Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications

Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna

Department of Computer Science,
University of California Santa Barbara
Santa Barbara, CA 93106-5110, USA
{marco,balzarot,rusvika,vigna}@cs.ucsb.edu

Abstract

In recent years, web applications have become tremendously popular, and nowadays they are routinely used in security-critical environments, such as medical, financial, and military systems. As the use of web applications for critical services has increased, the number and sophistication of attacks against these applications have grown as well. Most approaches to the detection of web-based attacks analyze the interaction of a web application with its clients and back-end servers. Even though these approaches can effectively detect and block a number of attacks, there are attacks that cannot be detected only by looking at the external behavior of a web application.

In this paper, we present Swaddler, a novel approach to the anomaly-based detection of attacks against web applications. Swaddler analyzes the internal state of a web application and learns the relationships between the application's critical execution points and the application's internal state. By doing this, Swaddler is able to identify attacks that attempt to bring an application in an inconsistent, anomalous state, such as violations of the intended workflow of a web application. We developed a prototype of our approach for the PHP language and we evaluated it with respect to several real-world applications.

Keywords: Web Attacks, Anomaly Detection, Dynamic Analysis, Code Instrumentation

1 Introduction

Web applications are quickly becoming the most common way to access services and functionality. Even applications such as word processors and spreadsheets are becoming web-based because of the advantages in terms of ubiquitous accessibility and ease of maintenance.

However, as web applications become more sophisticated, so do the attacks that exploit them. Some of these attacks are evolutions of well-known attacks, such as buffer overflows or command injections. In addition, there are attacks that are specific to web applications, such as forceful browsing and parameter manipulation.

Web applications are usually implemented as a number of server-side components, each of which can take a number of parameters from the user through both the request parameters (e.g., an attribute value) and the request header (e.g., a cookie). These components need to share and maintain state, so that the application can keep track of the actions of a user as he/she interacts with the application as a whole.

There are several attacks that exploit erroneous or inconsistent state management mechanisms in order to bypass authentication and authorization checks. Unfortunately, even though there are a number of tools and techniques to protect web applications from attacks, these approaches analyze the external behavior of an application, such as its request/response flow [1, 2] or its interaction with back-end databases [3, 4, 5], and do not take into account the *internal state* of a web application in order to identify anomalous or malicious behavior.

In this paper, we present Swaddler, a novel approach to the detection of attacks against web applications. The approach is based on a detailed characterization of the internal state of a web application, by means of a number of anomaly models. More precisely, the internal state of the application is monitored during a learning phase. During this phase the approach derives the profiles that describe the normal values for the application's state variables in critical points of the application's components. Then, during the detection phase, the application's execution is monitored to identify anomalous states.

The approach has been implemented by instrumenting the PHP interpreter and has been validated against real-world applications. Our experiments show that by modeling the internal state of a web application one can detect attacks that cannot be identified by examining the external flow of requests and responses only. For example, attacks that violate the intended workflow of an application cannot be detected by examining requests and responses in isolation.

The contributions of our paper are the following:

- We introduce a novel approach that analyzes the internal state of a web application using anomaly detection techniques. To the best of our knowledge, there are no other approaches that are able to analyze a web application's state at the granularity that our approach supports.
- We show that anomaly detection based on both the value of single variables and the relationships among multiple variables is an effective way to detect complex attacks against web applications.
- We demonstrate that our technique is able to detect attacks that mainstream techniques based on request analysis are unable to detect, such as workflow violations.

The rest of this paper is structured as follows. In Section 2, we describe our threat model and the type of attacks that our approach detects using a sample application. Then, in Section 3, we present our approach to modeling the state of a web application to detect complex attacks. In Section 4, we describe the implementation of our tool, and, in Section 5, we present a number of experiments that we carried out to evaluate its effectiveness. Finally, Section 6 presents related work and Section 7 concludes.

2 Threat Model

Web applications are the target of many different types of attacks. In this section, we present two common classes of attacks: the ones that exploit errors in the input validation process and the ones that exploit flaws in the enforcement of an application's intended workflow.

2.1 Input Validation Attacks

Input validation attacks exploit the application's inability to properly filter the values of the parameters provided by the user, allowing an attacker to inject malicious data (e.g., a piece of JavaScript code) into a web application. In particular, the two most common attacks that belong to this category are SQL injection and cross-site scripting (XSS).

A web application is vulnerable to a SQL injection attack when the input provided by the user is used to compose a database query without being previously sanitized. A SQL injection attack can allow a malicious user to execute arbitrary queries on the database server. As a result, the attacker can steal sensitive information and/or modify the information stored in the database tables.

Consider, for example, a typical web application where the authentication module compares the user-provided credentials with the known accounts contained in the database. The username provided by the user is used to compose the query, without any checks on its contents:

```
"SELECT * FROM users WHERE name = '" + userName + "';"
```

Since the username is not sanitized, it can be crafted by the attacker so that arbitrary SQL code is injected into the query. For example, if the value of the user name is set to `';DROP TABLE users;`, the database would evaluate the `DROP` query just after the `SELECT` one.

In cross-site scripting attacks, an attacker is able to force a user's web browser to evaluate attacker-supplied code (typically JavaScript) in the context of a trusted web site. The goal of these attacks is to circumvent the browsers' *same-origin policy*, which prevents scripts or documents loaded from one site from getting or setting the properties of documents originating from other sites.

In a typical XSS attack, the attacker inserts the malicious code as part of a message that is stored by the web application (e.g., JavaScript code is added to a blog comment). When a normal user accesses the page that shows the message, the malicious code is evaluated by the user's browser under the assumption that it originates from the vulnerable application rather than from the attacker. Therefore, the malicious code has access to sensitive information associated with the trusted web site, such as login credentials stored in cookies.

SQL injection and XSS attacks are very common and very dangerous at the same time. However, there is a large number of static and dynamic techniques to detect this kind of input validation attacks [6, 5, 7, 8, 9, 10, 11]. For this reason, in this paper we concentrate on the less-known and often-overlooked attacks against the workflow of web applications.

2.2 Workflow Violation Attacks

Workflow violation attacks exploit logical errors in web applications in order to bypass the intended workflow of the application. The intended workflow of a web application represents a model of the expected user interactions with the application. Examples of workflow violation attacks include authentication and authorization bypass, parameter tampering, and code inclusion attacks.

To better illustrate this class of attacks, we present a small PHP application that contains a number of common workflow vulnerabilities. The application is a simple online store that sells different items to its users. New users can register and, once logged in, browse and buy the items available in the store. The application uses the standard PHP session mechanism [12] to store the session information and the shopping carts of the users. In addition, the store provides an administrative interface to manage the inventory and to review information about its users.

The first example of vulnerability contained in the store application is an authentication bypass vulnerability. The program uses two session variables, `loggedin` and `username`, to keep track of whether a user is currently logged in and if she has administrative privileges. A simplified version of the code of `login.php`, the application module that initializes these variables, is shown in Figure 1. Every time the user requests one of the administrative pages, the variables `loggedin` and `username` are checked, as shown in `viewusers.php` in Figure 1, to verify that the user is correctly authenticated as administrator.

Since the application utilizes the PHP session mechanism, the session variables are kept inside the superglobal `_SESSION` array. However, if the `register_globals` option of the PHP interpreter is enabled, an application can refer to a session variable by simply using the variable name, as if it was a normal global variable. Since our application uses this "shortcut" to access the session variables (unfortunately a common practice among inexperienced developers), an attacker can easily bypass the checks by providing the required variables as part of the GET request to one of the protected pages. In fact, when

```
include ’config.php’;
session_start();
$username = $_GET[’username’];
$password = $_GET[’password’];
if($username == $admin_login
   && $password == $admin_pass) {
  $_SESSION[’loggedin’] = ’yes’;
  $_SESSION[’username’] = $admin_login;
} else if(checkuser($username,$password)) {
    $_SESSION[’loggedin’] = ’yes’;
    $_SESSION[’username’] = $username;
} else {
  diefooter("Login failed");
}
```

login.php

```
include ’config.php’;
session_start();
if($loggedin != ’yes’
   || $username != $admin_login) {
  diefooter("Unathorized access");
}
printusers();
```

admin/viewusers.php

Figure 1: Authentication bypass vulnerability in the store application.

register_globals is enabled, the PHP interpreter automatically binds the parameters coming from the user’s requests to global variables. Thus, if the variable loggedin is not present in the session (i.e, the user did not authenticate herself), it can be provided by an attacker using the request parameters, as shown in the following request:

```
http://store.com/admin/viewusers.php?loggedin=yes&username=admin
```

The login.php module is the only part of the application that sets the loggedin and username variables, and, as it can be seen from the code given in Figure 1, the variable username is set to the administrator’s name only if a user provides the correct administrator’s name and password. Thus, even if the attacker manages to bypass the authorization check in viewusers.php, she will not be able to set the _SESSION[’username’] to the correct value. Thus, the attack would force the application to move into an anomalous state (corresponding to the administrative code being executed with the _SESSION[’username’] not set to the expected *admin* value).

The store application is also vulnerable to a second workflow violation attack. In this case, the attack exploits the fact that the application computes the amount of money to be charged to the user’s credit card in several different steps. During the checkout phase, the user navigates through several pages where she has to provide various pieces of information, including her state of residency (for tax calculation), shipping address, shipping method, and credit card number. For simplicity, suppose that the checkout process consists of four main steps as shown in Figure 2, where the first three steps calculate the purchase total based on the user-provided information and the final step proceeds with the order submission. Now, suppose that the application fails to enforce the policy that the *Step 3* page should be accessible only after *Step 2* has been completed. As a result of this flaw, an attacker can directly go from *Step 1* to *Step 3* by simply entering the correct URL associated with *Step 3*. In this case, the total amount charged to the attacker’s credit card will be equal to the shipping cost only.

It is important to note that while this attack would be very difficult to detect analyzing the HTTP traffic, it clearly manifests itself as an anomaly in the web application state. In fact, under a normal operation, the total amount charged to the user is always equal to the sum of the purchased price, taxes and shipping cost. However, if the user is able to change the order of the steps in the checkout process, this relationship will not hold.

Finally, the store application contains an example of a parameter tampering vulnerability. When a user chooses a shipping method from a select box, the name of the shipping method and its cost, which are set as hidden parameters in the form, are submitted to the application. Unfortunately, the application fails to make additional server-side checks for the shipping costs, and, as a result, an attacker can set the cost of

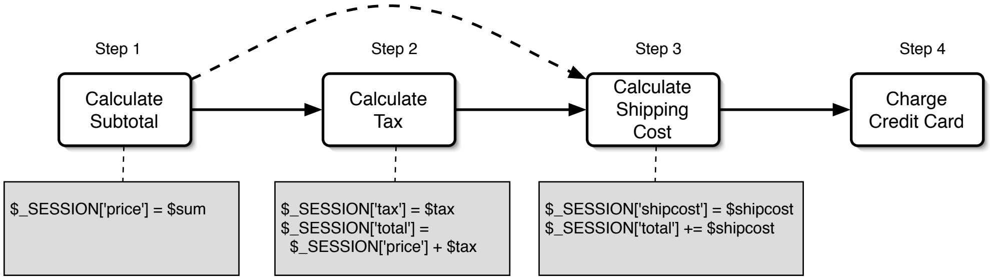
Step 1
Step 2
Step 3
Step 4
Calculate Subtotal
Calculate Tax
Calculate Shipping Cost
Charge Credit Card
$_SESSION['price'] = $sum
$_SESSION['tax'] = $tax
$_SESSION['total'] = $_SESSION['price'] + $tax
$_SESSION['shipcost'] = $shipcost
$_SESSION['total'] += $shipcost

Figure 2: Checkout workflow in the store application.

the chosen shipping method to an arbitrary value. This vulnerability is characterized by the fact that, in a normal execution, the variable containing the shipping cost always assumes the same values, depending on the shipping method that has been selected by the user. Thus, if an attacker tampers with the hidden parameter to change the shipping cost to an arbitrary value, the `_SESSION['shipcost']` variable will assume an anomalous value that can be easily detected analyzing the state of the application.

3 Approach

As shown in the previous section, not all web-based attacks rely on sending malicious input to web applications. Some of them exploit weaknesses in the intended workflow of the application, allowing the user to navigate through the different application's modules in a way that leads the application to an insecure state. In this case, the attacker performs a sequence of actions in which all the provided input values can be perfectly harmless, and the vulnerability is exploited through the particular order (or timing) of the various requests.

This type of attacks can be very difficult to detect "from the outside," that is, using sensors that only analyze HTTP requests and responses in isolation. Nevertheless, regardless of how the attack is performed, its final effect is to force the application to enter an insecure state. For this reason, we believe that a more effective approach to the detection of workflow attacks consists of monitoring, at runtime, the state of the web application "from the inside." This is true, of course, under the assumption that there is a strong relationship between insecure and anomalous states, i.e., any insecure state is also likely anomalous and vice versa.

Before describing our approach, which we call Swaddler, we need to introduce the concept of *web application state*. We define the state of a web application at a certain point in the execution as the information that survives a single client-server interaction: in other words, the information associated with the *user session*. Part of this information is kept on the server, while part of it can be sent back and forth between the user's browser and the server in the form of cookies, hidden form fields, and request parameters.

Given this definition of application state, it is possible to associate each instruction of the application with a model of the state in which that instruction is normally executed. For example, code contained in the `admin` directory of our sample application shares the fact that, when it is executed, the variable `_SESSION['username']` should always be equal to *admin*. Any runtime violation of this requirement represents the evidence that a low-privilege user was able to access an administrative functionality bypassing the constraints implemented by the application's developer.

Ideally, a complete set of these relationships among code execution points and state variables would be provided by the developers as part of the application's specification. However, since in reality this information is never explicitly provided, the models of the normal state for each program instruction have to be

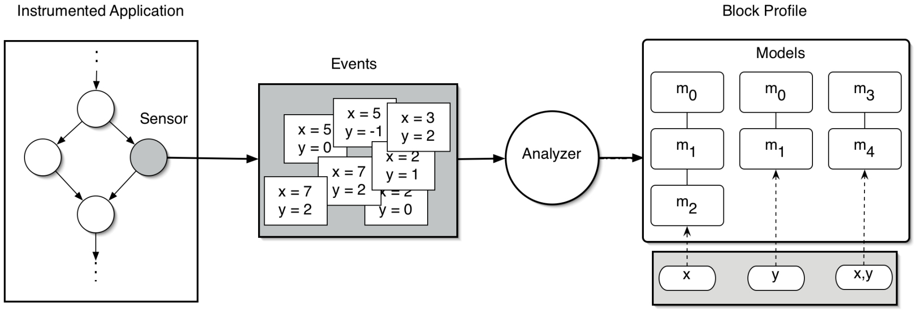

Figure 3: Description of the training phase.

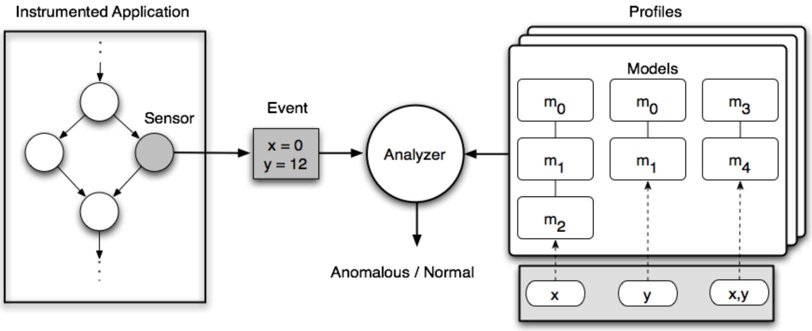

Figure 4: Description of the detection phase.

inferred from a set of attack-free execution traces. To perform this task, we propose to automatically instrument the web application with the code required to extract the runtime values of state variables. Depending on the language in which the application is developed, this instrumentation can be performed in several ways. For example, in our prototype implementation we decided to add the instrumentation as a module to the PHP interpreter. This solution allows our approach to be applied to a large set of web applications without the need to modify the source code of the applications.

Swaddler associates a model of the web application state with each instruction executed by the application. However, this solution can be optimized by limiting the instrumentation to the first instruction of each basic block. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [13]. In fact, since the control-flow inside a basic block is a simple sequence of instructions without branches, the application state at the beginning of the basic block univocally determines the state of each instruction inside the block. Once the models that describe the normal state associated with each basic block have been properly extracted, they can be used to detect (and prevent) attacks that violate the normal application state.

Figure 3 and Figure 4 show the architecture of our anomaly detection system during the training and detection phases. Swaddler consists of two main components: the *sensor* and the *analyzer*. The sensor is represented by the instrumentation code, which collects the application's state data (i.e., the values of state variables) at the beginning of each basic block, and encapsulates them in an *event* that is sent to the analyzer. An event generated by the sensor defines a mapping between the variable names and their current values. For each basic block of the application, the analyzer maintains a *profile*, i.e., a set of statistical models used to characterize certain features of the state variables. These models can be used to capture various properties of single variables as well as to describe complex relationships among multiple variables associated with a block. In training mode, profiles for application blocks are established using the events generated by the sensor, while in detection and prevention modes these profiles are used to identify anomalous application states. When an anomalous state is detected, the analyzer raises an alert message, and, optionally, it can immediately stop the execution of the application.

Since Swaddler is based on anomaly detection techniques, it can be vulnerable to mimicry attacks [14], in which an attacker crafts an exploit in a way that closely resembles normal activity (or normal state values). Therefore, in principle, one could find a way to perform an attack that brings a web application into an insecure state without triggering any alert. Nonetheless, the fine granularity at which our approach analyzes the application state makes this type of attacks much more difficult to perform.

Even though our approach is general and, in principle, can be applied to other languages and execution environments, in the following sections we will describe in detail the solution that we have developed for the PHP language.

4 Implementation

The implementation of our approach consists of two main components: the *sensor*, which is an extension of the PHP interpreter that probes the current state of an executing application, and the *analyzer*, which is an anomaly-based system that determines the normality of the application's state. In the current prototype, the sensor is implemented as a module of the open-source Zend Engine interpreter [15] and the analyzer is built on top of the `libAnomaly` framework [16].

4.1 Event Collection

The Zend Engine is the standard interpreter for the PHP language. It implements a virtual machine that is responsible for parsing programs written in PHP and compiling them into an intermediate format, which is then executed.

To probe an application's state and generate responses to detected attacks, our implementation extends the Zend Engine in two points of its processing cycle: after the standard compilation step is completed and before the standard execution step is initiated. Whenever the execution of a PHP script is requested (e.g., in response to a user's request to a web application), the Zend Engine parses the script's source code, checks for its correctness, and compiles it into a sequence of statements in an intermediate, architecture-independent language. The binary representation of each statement holds a reference to a handler function, which interprets the statement and changes the state of the virtual machine accordingly. During execution, the Zend Engine decodes in turn each statement and dispatches it to its handler function.

Our extension is invoked by the engine after it has produced the compiled version of a program. The extension performs a linear scan of the sequence of intermediate statements, identifies the corresponding basic blocks, and associates a unique ID with each of them. The first statement of each basic block is modified by overwriting its handler with a custom instrumentation handler. The effect of this modification is that, during the program's execution, our instrumentation code is invoked each time a basic block is entered.

The cost of this phase is linear in the number of intermediate statements, which is proportional to the size of the program. By default, the Zend Engine does not cache the intermediate statements for reuse during subsequent executions, and, therefore, the compilation and our instrumentation of the application's code is repeated for every access of the page. However, if one of the available caching mechanisms is added to the standard engine, our technique will be able to take advantage of the caching functionality, thus reducing the cost of the instrumentation.

After the compilation step, the Zend Engine starts executing the application's code. Our instrumentation handler is invoked every time the first statement of a basic block is executed. The instrumentation handler creates an event corresponding to the basic block being executed and makes the event available to the analyzer for inspection. The event contains the information collected about the current application state. Since in the current implementation of our approach we focus on the detection of workflow-based attacks, by default the events contain the values of the variables defined in the application's session, i.e., the content of the `_SESSION` array. In our experiments we found this information to be sufficient to detect most of the workflow attacks. However, the system can be configured to extract other parts of the application's state. For example, one could use the content of the `_REQUEST` array (a global array automatically populated by the interpreter with the values of the user's input to the application) to detect attacks that exploit insufficient validation of input parameters. In addition, further customizations of the sensor are possible if more information is known about an application. For example, if one knows that only some portions of the application's state can be used to attack the application, one could configure the sensor to only extract those parts of the state. Note that all the configurable settings of the sensor (e.g., the set of state variables to extract or the models to use) are set by using the standard `.htaccess` mechanism and, therefore, can be changed on-the-fly without the need to modify the sensor's code.

After delivering an event to the analyzer, the instrumentation handler behaves differently depending on the current execution mode. During training, it takes no further action. However, if the system is in detection or prevention mode, and it has determined that the state associated with the block about to be executed is anomalous, an alert is generated, the request is logged, and, in prevention mode, the execution is automatically blocked.

When the sensor has finished its processing (and the execution has not been abnormally terminated), it invokes the original handler of the statement, passing the control back to the Zend Engine. The execution of the statement, then, proceeds as in the normal, unmodified interpreter.

Our implementation, based on the modification of the web application's interpreter, has several strengths. First, the sensor has direct access to all of the interpreter's data structures, and, thus, it has an unambiguous view of the application's state. Other implementation strategies of our approach, e.g., those based on the analysis of request/response traces, would have to infer the application's state and, thus, in general, would provide a less precise input to the analyzer component. Second, the sensor has the capability of blocking attacks before they reach a vulnerable point in the application.

4.2 Anomaly Detection

Our implementation of the detection engine is based on a modified version of the `libAnomaly` framework [16]. The anomaly detection process uses a number of different models to identify anomalous states for each basic block of a web application. A model is a set of procedures used to evaluate a certain feature of a state variable associated with the block (e.g., the range of its possible values) or a certain feature that involves multiple state variables (e.g., the presence and absence of a subset of them). Each block has an associated *profile* that keeps the mapping between variables and models. Consider, for example, a block of code in a web application whose corresponding state can be described with two variables, `username` and `password`. Suppose that one wants to associate a certain number of different models with each of these variables in order to capture various properties, such as length and character distribution, that their values can take under normal execution. In this case there will be a profile, associated with the block, that contains a mapping between each variable and the corresponding models. Whenever an event is generated for that block, the profile is used to find the models to evaluate the features of the state variables.

In our implementation, the task of a model is to assign a probability value to a feature of a state variable or a set of state variables associated with the block that is about to be executed. This value reflects the probability of the occurrence of a given feature value with regards to an established model of "normality." The assumption is that feature values with a sufficiently low probability indicate a potential attack. The overall anomaly score of a block is derived from the probability values returned by the models that are associated with the block and its variables. The anomaly score value is calculated through the weighted sum shown in Equation 1. In this equation, w_m represents the weight associated with model m, while p_m is the probability value returned by model m.

$$AnomalyScore = \sum_{m \in Models} w_m * p_m \tag{1}$$

A model can operate in one of two modes, training or detection. The training phase is required to determine the characteristics of normal events (that is, the profile of a block according to specific models) and to establish anomaly score thresholds to distinguish between regular and anomalous values of the state variables. This phase is divided into two steps. During the first step, the system creates a profile and trains the associated models for each block in the applications. During the second step, suitable thresholds are established. This is done by evaluating the states associated with the blocks using the profiles created during the previous step. For each block, the most anomalous score (i.e., the lowest probability) is stored in the block's profile and the threshold is set to a value that is a certain adjustable percentage lower than this minimum. The default setting for this percentage is 10%. By modifying this value, the user can adjust the sensitivity of the system and perform a trade-off between the number of false positives and the expected detection accuracy.

Once the profiles have been created—that is, the models have learned the characteristics of normal events and suitable thresholds have been derived—the system switches to detection mode. In this mode, anomaly scores are calculated and anomalous states are reported.

`libAnomaly` provides a number of built-in models that can be combined to model different features. By default, block profiles are configured to use all the available models with equal weights. However, to improve performance, if some application-specific knowledge is available, the user can configure profiles to only use a subset of the models, or fine-tune the way they are combined.

In Swaddler, we used a number of existing `libAnomaly` models to represent the normal values of single variables and we developed two additional models to capture relationships among multiple variables associated with a block. We describe the models we used in the next two sections.

4.3 Univariate Models

In the context of this paper, we will use the term *univariate models* to refer to the anomaly models that are used to capture various properties of single variables associated with a block. `libAnomaly` already contains a number of univariate models. These models can be used to characterize the normal length of a variable (*Attribute Length* model), the structure of its values (*Attribute Character Distribution* model), the set of all the possible values (*Token Finder* model), etc. In the following, we provide a brief description of some of the univariate models used by the current Swaddler implementation. A more in-depth description of these and other available models can be found in [17, 10].

4.3.1 Token Finder.

In our implementation, the purpose of the Token Finder model is to determine whether the values of a certain variable are drawn from a limited set of possible alternatives (i.e., they are elements of an enumeration). In web applications, certain variables often take one of few possible values. For example, in our shopping cart application, the variable `_SESSION['shipcost']` can be set to one of three predefined values depending on which shipping method is chosen by the user. If a malicious user attempts to set her shipping cost to a value that is not part of the enumeration, the attack is detected. When no enumeration can be identified, it is assumed that the attribute values are random.

The classification of an argument as an enumeration or as a random value is based on the observation that the number of different occurrences of variable values is bound by some unknown threshold t in the case of an enumeration while it is unrestricted in the case of random values. During the training phase, when the number of different values for a given variable grows proportionally to the total number of its samples, the variable is characterized as random. If such an increase cannot be observed, the variable is modeled with an enumeration.

Once it has been determined that the values of a variable are tokens drawn from an enumeration, any value seen during the detection phase is expected to appear in the set of known values. When this happens, 1 is returned by the model (indicating normality), and 0 is returned otherwise (indicating an anomalous condition). If it has been determined that the variable values are random, the model always returns 1.

4.3.2 Attribute Length.

The length of a variable value can be used to detect anomalous states, for example when typical values are either fixed-size tokens (as it is common for session identifiers) or short strings derived from human input. In these cases, the length of the parameter values does not vary significantly between executions of the same block. The situation may look different when malicious input is passed to the program. For example, XSS attacks that attempt to inject scripts in pages whose content is generated dynamically, often require to send an amount of data that can significantly exceed the length of legitimate parameters.

Thus, the goal of this model is to approximate the actual but unknown distribution of the length of values of a variable and detect instances that significantly deviate from the observed normal behavior. During the training phase, the value length distribution is approximated through the sample mean and variance. Then, during the detection phase, the abnormality of a given value for a variable is assessed by the “distance” of the given length from the mean value of the length distribution. The calculation of this distance is based on the Chebyshev inequality [18].

4.3.3 Attribute Character Distribution.

The attribute character distribution model captures the concept of a “normal” or “regular” value of a variable by looking at its character distribution. The approach is based on the observation that values of variables in web applications are mostly human-readable and mostly are drawn from a small subset of the ASCII characters. In case of attacks that send binary data or repetitions of a single character, a completely different character distribution can be observed.

During the training phase, the idealized character distribution of a variable values (i.e., the distribution that is perfectly normal) is approximated based on the sorted relative character frequencies that were observed. During the detection phase, the probability that the character distribution of a string parameter fits the normal distribution established during the training phase is calculated using a statistical test (Pearson χ^2-test).

4.4 Multivariate Models

In the context of this paper, we will use the term *multivariate models* to refer to anomaly models that are used to capture relationships among multiple variables associated with a block. In particular, Swaddler adds two multivariate models to the `libAnomaly` framework: a *Variable Presence or Absence model*[1] and a *Likely Invariants* model.

4.4.1 Variable Presence or Absence.

The purpose of the Variable Presence or Absence model is to identify which variables are expected to be always present when accessing a basic block in an application. For example, in our sample store application, the variables `_SESSION['loggedin']` and `_SESSION['username']` have to be always present when accessing one of the administrative pages. When a malicious user tries to directly access one of the protected pages, these variables will not be present and the attack will be detected.

During the training phase, the model keeps track of which variables are always set when accessing a particular block of code. Based on this information, each state variable associated with the block is given a weight, where variables that were always present are given a weight of 1 and variables that were sometimes absent are given a weight in the range from 0 to 1, depending on the number of times that the variable has been seen. The total score for a block is calculated as the sum of all variables scores divided by the number of variables in the block. This score is always between 0 and 1.

During the detection phase, the total score of the block is calculated based on the established weights. Therefore, the absence of a variable with a higher weight results in a lower score for the state associated with the block.

4.4.2 Likely Invariants.

A program invariant is a property that holds for every execution of the program. If the property is not guaranteed to be always true in all the possible executions, it is called a *likely invariant*. To be able to automatically detect and extract state-related likely invariants, we integrated the Daikon engine [19, 20] in the `libAnomaly` framework.

Daikon is a system for the dynamic detection of likely invariants, which was originally designed to infer invariants by observing the variable values computed over a certain number of program executions. Daikon is able to generate invariants that predicate both on a single variable value (e.g., $x == 5$) and on complex compositions of multiple variables (e.g., $x > abs(y)$, $y = 5 * x - 2$). Its ability to extract invariants predicating on multiple variables is one of the main reasons for including Daikon in our tool.

In training mode, Daikon observes the variable values at particular program points decided by the user. In order to integrate it in our system, we developed a new component that translates the events generated by our sensor into the Daikon trace format. Our component is also able to infer the correct data type of each variable, by analyzing the values that the variables assume at runtime. The output of this type inference process is required for the correct working of the Daikon system.

At the end of the training phase, the Daikon-based model calculates the set of likely invariants and computes the probability that each of them appears in a random data set. If this is lower than a certain threshold (i.e., if it is unlikely that the invariant has been generated by chance) the invariant is kept, otherwise it is discarded.

For example, in our store application, Daikon detects that the block of code that charges the user's credit card is associated with the following invariants on the state variables:

```
loggedin == 'yes'
total > price
```

[1] Even though based on similar idea, this model is different from the *Attribute Presence or Absence* model described in [10].

This means that, when the basic block is executed, the `loggedin` variable is always set to *yes* (because the user must be logged in order to be able to buy items) and the `total` value charged to the user is always greater than the price of the purchased items (because of the taxes and shipping costs).

When the system switches to detection, all the invariants that apply to the same block are grouped together. The algorithm then automatically generates the C++ code of a function that receives as a parameter an event created by the sensor. The function performs three actions:

- it fetches the value of the variables predicated by the invariants (in our example, `loggedin`, `total`, and `price`);
- it verifies that the runtime type of each variable is correct (in our example, we expect `total` and `price` to be integers and `loggedin` to be a string);
- finally, it evaluates the invariants, which, in our example, are represented by the following snippet of C++ code:

```
if (strcmp(loggedin, "yes")!=0) return 0;
if (total <= price) return 0;
return 1;
```

The result of the function is a value (0 or 1) that represents whether the application state associated with the PHP block violates the likely invariants inferred during the training phase. This value is then combined with the ones provided by the other `libAnomaly` models to decide if the state is anomalous or normal as a whole.

5 Evaluation

We evaluated our system on several real-world, publicly available PHP applications, which are summarized in Table 1. BloggIt is a blog application that allows users to manage a web log, publish new messages, and comment on other people's entries. PunBB is a discussion board system that supports the building of community forums. Scarf is a conference management application that supports the creation of different sessions, the submission of papers, and the creation of comments about a submitted paper. SimpleCms is a web application that allows a web site maintainer to write, organize, and publish online content for multiple users. WebCalendar is an online calendar and event management system. These applications are a representative sample of the different type of functionality and levels of complexity that can be found in commonly-used PHP applications.

The evaluation consisted of a number of tests in a live setting with each of the test applications. All the experiments were conducted on a 3.6GHz Pentium 4 with 2 GB of RAM running Linux 2.6.18. The server was running the Apache web server (version 2.2.4) and PHP version 5.2.1. Apache was configured to serve requests using threads through its `worker` module.

Attack-free data was generated by manually operating each web application and, at the same time, by running scripts simulating user activity. These scripts controlled a browser component (the KHTML component of the KDE library [21]) in order to exercise the test applications by systematically exploring their workflow.

In particular, for each application, we identified the set of available user profiles (e.g., administrator, guest user, and registered user) and their corresponding atomic operations (e.g., login, post a new message, and publish a new article), and then we combined these operations to model a typical user's behavior. For

Application Name	PHP Files	Description	Known Vulnerabilities
BloggIt 1.01	24	Blog engine	CVE-2006-7014
PunBB 1.2.4	67	Discussion board system	BID 20786
Scarf 2006-09-20	18	Conference management system	CVE-2006-5909
SimpleCms	22	Content management system	BID 19386
WebCalendar 1.0.3	123	Calendar application	BID 23054

Table 1: Applications used in the experiments. For each application, we report the number of files that compose the application as an indication of its size, and the known attacks against it, if any. Vulnerabilities are referenced by their Common Vulnerabilities and Exposures ID (CVE) or their Bugtraq ID (BID).

example, a common behavior of a blog application's administrator consists of visiting the home page of the blog, reading the comments added to recent posts, logging in, and, finally, publishing a new entry. The sequences of requests corresponding to each behavior were then replayed with a certain probability reflecting how often one expects to observe that behavior in the average application traffic.

In addition, we developed a number of libraries to increase the realism of the test traffic. In particular, one library was used to create random user identities to be used in registration forms. In this case, we leveraged a database of real names, addresses, zip codes, and cities. Another library was used to systematically explore a web site from an initial page in accordance with a selected user profile's behavior. For example, when simulating a blog's guest user, the library extracts from the current page the links to available blog posts, randomly chooses one, follows it, and, with a certain probability, leaves a new comment on the post's page by submitting the corresponding form.

We used this technique to generate three different datasets: the first was used for training the `libAnomaly` models, the second for choosing suitable thresholds, and the third one was the clean dataset used to estimate the false positive rate of our system.

Since our tests involved applications with known vulnerabilities (and known exploits), it was not sensible to collect real-world attack data by making our testbed publicly accessible. Therefore, attack data was generated by manually performing known or novel attacks against each application, while clean background traffic was directed to the application by using the user simulation scripts. We used the datasets produced this way to assess the detection capability of our system.

5.1 Detection Effectiveness

We evaluated the effectiveness of our approach by training our system on each of the test applications. For these experiments, we did not perform any fine-tuning of the models, equal weights were assigned to each model, and we used the default 10% threshold adjustment value. Then, we recorded the number of false positives generated when testing the application with attack-free data and the number of attacks correctly detected when testing the application with malicious traffic.

Table 2 summarizes the results of our experiments. The size of the training and clean sets is expressed as the number of requests contained in each dataset. Coverage represents the number of lines in each application that have been executed at least once during training. Note that in all cases the coverage was less than 100%. Unexplored paths usually correspond to code associated with the handling of error conditions or with alternative configuration settings, e.g., alternative database libraries or layout themes. The false positives column reports the total number of legitimate requests contained in the clean set that Swaddler incorrectly flagged as anomalous during the detection phase. The attack set size illustrates the number of

Application	Training Set Size (# requests)	Coverage (%)	Clean Set Size (# requests)	False Positives	Attack Set Size (# requests)	Attacks Detected
BloggIt	9779	91	1586	0	15	15
PunBB	10200	67	1360	5	1	1
Scarf	9615	86	1000	1	10	10
SimpleCms	9333	95	1969	0	10	10
WebCalendar	19800	66	3300	1	1	1

Table 2: Detection effectiveness.

different malicious requests contained in the attack dataset of each application. This reflects the number of different attacks we used to exploit the vulnerabilities present in the application. For example, an authentication bypass vulnerability can be exploited to get access to several restricted pages. In this case, the attack set contained requests to gain access to each of these pages. Finally, the last column reports how many of these malicious requests were successfully identified by Swaddler.

In our experiments, all attacks were successfully detected by Swaddler. For each application, we describe the vulnerability exploited by the corresponding attacks and how our system detected the attacks.

BloggIt is vulnerable to two types of attacks. First, it contains a known authentication bypass vulnerability that allows unauthenticated users to access administrative functionality. More precisely, the application stores in the session variable `login` the value "`ok`" if the user has been successfully authenticated. Whenever a user requests a restricted page, the page's code correctly checks whether the user has logged in by inspecting the session variable, and, if not, redirects her to the login page. However, the page's code fails to stop the execution after issuing the redirection instruction to the user's browser, and continues executing the code that implements the restricted functionality. Our system easily detects this attack: for example, the Variable Presence or Absence model returns a high anomaly score if the restricted code is accessed when the session does not contain the `login` variable; similarly, the Likely Invariant and Token Finder models produce an alert if the `login` variable has a value other than "`ok`".

The second flaw in BloggIt is a novel file injection vulnerability that we discovered. The application allows users to upload files on the server. The uploaded files can then be accessed online and their content (e.g., a picture) be used in blog entries and comments. However, if the uploaded file's name terminates with the `php` extension and a user requests it, the application interprets the file's content as a PHP script and blindly executes it, thus allowing an attacker to execute arbitrary commands. Our system detects the attack since all models report high anomaly scores for the unknown blocks associated with the injected script.

PunBB is vulnerable to a known file injection attack that allows arbitrary code to be executed. In fact, PunBB utilizes a user-controlled variable to present the site in the language chosen by the user, by including appropriate language files. Unfortunately, the variable value is not sanitized and, therefore, it can be modified by an attacker to include malicious code. Also in this case, Swaddler detects the attack when the blocks corresponding to the injected code are executed.

Scarf is vulnerable to a known authentication bypass attack. One of its administrative pages does not check the user's status and allows any user to arbitrarily change site-wide configuration settings (e.g., user profiles information, web site configuration). The status of a user is stored in the application's session using three variables, namely, `privilege`, `user_id`, and `email`. The flaw can be exploited by users that do not have an account on the vulnerable web site or by registered users that lack administrative privileges.

Application	Avg. Instrumentation Overhead (msec)	Avg. Detection Overhead (msec)
BloggIt	5	8
PunBB	23	115
Scarf	3	13
SimpleCms	1	5
WebCalendar	15	75

Table 3: Detection overhead.

In the first case, during an attack the vulnerable page is accessed with an empty session, and thus all our models will report a highly anomalous score; in the second case, the session variables contain values that, for example, are not recognized by the Token Finder model and that do not satisfy the predicates learned by the Likely Invariant model.

SimpleCms is vulnerable to a known authentication bypass attack. It insecurely uses the `register_globals` mechanism in a way similar to the example application described in Section 2. An attacker can simply set the request parameter `loggedin` to 1 and have access to the administrative functionality of the application. Note that this allows the attacker to bypass the authorization check but does not modify the corresponding variable in the session. Therefore, during an attack, all our models report high anomalous scores.

Finally, WebCalendar is vulnerable to a file inclusion attack. In this case, the vulnerability cannot be exploited to execute arbitrary code, but it allows an attacker to modify the value of several state variables, and, by this, to gain unauthorized privileges. Swaddler detects the attack since several models, e.g., the Token Finder and Likely Invariant, flag as anomalous the modified variables.

In our experiments, Swaddler raised a few false positives. Our analysis indicates that, in all cases, the false alarms were caused by the execution of parts of the applications that were exercised by a limited number of requests during the training phase. For example, this is the case with pages that handle the submission of complex forms containing a large number of input parameters: during training, only a subset of all the possible combinations of the input parameters were tested, and, therefore, the models associated with the portions of the page that were least visited were not sufficiently trained.

5.2 Detection Overhead

Our system introduces runtime overhead in two points during the request-serving cycle. First, for each request, some time is spent to analyze and instrument the compiled code of the requested application's page. We refer to this overhead as "instrumentation overhead". Second, during execution, whenever a basic block is entered, the analyzer has to determine whether the current state is anomalous. We call the total time spent performing this operation "detection overhead".

A test was performed to quantify the overhead introduced by our system. For each application, we ran again the requests contained in the clean set used during the detection evaluation and we recorded the time required to perform instrumentation and detection.

Table 3 presents the results of this test. It shows the average overhead per user's request broken down in its instrumentation and detection components. A direct comparison of the average request-serving time on our modified PHP interpreter and the standard interpreter is presented in Figure 5.

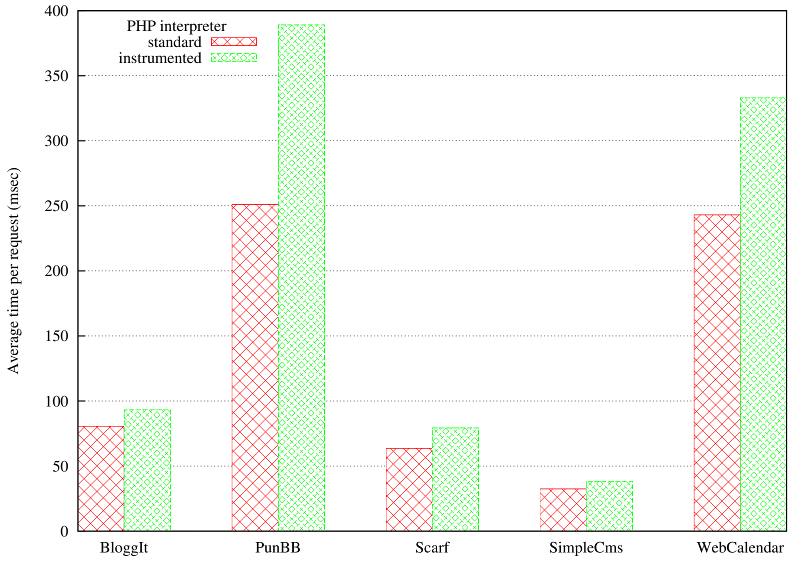

Figure 5: Total overhead for each application.

There are two main factors influencing the performance of our tool: the number of state variables that need to be analyzed for each basic block and the number of basic blocks that are traversed when serving a page. To better assess how these factors influence the performance of our system, we measured the Swaddler overhead on a set of test programs. Each program defines a certain number of variables in its session and executes a well-defined number of basic blocks. The values of the defined session variables were chosen carefully in order to avoid artificial simplifications in the trained models (e.g., the values used were not random to avoid that, in detection mode, the Token Finder model would immediately return a normal value). Furthermore, the same number of session variables was defined in all basic blocks, so that the corresponding models had to be trained in all the basic blocks of the program.

We ran the test programs on the standard PHP interpreter and on a version of the interpreter extended with our tool (after performing the training phase) and recorded the difference in the running time in the two cases. Figure 6 shows how the overhead introduced by our system changes as a function of the number of executed basic blocks and the number of examined state variables.

The overhead grows linearly as the number of executed basic blocks increases. This was expected because there is both an instrumentation and a detection overhead associated with each basic block in the program. Similarly, the overhead increases roughly linearly with the number of state variables defined. This can be explained observing that, during detection, the current value of each state variable must be extracted from the execution context and must be checked with respect to the appropriate anomaly models.

In many cases, by tuning the two performance factors (number of executed blocks and number of modeled state variables), it is possible to limit the overhead caused by our instrumentation. First, sometimes it is possible to identify state variables whose value is unlikely to be affected by an attack. For example, an application might store in a state variable the background color preferred by the current user. In this case, it is reasonable that an attack will not manifest itself with anomalous modifications to that variable. Therefore, the variable can be excluded from the subset of the monitored application state without affecting the detection capability of our tool and reducing the detection overhead.

Second, sometimes the number of basic blocks executed by an application causes the overhead to be larger than it is desired. For example, an application might execute some blocks in a loop a large number of times. In this case, it is possible to configure our sensor so that, during a request, the instrumentation

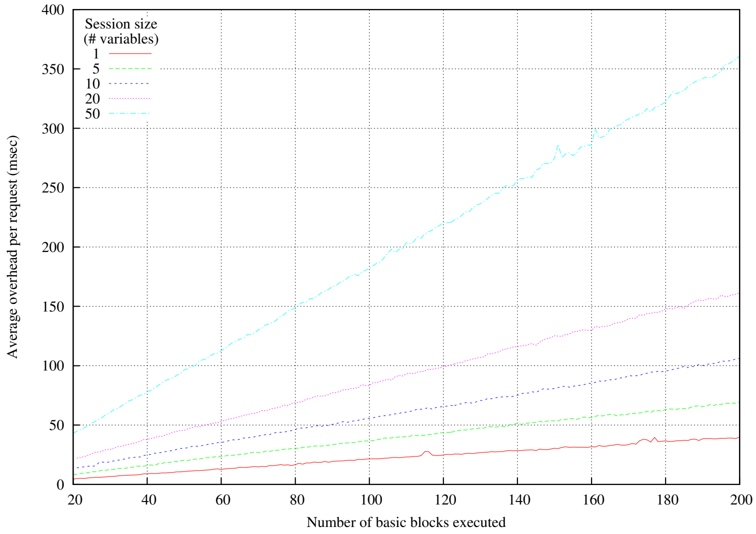

Figure 6: Factors influencing the overhead.

routine is invoked no more than a certain number of times for each block. If an attack manifests itself even if the analyzer monitors the execution of loops only up to a certain bound, this optimization will reduce the overhead without introducing false negatives.

The results of the performance tests both on real-world applications and on synthesized programs indicate that our approach introduces an acceptable overhead for most applications. These results are quite encouraging especially considering that performance was not a priority in the implementation of the current prototype of our tool.

6 Related Work

The work described in this paper is related to different previous research results in the intrusion detection field. First of all, there is a corpus of work on detecting intrusions using anomaly detection techniques (see, for example, [22, 23, 24, 25, 26, 27]), which we cannot discuss in detail here.

However, there are several proposed anomaly detection approaches that are more closely related to the solution proposed here. First of all, there is the previous work from our group on performing anomaly detection of web-based attacks by analyzing the requests and replies exchanged between clients and servers [10, 11]. This previous work introduced the idea of using statistical models to characterize the normal values of the parameters of web requests. This work suggested that this technique could be applied to other event streams and resulted in the `libAnomaly` framework, which is used (and extended) in the research presented here. The main difference between the two approaches resides in the type of attacks that can be detected. More precisely, the analysis of requests and replies does not allow for the identification of attacks that subvert the intended, normal workflow of a web application.

Another set of results that this work is related to is in the contextualization of intrusion detection, that is, the use of detection models that take into account the different phases (or states) in which an application might be when an attack is executed. The contextualization has been initially introduced in intrusion detection systems that analyze sequences of system calls as a countermeasure against mimicry attacks [14]. For example, in [28, 29] the detection of anomalous system call sequences is contextualized using the program counter value at the moment of the system call invocation. Extensions to this approach leveraged the call stack information to characterize different execution states [30, 31, 32, 33].

The combination of contextualization techniques with the detection of anomalous system calls based on the analysis of their parameters was proposed in [34]. Even though this approach is also based on `libAnomaly` [35, 36, 17], our approach is different from the one proposed in [34] because it operates on the variables that represent the overall state of an application and not on the values used in its interaction with the underlying operating system. In addition, we introduce the concept of likely invariants as a way to characterize anomalous states, which was not considered in these previous works.

7 Conclusions

Web applications have become a common way to access information and services. These applications are vulnerable to a number of attacks that cannot always be detected by observing the application from the outside.

This paper presented Swaddler, an approach to the detection of attacks against web applications, based on the analysis of the internal application state. The approach is the first that models the values of session variables in association with critical execution points in a web application. In addition, we introduced a novel detection model that relies on multi-variable invariants to detect web-based attacks.

We developed a prototype of our system for the PHP language and we evaluated it against several real-world applications. The results show that by leveraging the internal, hidden state of a web application it is possible to detect attacks that violate its intended workflow, confirming our hypothesis that any insecure state usually corresponds to an anomalous state.

Future work will focus on two directions. First, we will extend our approach to consider other parts of the internal state of an application. Second, we will focus on optimizations that will reduce the overhead introduced by the instrumentation of the PHP interpreter.

8 Acknowledgments

This research was partially supported by the National Science Foundation, under grants CCR-0238492 and CCR-0524853.

References

[1] Almgren, M., Debar, H., Dacier, M.: A Lightweight Tool for Detecting Web Server Attacks. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA (February 2000)

[2] Vigna, G., Robertson, W., Kher, V., Kemmerer, R.: A Stateful Intrusion Detection System for World-Wide Web Servers. In: Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV (December 2003) 34–43

[3] Lee, S.Y., Low, W.L., Wong, P.Y.: Learning Fingerprints for a Database Intrusion Detection System. In: 7th European Symposium on Research in Computer Security (ESORICS). (2002)

[4] Valeur, F., Mutz, D., Vigna, G.: A Learning-Based Approach to the Detection of SQL Attacks. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'05). (July 2005)

[5] Halfond, W., Orso, A.: AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In: Proceedings of the International Conference on Automated Software Engineering (ASE'05). (November 2005) 174–183

[6] Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In: Proceedings of the IEEE Symposium on Security and Privacy. (May 2006)

[7] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Evans, D.: Automatically Hardening Web Applications Using Precise Tainting. In: Proceedings of the 20th International Information Security Conference (SEC'05). (May 2005) 372–382

[8] Su, Z., Wassermann, G.: The Essence of Command Injection Attacks in Web Applications. In: Proceedings of the 33rd Annual Symposium on Principles of Programming Languages (POPL'06). (2006) 372–382

[9] Xie, Y., Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In: Proceedings of the 15th USENIX Security Symposium (USENIX'06). (August 2006)

[10] Kruegel, C., Vigna, G.: Anomaly Detection of Web-based Attacks. In: Proceedings of the 10^{th} ACM Conference on Computer and Communication Security (CCS '03), Washington, DC, ACM Press (October 2003) 251–261

[11] Kruegel, C., Vigna, G., Robertson, W.: A Multi-model Approach to the Detection of Web-based Attacks. Computer Networks **48**(5) (August 2005) 717–738

[12] PHP: Session Support in PHP. `http://php.net/manual/en/ref.session.php/`

[13] Aho, A.V., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc. (1986)

[14] Wagner, D., Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems. In: Proceedings of the 9^{th} ACM Conference on Computer and Communications Security, Washington DC, USA (November 2002) 255–264

[15] Zend: Zend Engine. `http://www.zend.com/products/zend_engine`

[16] The Computer Security Group at UCSB: libAnomaly Project Homepage. `http://www.cs.ucsb.edu/~seclab/projects/libanomaly`

[17] Mutz, D., Valeur, F., Kruegel, C., Vigna, G.: Anomalous System Call Detection. ACM Transactions on Information and System Security **9**(1) (February 2006) 61–93

[18] Billingsley, P.: Probability and Measure. 3 edn. Wiley-Interscience (April 1995)

[19] Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2) (February 2001) 99–123 A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[20] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming (2007)

[21] KDE Project: KDE HTML widget. `http://api.kde.org/3.5-api/kdelibs-apidocs/khtml/html/`

[22] Denning, D.: An Intrusion Detection Model. IEEE Transactions on Software Engineering **13**(2) (February 1987) 222–232

[23] Ko, C., Ruschitzka, M., Levitt, K.: Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In: Proceedings of the 1997 IEEE Symposium on Security and Privacy, Oakland, CA (May 1997) 175–187

[24] Ghosh, A., Wanken, J., Charron, F.: Detecting Anomalous and Unknown Intrusions Against Programs. In: Proceedings of the Annual Computer Security Application Conference (ACSAC'98), Scottsdale, AZ (December 1998) 259–267

[25] Lee, W., Stolfo, S., Mok, K.: Mining in a Data-flow Environment: Experience in Network Intrusion Detection. In: Proceedings of the 5^{th} ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '99), San Diego, CA (August 1999)

[26] Javitz, H.S., Valdes, A.: The SRI IDES Statistical Anomaly Detector. In: Proceedings of the IEEE Symposium on Security and Privacy. (May 1991)

[27] Forrest, S.: A Sense of Self for UNIX Processes. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (May 1996) 120–128

[28] Sekar, R., Bendre, M., Bollineni, P., Dhurjati, D.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (May 2001)

[29] Sekar, R., Venkatakrishnan, V., Basu, S., S, B., DuVarney, D.: Model-carrying code: A practical approach for safe execution of untrusted applications. In: Proceedings of the ACM Symposium on Operating Systems Principles. (2003)

[30] Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: Proceedings of the IEEE Symposium on Security and Privacy. (May 2003)

[31] Gao, D., Reiter, M., Song, D.: Gray-Box Extraction of Execution Graphs for Anomaly Detection. In: Proceedings of the 11th ACM Conference on Computer and Communication Security (CCS), Washington, DC, USA (October 2004) 318–329

[32] Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., Miller, B.: Formalizing Sensitivity in Static Analysis for Intrusion Detection. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (May 2004)

[33] Giffin, J., Jha, S., Miller, B.: Efficient Context-Sensitive Intrusion Detection. In: Proceedings of 11th Network an Distributed System Security Symposium, San Diego, California (February 2004)

[34] Mutz, D.: Context-sensitive Multi-model Anomaly Detection. PhD thesis, UCSB (June 2006)

[35] Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the Detection of Anomalous System Call Arguments. In: Proceedings of the 8^{th} European Symposium on Research in Computer Security (ESORICS '03). LNCS, Gjovik, Norway, Springer-Verlag (October 2003) 326–343

[36] Kruegel, C., Mutz, D., Robertson, W., Valeur, F.: Bayesian Event Classification for Intrusion Detection. In: Proceedings of ACSAC 2003, Las Vegas, NV (December 2003)