. This permits the attacker to get unauthorized access to, or to damage, the data stored in a database. To prevent this attack, applications need to sanitize input values that are used in constructing SQL statements, or else reject potentially dangerous inputs.

First-order XSS. A first-order XSS (also known as Type 1, or reflected, XSS) vulnerability results from the application inserting part of the user's input in the next HTML page that it renders. The attacker uses social engineering to convince a victim to click on a (disguised) URL that contains malicious HTML/JavaScript code. The user's browser then displays HTML and executes JavaScript that was part of the attacker-crafted malicious URL. This can result in stealing of browser cookies and other sensitive user data. To prevent first-order XSS attacks, users need to check link anchors before clicking on them, and applications need to reject or modify input values that may contain script code.

Second-order XSS. A second-order XSS (also known as persistent, stored, or Type 2 XSS) vulnerability results from the application storing (part of) the attacker's input in a database, and then later inserting it in an HTML page that is displayed to multiple victim users (e.g., in an online bulletin board application). It is harder to prevent second-order XSS than first-order XSS, because applications need to reject or sanitize input values that may contain script code and are displayed in HTML output, *and* need to use different techniques to reject or sanitize input values that may contain SQL code and are used in database commands.

Second-order XSS is much more damaging than first-order XSS, for two reasons: (a) social engineering is not required (the attacker can directly supply the malicious input without tricking users into clicking on a URL), and (b) a single malicious script planted once into a database executes on the browsers of many victim users.

2.1 Example PHP/MySQL Application

PHP is a server-side scripting language widely used in creating Web applications. The program in Figure 1 implements a simple message board that allows users to read and post messages, which are stored in a MySQL database. To use the message board, users of the program fill an HTML form (not shown here) that communicates the inputs to the server via a specially formatted URL, e.g.,

```
http://www.mysite.com/?mode=display&topicid=1
```

Input parameters passed inside the URL are available in the `$_GET` associative array. In this example URL, the input has two key-value pairs: `mode=display` and `topicid=1`.

This program can operate in two modes: posting a message or displaying all messages for a given topic. When

```
1  // exit if parameter 'mode' is not provided
2  if(!isset($_GET['mode'])){
3    exit;
4  }
5
6  if($_GET['mode'] == "add")
7    addMessageForTopic();
8  else if($_GET['mode'] == "display")
9    displayAllMessagesForTopic();
10 else
11   exit;
12
13 function addMessageForTopic(){
14   if(!isset($_GET['msg']) ||
15      !isset($_GET['topicid']) ||
16      !isset($_GET['poster'])){
17        exit;
18   }
19
20   $my_msg = $_GET['msg'];
21   $my_topicid = $_GET['topicid'];
22   $my_poster = $_GET['poster'];
23
24   //construct SQL statement
25   $sqlstmt = "INSERT INTO messages VALUES('$my_msg','$my_topicid')";
26
27   //store message in database
28   $result = mysql_query($sqlstmt);
29   echo "Thank you $my_poster for using the message board";
30 }
31
32 function displayAllMessagesForTopic(){
33   if(!isset($_GET['topicid'])){
34     exit;
35   }
36
37   $my_topicid = $_GET['topicid'];
38
39   $sqlstmt = "SELECT msg FROM messages WHERE topicid='$my_topicid'";
40   $result = mysql_query($sqlstmt);
41
42   //display all messages
43   while($row = mysql_fetch_assoc($result)){
44     echo "Message " . $row['msg'];
45   }
46 }
```

Figure 1: Example PHP program that implements a simple message board using a MySQL database. This program is vulnerable to SQL injection and cross-site scripting attacks. Section 2.1 discusses the vulnerabilities. (For simplicity, the figure omits code that establishes a connection with the database.)

posting a message, the program constructs and submits the SQL statement to store the message in the database (lines 25 and 28) and then displays a confirmation message (line 29). In the displaying mode, the program retrieves and displays messages for the given topic (lines 39, 40, and 44).

This program is vulnerable to the following attacks, all of which our technique can automatically generate:

SQL injection attack. Both database queries, in lines 28 and 40, are vulnerable but we discuss only the latter, which exploits the lack of input validation for topicid.

Consider the following string passed as the value for input parameter topicid:

```
1' OR '1'='1
```

This string leads to an attack because the query that the program submits to the database in line 40,

```
SELECT msg FROM messages WHERE topicid='1' OR '1'='1
```

contains a tautology in the WHERE clause and will retrieve all messages, possibly leaking private information.

To exploit the vulnerability, the attacker must create an *attack vector*, i.e., the full set of inputs that make the program follow the exact path to the vulnerable mysql_query call and execute the attack query. In our example, the attack vector must contain at least parameters mode and topicid set to appropriate values. For example:

```
mode    → display
topicid → 1' OR '1'='1
```

First-order XSS attack. This attack exploits the lack of validation of the input parameter poster. After storing a message, the program displays a confirmation note (line 29) using the local variable my_poster, whose value is derived directly from the input parameter poster. Here is an attack vector that, when executed, opens a popup window on the user's computer:

```
mode    → add
topicid → 1
msg     → Hello
poster  → Villain<script>alert("XSS")</script>
```

This particular popup is innocuous; however, it demonstrates the attacker's ability to execute script code in the victim's browser (with access to the victim's session data and permissions). A real attack might, for example, send the victim's browser credentials to the attacker.

Second-order XSS attack. This attack exploits the lack of SQL validation of parameter msg when storing messages in the database (line 25) and the lack of HTML validation when displaying messages (line 44). The attacker can use the following attack vector to store the malicious script in the application's database.

```
mode    → add
topicid → 1
msg     → Hello<script>alert("XSS")</script>
poster  → Villain
```

Now *every* user whose browser displays messages in topic 1 gets an unwanted popup. For example, executing the following innocuous input results in an attack:

```
mode    → display
topicid → 1
```

3 Technique

Our attack-creation technique generates a set of concrete inputs, executes the program under test with each input, and dynamically observes whether data flows from an input to a sensitive sink (e.g., a function such as mysql_query or echo), including any data-flows that pass through a database. If an input reaches a sensitive sink, our technique modifies the input by using a library of attack patterns, in an attempt to pass malicious data through the program.

This section first shows the four components of our technique (Section 3.1) and then describes the algorithms for au-
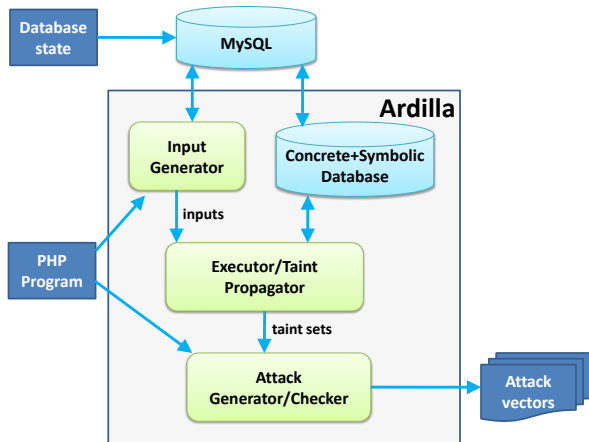
Figure 2: The architecture of ARDILLA. The inputs to ARDILLA are the PHP program and its associated MySQL database. The output is a set of attack vectors for the program, each of which is a complete input that exposes a security vulnerability.

tomatically generating first-order (Section 3.2) and second-order (Section 3.3) attacks.

3.1 Technique Components

Figure 2 shows the architecture of our technique and of the ARDILLA tool that we created as an implementation of the technique for PHP. Here, we briefly describe its four components as an aid in understanding the algorithms. Section 4 describes ARDILLA and the four components in detail.

- The **Input Generator** creates a set of inputs for the program under test, aiming to cover many control-flow paths.
- The **Executor/Taint Propagator** runs the program on each input produced by the input generator and tracks which parts (parameters) of the input flow into sensitive sinks. For each sensitive sink, the executor outputs a set of input parameters whose values flow into the sink (called a *taint set*).
- The **Attack Generator/Checker** takes a list of taint sets (one for each sensitive sink), creates candidate attacks by modifying the inputs in the taint sets using a library of SQLI and XSS attack patterns, and runs the program on the candidate attacks to determine (check) which are real attacks.
- The **Concrete+Symbolic Database** is a relational database engine that can execute SQL statements both concretely and symbolically. Our technique uses this component to track the flow of tainted data through the database, which is critical for accurate detection of second-order XSS attacks.

parameters: program $\mathcal{P}$, database state *db*
result   : SQLI or first-order XSS attack vectors
1   $attacks := \varnothing$;
2   **while** *not timeExpired()* **do**
3     $input := generateNewInput(\mathcal{P})$;
4     $\langle taints, db' \rangle := exec\&PropagateTaints(\mathcal{P}, input, db)$;
5     $attacks := attacks \cup gen\&CheckAttacks(taints, \mathcal{P}, input)$;
6   **return** *attacks*;

Figure 3: Algorithm for creating SQLI and first-order XSS attacks. Section 3.2 describes the algorithm.

3.2 First-order Attacks

Figure 3 shows the algorithm for generating SQLI and first-order XSS attacks (both called *first-order* because they do not involve storing malicious inputs in the database). The algorithms for creating SQLI and first-order XSS attacks are identical except for the sensitive sinks (`mysql_query` for SQLI, `echo` and `print` for XSS) and certain details in the attack generator/checker.

The algorithm takes the program $\mathcal{P}$ under test and its associated database *db* populated with the proper tables and initial data (usually done via an installation script or taken from an existing installation). Until a time limit expires, the algorithm generates new concrete inputs (line 3), runs the program on each input and collects taint sets (line 4), and then creates attack vectors (line 5).

Example. Here is how our technique generates the first-order XSS attack presented in Section 2.1:

First, new inputs are successively generated and the program executes on each input (and propagates taints) until some input allows the program to reach line 29 in the code in Figure 1, which contains the sensitive sink `echo`. An example of such an input *I* is:

```
mode     → add
topicid  → 1
msg      → 1
poster   → 1
```

(Even though only the value of `mode` determines whether execution reaches line 29, all parameters are required to be set; otherwise the program rejects the input in line 17. Our input generator picks 1 as the default "don't care" value.)

Second, the executor/taint propagator runs the program on *I* and creates taint sets for sensitive sinks. In the example, the executor marks all input parameters as tainted and determines that the value of the parameter `poster` flows into the local variable `my_poster`, which flows into the sensitive sink `echo` in line 29:

```
$my_poster = $_GET['poster'];
...
echo "Thank you $my_poster for using the message board";
```

Thus, the taint set of this `echo` call contains (only) the input parameter `poster`.

Third, the attack generator mutates the input *I* by replacing the value of all parameters in the taint set (here only `poster`) with XSS attack patterns. An example pattern is

parameters: program $\mathcal{P}$, database state db
result   : second-order XSS attack vectors

```
1   inputs := ∅;
2   attacks := ∅;
3   db_sym := makeSymbolicCopy(db);
4   while not timeExpired() do
5     inputs := inputs ∪ generateNewInput(P);
6     input₁ := pickInput(inputs);
7     input₂ := pickInput(inputs);
8     ⟨taints₁, db'_sym⟩ := exec&PropagateTaints(P, input₁, db_sym);
9     ⟨taints₂, db''_sym⟩ := exec&PropagateTaints(P, input₂, db'_sym);
10    attacks := attacks ∪ gen&CheckAttacks(taints₂, P, ⟨input₁, input₂⟩);
11  return attacks;
```

Figure 4: Algorithm for creating second-order XSS attacks. Section 3.3 describes the algorithm.

`<script>alert("XSS")</script>`. Picking this pattern alters input I into I':

```
mode      →  add
topicid   →  1
msg       →  1
poster    →  <script>alert("XSS")</script>
```

Fourth, the attack checker runs the program on I' and determines that I' is a real attack.

Finally, the algorithm outputs I' as an attack vector for the first-order XSS vulnerability in line 29 of Figure 1.

3.3   Second-order Attacks

Figure 4 shows the algorithm for generating second-order XSS attacks, which differs from the first-order algorithm by using a concrete+symbolic database and by running the program on two inputs during each iteration. The first input represents one provided by an attacker, which contains malicious values. The second input represents one provided by a victim, which does not contain malicious values. The algorithm tracks the flow of data from the attacker's input, through the database, and to a sensitive sink in the execution on the victim's innocuous input.

The algorithm takes the program $\mathcal{P}$ under test and a database db. In the first step (line 3), the algorithm makes a symbolic copy of the concrete database, creating a concrete+symbolic database. Then, until a time limit expires, the algorithm generates new concrete inputs and attempts to create attack vectors by modifying the inputs. The algorithm maintains a set of inputs generated so far (in the *inputs* variable), from which, in each iteration, the algorithm picks two inputs (lines 6 and 7). Then, the algorithm executes the two inputs in sequence (lines 8 and 9) using the concrete+symbolic database. The first execution (simulating the attacker) sets the state of the database (db'_{sym}) that the second execution (simulating the victim) uses. Finally, the attack generator/checker (line 10) creates second-order XSS attack scenarios (i.e., input pairs).

To favor execution paths that lead to second-order XSS attacks, on line 6 our implementation picks an input that executes a database write, and on line 7 picks an input that executes a database read on the same table.

Example. Here is how our technique generates the second-order XSS attack introduced in Section 2.1:

First, the input generator creates inputs and picks the following pair I_1:

```
mode      →  add
topicid   →  1
msg       →  1
poster    →  1
```

and I_2:

```
mode      →  display
topicid   →  1
```

Second, the executor/taint propagator runs the program on I_1, using the concrete+symbolic database. During this execution, the program stores the value 1 of the input parameter `msg` (together with the taint set that contains the parameter `msg` itself) in the database (line 25 of Figure 1).

Third, the executor/taint propagator runs the program on I_2, using the concrete+symbolic database. During this execution, the program retrieves the value 1 from the database (together with the value's stored taint set that contains `msg`) and outputs the value via the `echo` in line 44. `echo` is a sensitive sink, and its taint set contains the parameter `msg` from I_1. Thus, the algorithm has dynamically tracked the taint from `msg` to the local variable `my_msg` (line 20), *into the database* (line 28), back out of the database (line 40), into the `$row` array (line 43), and finally as a parameter to `echo` (line 44), across two executions.

Fourth, the attack generator uses the library of attack patterns to alter `msg` in I_1 to create an attack candidate input I'_1:

```
mode      →  add
topicid   →  1
msg       →  <script>alert("XSS")</script>
poster    →  1
```

Fifth, the attack checker runs the program, in sequence, on I'_1 and I_2 (note that I_2 remains unchanged), and determines that this sequence of inputs is an attack scenario.

Finally, the algorithm outputs the pair $\langle I'_1, I_2 \rangle$ as a second-order XSS attack scenario that exploits the vulnerability in line 44 of Figure 1.

4   The ARDILLA Tool

As an implementation of our technique, we created ARDILLA, an automated tool that generates concrete attack vectors for Web applications written in PHP. The user of ARDILLA needs to specify the type of attack (SQLI, first-order XSS, or second-order XSS), the PHP program to analyze, and the initial database state. The outputs of ARDILLA are attack vectors. This section describes ARDILLA's implementation of each component of the technique described in Section 3.

4.1   Dynamic Input Generator

The dynamic input generator creates inputs for the PHP program under test. Inputs for PHP Web applications are

Web server requests: their parameters are mappings from keys (strings) to values (strings and integers) in associative arrays such as $_GET[] and $_POST[].

ARDILLA uses the input-generation component from Apollo [1], but ARDILLA could potentially use any generator for PHP applications such as the one described by Wassermann et al. [30]. The Apollo input generator is based on systematic dynamic test-input generation that combines concrete and symbolic execution [11]. Here, we briefly describe this technique, which ARDILLA uses as a black box.

For each program input (starting with an arbitrary well-formed concrete input, and then using subsequently-generated ones), the input generator executes the program concretely and also collects symbolic constraints for each runtime value. These constraints describe an input that follows a given control-flow path through the program. Negating the symbolic constraint at a branch-point (e.g., an if statement) and discarding subsequent constraints gives a set of constraints for a different path through the program. The input generator then attempts to solve those constraints to create a concrete input that executes the new path. The input generator repeats this process for each branch-point in an execution, possibly generating many new inputs from each executed one.

4.2 Executor and Taint Propagator

The Executor and Taint Propagator runs the program under test on each input and tracks the dynamic data-flow of input parameters throughout the execution. For each sensitive sink, the executor outputs the set of input parameters, the taint set, whose values flow into the sink. ARDILLA's taint propagation is unique in that it can track the flow of tainted data through the database, by using a concrete+symbolic database (Section 4.4). Dynamic taint propagation in ARDILLA can be characterized by the following five components.

1. Taint sources give rise to tainted data during execution of the PHP program under test. Taint sources are inputs (e.g., $_GET and $_POST). ARDILLA assigns a unique taint to each value read from an input parameter, identified by the value's origin. For example, ARDILLA assigns taint msg to a value retrieved from $_GET['msg'].

2. Taint sets describe how each runtime value is influenced by taint sources, and can contain any number of elements. For example, taint set {msg, poster} may correspond to a runtime value derived from input parameters msg and poster (e.g., via string concatenation).

3. Taint propagation specifies how runtime values acquire and lose taint. ARDILLA propagates taint sets unchanged across assignments and procedure calls in application code. At a call to a built-in PHP function (e.g., chop, which removes trailing whitespace from a string) that is not a *taint filter* (see next component), ARDILLA constructs a taint set for the return value that is a union of taint sets for function argument values. ARDILLA also constructs taint

sets for string values created from concatenation by taking a union of taint sets for component strings. At a call to a database function (e.g., mysql_query), ARDILLA stores or retrieves taint for the data values. (Section 4.4 describes the interaction of taint propagation with the database.)

4. Taint filters are built-in PHP functions that are known to sanitize inputs (i.e., modify the inputs to make them harmless for XSS or SQLI attacks). For example, htmlentities converts characters to HTML entities (e.g., < to <) and makes the output safe from XSS attacks. At a call to a taint filter function, ARDILLA creates an empty taint set for the return value. A user of ARDILLA can optionally specify a list of taint filters.

5. Sensitive taint sinks are built-in PHP functions that are exploitable in XSS and SQLI attacks: for example, echo and print for XSS and mysql_query for SQLI. When reaching a call to a sensitive sink, ARDILLA records the taint sets of the argument, indicating a data-flow from the inputs to the sink, and thus a possibility of an attack.

ARDILLA's Executor and Taint Propagator[1] is implemented by modifying the Zend PHP interpreter to perform regular program execution and to simultaneously propagate taints from inputs to other runtime values.

4.3 Attack Generator and Checker

The attack generator creates candidate attack vectors that are variants of the given input. The attack checker determines whether a candidate is an attack, by comparing the candidate's execution to that of the original input.

The attack generator and checker ensure that ARDILLA creates concrete exploits, which are are much easier for programmers to fix than reports of abstract traces [4, 8].

ARDILLA generates candidate attack vectors and checks their validity. Not every flow of tainted data to a sensitive sink indicates a vulnerability. The data may flow through routines that check or sanitize it. ARDILLA generates candidate attack vectors by mutating innocuous inputs that demonstrate the data flow to sensitive sinks. ARDILLA checks the validity of the candidate attacks by comparing innocuous and candidate-attack executions.

4.3.1 Attack Generator

The attack generator starts with an input for which there is dataflow from a parameter to a sensitive sink. For each parameter whose value flows into the sink (member of the *taint set*), the generator creates new inputs that differ only for that parameter. The generator systematically replaces the value of that parameter by values taken from an *attack pattern library*—a set of values that may result in an attack if supplied to a vulnerable input parameter.

ARDILLA uses attack patterns developed by security professionals. ARDILLA's SQLI attack pattern library contains 6

_____

[1]http://www.zend.com

patterns distilled from several lists[2,3] Ardilla's XSS attack pattern library[4] contains 113 XSS attack patterns, including many filter-evading patterns (that use various character encodings, or that avoid specific strings in patterns).

Ardilla's goal is creating concrete exploits, not verifying the absence of vulnerabilities. Moreover, Ardilla checks every candidate attack input. Therefore, Ardilla is useful even given the pattern library's inevitable incompleteness (missing attack patterns), and potential unsoundness (patterns that do not lead to attacks).

The attack library needs to be integrated in Ardilla to be effective; the library alone is not enough to construct attacks. Ardilla constructs each attack input so that the execution reaches the vulnerable call site (using random values is ineffective [1]). In particular, the constructed attack inputs contain many key-value pairs and strings from the attack library constitute only 1 value in each attack input.

A string constraint solver is a potential replacement for the attack library. Given the string at the vulnerable call site, the solver could find a string that transforms the innocuous input into a malicious input. The solver needs to define 'malicious' formally, e.g., SQL query with a tautology in the WHERE clause. Fu et al. used a custom-made solver in a static analysis [9]. We have built a prototype string-constraint solver, Hampi [16], and preliminary results indicate that creating concrete attacks can be successfully reduced to generating and solving string constraints.

4.3.2 Attack Checker

In SQLI and XSS attacks, the PHP program interacts with another component (a database or a Web browser) in a way the programmer did not intend. The essence of an SQLI attack is a change in the structure of the SQL statement that preserves its syntactic validity (otherwise, the database rejects the statement and the attack attempt is unsuccessful) [27]. The essence of an XSS attack is the introduction of additional script-inducing constructs (e.g., <script> tags) into a dynamically-generated HTML page [29].

Ardilla detects attacks by looking for differences in the way the program behaves when run on two inputs: one innocuous and the other potentially malicious. We assume that the input generator creates innocuous (non-attack) inputs, since the input parameters' values are simple constants such as 1 or literals from the program text. Therefore, the innocuous input represents how the program is intended to interact with a component (database or browser). The attack generator creates the potentially malicious input.

---

[2]http://www.justinshattuck.com/2007/01/18/
mysql-injection-cheat-sheet,
http://ferruh.mavituna.com/sql-injection-cheatsheet-oku,
http://pentestmonkey.net/blog/mysql-sql-injection-cheat-sheet

[3]Ardilla's list omits attacks that transform one query into multiple queries, because the PHP mysql_query function only allows one query to be executed per call.

[4]http://ha.ckers.org/xss.html

msg	topicid	msg_s	topicid_s
Test message	1	∅	∅
Hello	2	{msg}	{topicid}

Figure 5: Example state of the concrete+symbolic database table `messages` used by the PHP program of Figure 1. Each concrete column (left-most two columns) has a symbolic counterpart (right-most two columns) that contains taint sets. The ∅ values represent empty taint sets.

The checker runs the program on the two inputs and compares the executions. Running the program on the attack candidate input avoids two potential sources of false warnings: (i) input sanitizing—the program may sanitize (i.e., modify to make harmless) the input before passing it into a sensitive sink. Ardilla does not require the user to specify a list of sanitizing routines. (ii) input filtering—the program may reject inputs that satisfy a malicious-input pattern (blacklisting), or else fail to satisfy an innocuous-input pattern (whitelisting). However, the taint sets are unaffected by control-flow (taint sets only reflect data-flow) and cannot capture input filtering.

The **SQLI attack checker** compares database statements (e.g., SELECT, INSERT) issued by the PHP program executed separately on the two inputs. The checker compares the first pair of corresponding statements, then the second, etc. The checker signals an attack if the statements in any pair are both valid SQL but have different syntactic structure (i.e., parse tree).

The **XSS attack checker** signals an attack if the HTML page produced from the execution of a candidate attack input (or sequence of inputs, for second-order attacks) contains additional script-inducing constructs.

4.4 Concrete+Symbolic Database

The concrete+symbolic database stores both concrete and symbolic values for each data record. In a PHP Web application, the database is shared state that enables the exchange of data between users. The concrete+symbolic database tracks the flow of user-provided data between *different runs* of the PHP program and is critical in creating second-order XSS attacks.

The concrete+symbolic database is implemented as a duplicate of the concrete database, with each table having additional columns that store symbolic data. Ardilla uses these columns to store taint sets, but it is also possible to store symbolic expressions there.

Figure 5 shows an example database state during the execution of the program in Figure 1. Assume the database was pre-populated with a test message in topic 1, so the taint sets for fields in the first row are empty. When the user posts a message Hello in topic 2 (line 28), the taint sets from the respective input parameters are stored along with their concrete values in the second row. Later, when the user fetches

205

data from that row (line 43), the taint sets are also fetched and propagated to the assigned variables.

ARDILLA dynamically rewrites each SQL statement in the PHP program to account for the new columns—either updating or reading taint sets, as appropriate. Our current implementation handles a subset of SQL, rewriting their strings before passing them into `mysql_query`: `CREATE TABLE`, `INSERT`, `UPDATE`, and (non-nested) `SELECT`. (Note that the `DELETE` statement and `WHERE` condition do not need to be rewritten—MySQL can locate the relevant rows using the concrete values.)

- `CREATE TABLE` creates a new table. ARDILLA rewrites the statement to add a duplicate for each column (e.g., the two right-most columns in Figure 5) to use for storing taint sets.
- `INSERT` adds new rows to tables. ARDILLA rewrites the statement to store taint sets in the duplicate columns. For example, consider the following PHP string representing an SQL statement (PHP automatically performs the string concatenation):

```
INSERT INTO messages VALUES('$_GET['msg']','$_GET['topicid']')
```

Consider an execution in which parameters `msg` and `topicid` have concrete values `Hello` and `2` and have one-element taint sets that contain only the parameters themselves. ARDILLA dynamically rewrites the statement as follows:

```
INSERT INTO messages VALUES('Hello','2', '{msg}','{topicid}')
```

- `UPDATE` modifies values in tables. For example, for:

```
UPDATE messages SET msg='$_GET['msg']'
 WHERE topicid='$_GET['topicid']'
```

ARDILLA's dynamic rewriting for `UPDATE` is similar to that for `INSERT` (the `WHERE` condition is unchanged):

```
UPDATE messages SET msg='Hi', msg_s='{msg}' WHERE topicid='3'
```

- `SELECT` finds and returns table cells. ARDILLA rewrites the statement to include the duplicate (symbolic) column names in the selection. Thereafter, ARDILLA uses the value retrieved from the duplicate column as the taint set for the concrete value retrieved from the original column. For example, consider the concrete statement executed in line 39 of the program in Figure 1 (given the example state of the concrete+symbolic database in Figure 5).

```
SELECT msg FROM messages WHERE topicid = '2'
```

ARDILLA rewrites the statement to:

```
SELECT msg, msg_s FROM messages WHERE topicid = '2'
```

The result of executing this rewritten statement on the table in Figure 5 is a 1-row table with concrete string `Hello` and associated taint set {msg}, in columns `msg` and `msg_s`. ARDILLA augments functions such as `mysql_fetch_assoc` to assign concrete values to the proper variables (e.g., `row` in line 43) and to simultaneously propagate their taint sets.

5 Evaluation

We evaluated ARDILLA on five open-source programs downloaded from `http://sourceforge.net`: school-mate 1.5.4 (tool for school administration, 8181 lines of code, or LOC), webchess 0.9.0 (online chess game, 4722 LOC), faqforge 1.3.2 (tool for creating and managing documents, 1712 LOC), EVE 1.0 (player activity tracker for an online game, 915 LOC), and geccbblite 0.1 (a simple bulletin board, 326 LOC). We used the latest available versions as of 5 September 2008.

We performed the following procedure for each subject program. First, we ran the program's installation script to create the necessary database tables. Second, we pre-populated the database with representative data (e.g., defaults where available). Third, we ran ARDILLA with a 30-minute time limit in each of three modes: SQLI, first-order XSS, and second-order XSS. The time limit includes all experimental tasks, i.e., input generation, program execution and taint propagation, and attack generation and attack checking. When necessary, we provided the input generator with (non-administrator) username and password combinations. Doing so poses no methodological problems because an attacker can use a legitimate account to launch an attack. Fourth, we manually examined attack vectors reported by ARDILLA to determine if they reveal true security vulnerabilities. We did not know any SQLI or XSS vulnerabilities in the subject programs before performing the experiments. (Thanks to previous studies [28, 29], we were *aware of* the presence of first-order XSS and SQLI vulnerabilities in **geccbblite** and **EVE**.)

We ran ARDILLA in two modes for checking validity of XSS attacks: lenient and strict. (The SQLI checker has only one mode.) In the lenient mode, the XSS checker reports a vulnerability when the outputs differ in script-inducing elements or HTML elements like `href`. In the strict mode, the XSS checker only reports a vulnerability when the outputs differ in script-inducing elements.

5.1 Measurements

Number of sensitive sinks (all) is the statically computed number of `echo`/`print` (for XSS) or `mysql_query` statements (for SQLI), whose parameter is *not a constant string*.
Number of reached sinks (reach) on all generated inputs is an indication of coverage achieved by the input generator. This measure is suitable for ARDILLA, because ARDILLA looks for attacks on sensitive sinks.
Number of tainted sinks (taint) is the number of sensitive sinks reached *with non-empty taint sets* during execution. Each such occurrence *potentially* exposes a vulnerability, which ARDILLA uses the attack generator and checker to test.
Number of verified vulnerabilities (Vuln): We count at most *one* vulnerability per sensitive sink, since a single-line code-fix would eliminate all attacks on the sink. If a single

program	mode	sensitive sinks			lenient		strict	
		all	reach	taint	Vuln	F	Vuln	F
schoolmate	SQLI	218	28	23	**6**	0	**6**	0
	XSS1	122	26	20	**14**	6	**10**	0
	XSS2	122	4	4	**4**	0	**2**	0
webchess	SQLI	93	42	40	**12**	0	**12**	0
	XSS1	76	39	39	**13**	18	**13**	0
	XSS2	76	40	0	**0**	0	**0**	0
faqforge	SQLI	33	7	1	**1**	0	**1**	0
	XSS1	35	10	4	**4**	0	**4**	0
	XSS2	35	0	0	**0**	0	**0**	0
EVE	SQLI	12	6	6	**2**	0	**2**	0
	XSS1	24	5	4	**2**	0	**2**	0
	XSS2	24	5	3	**3**	0	**2**	0
geccbblite	SQLI	10	8	6	**2**	0	**2**	0
	XSS1	17	17	11	**0**	0	**0**	0
	XSS2	17	17	5	**5**	0	**4**	0
Total	SQLI	366	91	76	**23**	0	**23**	0
	XSS1	274	97	78	**33**	24	**29**	0
	XSS2	274	66	12	**12**	0	**8**	0

Figure 6: Results of running ARDILLA to create SQLI, XSS1 (first-order XSS), and XSS2 (second-order XSS) attacks. The lenient and strict columns refer to ARDILLA modes (Section 5). Section 5.1 describes the remaining columns (Vuln columns in bold list the discovered real vulnerabilities).

attack vector attacks multiple sensitive sinks, then we examine and count each vulnerability separately. This number does not include false positives.

Number of false positives (F): We manually inspected each ARDILLA report and determined whether it really constituted an attack (i.e., corruption or unintended disclosure of data for SQL, and unintended HTML structure for XSS). For second-order XSS, we checked that the attacker's malicious input can result in an unintended Web page for the victim.

5.2 Results

ARDILLA found 23 SQLI, 33 first-order XSS, and 12 second-order XSS vulnerabilities in the subject programs (see Figure 6). The attacks that ARDILLA found, as well as the attack patterns we used, are available at `http://pag.csail.mit.edu/ardilla`.

We examined two of the three instances in which ARDILLA found no vulnerabilities. In **geccbblite**, we manually determined that there are no first-order XSS vulnerabilities. In **faqforge**, we manually determined that each database write requires administrator access, so there are no second-order XSS vulnerabilities. (We did not manually inspect **webchess** for second-order XSS attacks, due to the program's size and our unfamiliarity with the code.)

We examined all 23 SQLI reports issued by ARDILLA and found no false positives. All attacks involved disrupting the SQL WHERE clause. In 4 cases, attacks result in data corruption; in 19 cases, attacks result in information leaking, sometimes as serious as bypassing login authentication.

We examined all 69 (33+24+12) unique XSS reports issued by ARDILLA. We found 24 false positives in the lenient

mode for first-order XSS (42% false-positive rate), and 0% percent false-positive rate for all other cases: strict first-order XSS, lenient and strict second-order XSS.

The attack generator and checker show which potential vulnerabilities are exploitable. We examined cases of tainted sinks for which ARDILLA did not create attacks. The most common reason is that the same think may not be reachable with a malicious input because of control-flow filtering that dynamic tainting does not capture. Such reachability is very hard to determine manually from program text. ARDILLA's concrete attacks directly demonstrate the vulnerability and how to exploit it. Our prototype string solver [16] did not discover more attacks, which indicates that the attack pattern library was sufficiently complete for our subjects.

Example created SQLI attack. In **webchess**, ARDILLA found a vulnerability in `mainmenu.php` that allows an attacker to retrieve information about all players without entering a password. The application constructs the vulnerable statement directly from user input:

```
"SELECT * FROM players WHERE nick = '" . $_POST['txtNick']
. "' AND password = '" . $_POST['pwdPassword'] . "'"
```

The attack vector contains the following two crucial parameters (others omitted for brevity)

```
ToDo     → NewUser
txtNick  → foo' or 1=1 --
```

which causes execution to construct the following malicious SQL statement which bypasses authentication (`--` starts an SQL comment):

```
SELECT * FROM players WHERE nick = 'foo' or 1=1 --
' AND password = ''
```

Comparison to previous studies. Two of our subject programs were previously analyzed for vulnerabilities. In **geccbblite**, a previous study [29] found 1 first-order XSS vulnerabilities, and 7 second-order XSS vulnerabilities (possibly including false positives). However, ARDILLA and our manual examination of **geccbblite** found no first-order XSS vulnerabilities. In **EVE**, another study [28] found 4 SQLI vulnerabilities. The result data from neither study are available so we cannot directly compare the findings.

Comparison to black-box fuzzing. We compared ARDILLA's ability to find first-order XSS attacks to that of a black-box fuzzer for finding XSS attacks: Burp Intruder[5] (listed in the 10 most popular Web-vulnerability scanners[6]). We configured the fuzzer according to its documentation. The fuzzer requires manual setting up of HTTP request patterns to send to the Web application (and requires manual indication of variables to mutate). We ran the fuzzer using the same attack pattern library that ARDILLA uses, and on the same subject programs. (We have not been able to successfully configure **webchess** to run with the fuzzer.) We

---

[5] `http://portswigger.net/intruder`
[6] `http://sectools.org/web-scanners.html`

ran the fuzzer until completion (up to 8 hours). The fuzzer found 1 first-order XSS vulnerability in **schoolmate**, 3 in **faqforge**, 0 in **EVE**, and 0 in **geccbblite**. All 4 vulnerabilities reported by the fuzzer were also discovered by ARDILLA.

Limitations. ARDILLA can only generate attacks for a sensitive sink *if* the input generator creates an input that reaches the sink. However, effective input generation for PHP is challenging [1,21,30], complicated by its dynamic language features and execution model (running a PHP program often generates an HTML page with forms and links that require user interaction to execute code in additional files). In particular, the generator that ARDILLA uses can create inputs only for one PHP script at a time and cannot simulate sessions (i.e., user–application interactions that involve multiple pages), which is a serious hindrance to achieving high coverage in Web applications; line coverage averaged less than 50%. In fact, only on *one* application (**webchess**) did the input generator run until the full 30-minute time-limit—in all other cases, the generator finished within 2 minutes because it could not manage to cover more code. We also attempted to run the generator on a larger application, the **phpBB** Web-forum creator (35 KLOC), but it achieved even lower coverage (14%). ARDILLA uses the input generator as a black box and any improvement in input generation is likely to improve ARDILLA's effectiveness.

6 Related Work

We describe previous approaches to securing Web applications from input-based attacks.

Defensive coding relies on special libraries to create safe SQL queries [6, 22]. Defensive coding can, in principle, prevent all SQLI attacks. The technique is suitable for new code. However, it requires rewriting existing code, while our technique requires no change to the programming language, the libraries, or the application.

Static approaches can, in principle, prove the *absence* of vulnerabilities [12, 19, 28, 29, 31]. In practice, however, analysis imprecision causes false warnings. Additionally, static techniques do not create concrete attack vectors. In contrast, our technique does not introduce such imprecision, and it creates attack vectors.

Dynamic monitoring aims to prevent SQLI attacks by tracking user-provided values [13, 24, 25, 27] during operation of a deployed application. However, dynamic monitoring does not help to remove errors before software deployment, and requires either modifying the application, or running a modified server. For example, CANDID [3] modifies the application source and requires changing the runtime system, with performance overhead of up to 40% on the production application.

Information-flow control restricts the flow of information between pieces of software, either statically [26] or dynamically [17, 32]. Information-flow control enforces confidentiality and integrity policies on the data and prevents attacks that use inappropriate information flows. However, some SQLI and XSS attacks abuse legitimate information flows; the SQL queries or the JavaScript can be dynamically generated and can depend on legal user input. Information-flow control requires modifying the application and either the operating system and the libraries, or the programming language. System-level techniques may have runtime performance overhead up to 40% [17].

Static and dynamic approaches can be combined [14,15]. Lam et al. [18] combine static analysis, model checking, and dynamic monitoring. QED [20] combines static analysis and model checking to automatically create SQLI and first-order XSS attacks on Java applications. In contrast to ARDILLA, QED (i) does not target second-order XSS, and (ii) requires programmers to use a custom specification language to describe attacks.

Saner [2] combines static and dynamic analyses to find potential XSS and SQLI vulnerabilities. Saner focuses on the sanitization process and abstracts away other details of the application, i.e., Saner creates attack vectors only for extracted, possibly infeasible, paths from the static dependency graph (Saner does dynamically validate the exploitability of string-manipulating code from those paths, but ignores control flow). Saner also reports a vulnerability whenever a path from source to sink contains no custom sanitation. The path, however, may be infeasible or not exploitable. Saner tests each source-to-sink path independently and may miss attacks in which output is constructed from multiple sinks. To detect attacks, Saner simply searches for specific strings in the output, whereas ARDILLA compares the structure of HTML or SQL between innocuous and attack runs.

Apollo [1] generates test inputs for PHP, checks the execution for crashes, and validates the output's conformance to HTML standards. The goal of ARDILLA is different: to find security vulnerabilities. ARDILLA uses the test-input generator subcomponent of Apollo as a black box. ARDILLA's taint propagation implementation is partially based on that of Apollo, but we enhanced it significantly by adding propagation across function calls, taint filters, taint sinks, and tracing taint across database calls.

Emmi et al. [7] model a database using symbolic constraints and provide a custom string solver to create database states that help exercise various execution paths in the Web application. Our work differs in objective (finding security vulnerabilities vs. improving test coverage) and in the targeted language (PHP vs. Java).

Wassermann et al.'s tool [30] executes a PHP application on a concrete input and collects symbolic constraints. Upon reaching an SQL statement, the tool attempts to create an input that exposes an SQL injection vulnerability, by using a string analysis [23]. The tool has re-discovered 3 previously known vulnerabilities. The most important differences between Wassermann's work and ours are: (i) Their tool has not discovered any previously unknown vulnerabilities, and

requires a precise indication of an attack point. Our tool has discovered 68 previously unknown vulnerabilities and requires no indication of vulnerable points. (ii) Their technique focuses on SQLI, while ours targets both SQLI and XSS. (iii) Their tool performs source-code instrumentation and backward-slice computation by re-executing and instrumenting additional code. Our tool works on unchanged application code. (iv) Their tool requires manual loading of pages and supplying of inputs to the page, while ours is fully automatic.

7   Conclusion

We have presented a technique for creating SQL injection and cross-site scripting (XSS) attacks in Web applications and an automated tool, ARDILLA, that implements the technique for PHP. Our technique is based on input generation, dynamic taint propagation, and input mutation to find a variant of the input that exposes a vulnerability. Using a novel concrete+symbolic database to store taint, ARDILLA can effectively and accurately find the most damaging type of input-based Web application attack: stored (second-order) XSS. A novel attack checker that compares the output from running on an innocuous input and on a candidate attack vector allows ARDILLA to detect vulnerabilities with high accuracy. In our experiments, ARDILLA found 68 attack vectors in five programs, each exposing a different vulnerability, with few false positives.

Acknowledgements

References

[1] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in dynamic Web applications. In *ISSTA*, 2008.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *S&P*, 2008.

[3] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In *CCS*, 2007.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *FSE*, 2008.

[5] Cenzic. Application security trends report Q1 2008. http://www.cenzic.com.

[6] W. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE*, 2005.

[7] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[8] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *VMCAI*, 2004.

[9] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *COMPSAC*, 2007.

[10] P. Godefroid. The soundness of bugs is what matters (position statement). In *BUGS*, 2005.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[12] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, 2004.

[13] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. *IEEE TSE*, 34(1):65, 2008.

[14] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *ASE*, 2005.

[15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *WWW*, 2004.

[16] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for string constraints. Technical Report MIT-CSAIL-TR-2009-004, MIT, 2009.

[17] M. Krohn, M. Brodsky, M. Kaashoek, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[18] M. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web applications with static and dynamic information flow tracking. In *PEPM*, 2008.

[19] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security*, 2005.

[20] M. Martin and M. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *USENIX Security*, 2008.

[21] S. McAllister, E. Kirda, and C. Krügel. Leveraging user interactions for in-depth testing of Web applications. In *RAID*, 2008.

[22] R. McClure and I. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *ICSE*, 2005.

[23] Y. Minamide. Static approximation of dynamically generated Web pages. In *WWW*, 2005.

[24] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[25] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2005.

[26] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications*, 2003.

[27] Z. Su and G. Wassermann. The essence of command injection attacks in Web applications. In *POPL*, 2006.

[28] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.

[29] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, 2008.

[30] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for Web applications. In *ISSTA*, 2008.

[31] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*, 2006.

[32] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI*, 2008.