Generic Approach for Security Error Detection Based on Learned System Behavior Models for Automated Security Tests

Christian Schanes, Andreas Hübler, Florian Fankhauser, Thomas Grechenig
Vienna University of Technology
Industrial Software (INSO)
1040 Vienna, Austria
E-mail: christian.schanes,andreas.huebler,
florian.fankhauser,thomas.grechenig@inso.tuwien.ac.at

***Abstract*—The increasing complexity of software and IT systems creates the necessity for research on technologies addressing current key security challenges. To meet security requirements in IT infrastructures, a security engineering process has to be established. One crucial factor contributing to a higher level of security is the reliable detection of security vulnerabilities during security tests. In the presented approach, we observe the behavior of the system under test and introduce machine learning methods based on derived behavior metrics. This is a generic method for different test targets which improves the accuracy of the security test result of an automated security testing approach. Reliable automated determination of security failures in security test results increases the security quality of the tested software and avoids costly manual validation.**

***Keywords*—Machine learning; Unsupervised learning; System testing; Robustness; Security**

I. Introduction

Security tests are an important aspect for improving robustness and security of systems within a security engineering process. To increase security testing efficiency and effectiveness, automation of such tests is required. A main aspect in automated security testing is the accurate automated determination of security failures.

Security failures are defined by Thompson [1] as non specified side effects of the software, which are unknown, hidden functions in the system. To minimize the risk of security vulnerabilities, security tests aim to uncover these hidden functions by the usage of unspecified inputs for the system. Successfully detecting and correctly determining such side effects is the main goal in a security testing process.

A good indication of the quality of an automated security test approach is the number of wrongly detected errors. These fall into the category of either false-negative or false-positive security test results. False-negative results are defined as errors which are not detected by the security test. In contrast, false-positive results arise when the testing approach reports an error which is, in fact, no error. The aim of an automated testing approach is to increase the efficiency of error detection and to, thereby, reduce the amount of resources required. A high number of false-positives renders an automated testing approach non applicable, as all detected errors would have to be analyzed manually. A possible solution to this problem is to have the security testing program learn the system behavior and use a suitable decision algorithm to limit the number of false-positives to a minimum.

In this work we present the automated generation of a system behavior model. Our security testing approach analyzes the behavior of the System Under Test (SUT) by observing its behavior during functional test cases. With repeatedly executing functional test cases and extracting behavior metrics of the SUT the security testing approach can automatically generate the behavior model of the system. Based on these training cycles, the security testing approach learns the "normal" behavior of the SUT. It then executes security test cases and monitors the response metrics. By analyzing the divergence to the normal system behavior it is able to expose unexpected behavior, and, thereby, possible security vulnerabilities. The behavior of a SUT can be monitored via various interfaces to further improve the detection of errors.

Many security test approaches base the determination of a security error solely on the detection of application crashes. Application crashes are one important aspect; however, detecting security issues is more complex. Software failures can have numerous effects that are not as obvious but can, nevertheless, be dangerous, e.g., high CPU load, returning incorrect output, allowing privileged access, etc. With intensive monitoring of a number of different behavior metrics it is possible to detect various security errors, leading to a thorough security test result.

The remainder of this paper is structured as follows: Section II lists related work. Section III introduces the definition of the used system behavior model and gives details about the used learning approach to automatically build the system behavior model. The architecture of an implemented prototype is presented in Section IV. Section V discusses results of the approach applied to a benchmark for security testing tools and an open source software as SUT. The paper finishes with a conclusion and ideas for future work in Section VI.

II. Related Work

Miller *et al.* [2], [3] introduced fuzzing as a black-box vulnerability detection method by fault injection. Nowadays, fuzzing has become a widely used method to test software robustness and security of different applications [4], [5]. During

the fuzzing process random values and simulated attacks are used as program inputs to trigger unexpected behavior of the SUT [6].

Reliable error detection in automated security tests is an important aspect of a thorough test process. Various possible behavior aspects can be monitored for such security tests to automatically detect security errors. Monitoring the Graphical User Interface (GUI) is shown by Tuglular *et al.* [7]. Antunes and Vieira [8] presented an approach for monitoring the SUT during penetration tests to detect injection vulnerabilities. The latter also showed increased error detection in the software with additional monitoring compared to tools which are only monitoring the response of the software.

Song *et al.* [9] presented an approach for monitoring and analyzing network traffic. Another solution is using proxy approaches to intercept network connections, e.g., for database connections to detect SQL Injection attacks [10], [8]. This could also be applied for other injection attacks, e.g., Uniform Resource Locator (URL) injection. Another important metric is the response behavior of the SUT. Martin [11] monitors the response of web services and Taber *et al.* [12] monitor the response of Voice over IP (VoIP) applications during security test execution.

Wang *et al.* [13] worked on a list of key network attributes for intrusion detection. The results can also be applied as attributes for error detection during security test execution. Network monitoring is also a possible solution for security testing SUTs with many connected components. However, one problem of network monitoring is the use of encryption.

Steinbacher *et al.* [14] used system load metrics for analyzing Quality of Service (QoS) aspects of a SUT during Distributed Denial of Service (DDoS) attacks. The difference to the approach proposed in this work is that during performance- and load-tests many requests will be sent and only the load state during the whole test execution (based on many single requests) of the system is relevant. Additionally, the measured metrics are often not automatically processed to get the final test result. In automated security tests single requests are measured and automated analysis is used for error determination.

The shown references often focus on solutions for specific protocols or attacks, e.g., focus on injection vulnerabilities of web services [8], for network traffic [9] or monitored for application crashes [2]. Different applications have different behavior and output based on their functionality. For example, a Session Initiation Protocol (SIP) server behaves differently than a HTTP server. A generic approach to detect security errors for different applications requires support for monitoring and a combination of different captured behavior aspects of the SUT. The approach introduced in this paper supports the combination of several monitors in one application. This allows a generic analysis of the test execution independent of the specific protocol or SUT.

Modeling and learning behavior of components for various use cases in IT security are available in the literature. Kiziloren and Germen [15] introduce a network monitoring approach using Self Organizing Maps (SOM) to detect and classify network based behavior. Intrusion Detection Systems (IDS) are used to detect attacks on systems and log/alert the incidents. In contrast, Intrusion Prevention Systems (IPS) are used to prevent attacks in addition to detecting them. Such systems often use artificial intelligence and machine learning concepts to learn the normal system behavior and to detect abnormal behavior during runtime. E.g., Jalil *et al.* [16] compare various algorithms for the usage with IDS, Bao *et al.* [17] use a Support Vector Machine (SVM) algorithm and Gosh and Schwartzbard [18] use neural networks. Linda *et al.* [19] show monitoring of network timing behavior which is used within an IDS to detect suspicious network traffic. Some of these identification approaches used in other security research areas can also be applied to automated error detection during security test execution.

Behavior based classification is also used in the malware analysis area. Often, suspicious software is further analyzed if it shows a similar behavior to already known malware. Machine learning and classification approaches are used for analysis, e.g., Firdausi *et al.* [20] analyzed various classification algorithms for the use in malware analysis, Rieck *et al.* [21] presented an approach based on SVM and Gavrilut *et al.* [22] used the perceptron algorithm for classification.

Firdausi *et al.* [20] and Rieck *et al.* [21] provided a behavior based approach by collecting syscalls and building call graphs of the application. This would be a possible way for security tests if sufficient access to the SUT is available, e.g., during component tests. However, for system tests, often only the interfaces of the SUT are accessible and no access to the actual operating environment for tracing the syscalls is possible.

III. Definition of a System Behavior Model for Security Tests

The system behavior model in our approach is based on a set of behavior metrics of the SUT. By using learning cases based on functional test cases the "normal" behavior of the SUT is captured during the execution of the learning cases. In our approach, the set of all captured and extracted values for the metrics build the system behavior model. It can be used during the security test execution as a reference for determining unexpected behavior, and, therefore, security failures.

A. Concept for Building the System Behavior Model

Security failures are defined by Thompson [1], as unknown and unexpected behavior of the SUT. Figure 1 shows the difference between the specified and implemented software. For functional tests the specified expected behavior of the system is known and can be analyzed more easily. Using positive/negative functional test cases and comparison to the specified functionality, functional errors can be identified. Such errors are not or incorrectly implemented features.

Security tests have to discover side effects in the system that are not specified. To detect side effects and unexpected behavior the expected, normal behavior has to be known. Functional tests should cover the defined requirements of the software and, therefore, the expected behavior of the system. In the presented approach the normal behavior is determined by using both positive and negative functional test cases as learning samples.

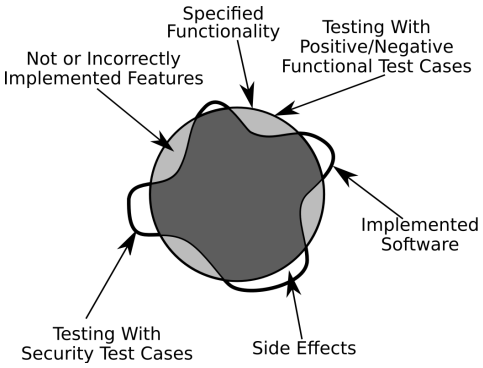


Fig. 1. Definition of the normal behavior and side effects of software systems based on Thompson [1]

The proposed model is based on a set of individually measured behavior metrics of the system and the relationship between them. The first step is to install sensors around the SUT to monitor the behavior of the system, using different available interfaces of the SUT, and derive measurable metrics. The kind of sensors used, often depends on the current test stage of the software development life cycle. During component tests most of the time direct access to the SUT is available which allows white-box monitoring approaches. This means, for example, the possibility to monitor the CPU load, memory usage of the application and the system, coverage of the application, syscall graphs, etc. During system tests, often no access to the host is available and, therefore, other methods to monitor the SUT are required. In such cases it may be possible to analyze the network traffic, timing behavior, message sizes, etc.

B. Representation of the System Behavior Model

For the representation of the system behavior model it is necessary to combine the individually derived behavior metrics to one model which can be used during security test execution for the determination of unexpected behavior. The form used for captured behavior depends on the interface. Regardless of the captured behavior of the interface, as a result a numeric representation for the metric is required for our approach. This is used for further processing and deriving one final result. For example, analyzing a log file for abnormal behavior requires a deduction from the textual representation of the observed behavior to a measurable metric, e.g., counting the occurrence of various critical keywords. Similar processing is required for system load behavior, e.g., CPU load, which represents the load during the test execution. Representative metrics have to be extracted from the observation of the CPU load, for example, min/max/average of the load during execution of the security test case.

In the presented approach the system behavior model is a set M of behavior metric vectors x_i:

$$M := \{x_i | x_i \in \mathbb{R}^n, i = 1, 2, \dots l\} \tag{1}$$

where n is the number of behavior metrics derived from the captured system behavior for the specific system and l is the number of learning cases used to build the system behavior model.

During the security test execution, the single metrics are collected after each executed security test case and a measurable form is derived. The measurable form is combined as a vector $t_j \in \mathbb{R}^n$ where j is the index of the security test case. The vector t_j represents the specific system behavior under security test case j.

C. Learning and Test Determination Approach

Machine learning in general can be separated into two major learning settings. In supervised learning, a training set is available and, therefore, the result of executing the system with the given parameters is known. In unsupervised learning, only the input parameters are available but the result is not known and has to be learned [23].

Since the exact outcome of test cases to classify the training set is not defined we use unsupervised learning. For the approach presented in this paper we used a SOM, which is one unsupervised learning method. In principle, a SOM is an artificial neural network that maps a possible high-dimensional input, in our approach the behavior metrics, to a two dimensional array of neurons. The system behavior model and the extraction of behavior metrics are generic and can also be used with other machine learning methods. As a possible future work, machine learning algorithms have to be evaluated to discern the most suitable algorithm for the presented problem.

The SUT is defined as a function f as $b_i = f(p_i)$. The result of the function f is the captured behavior b_i of the system for the learning case p_i. Further, a function $x_i = g(b_i)$ is required which converts the captured behavior b_i to a measurable form as vector x_i for the learning case p_i.

The SOM will be trained by capturing the single metrics of the SUT behavior, triggered by using a set L of learning cases p_i as input without attacking the SUT which is defined as:

$$L := \{p_i | i = 1, 2, \dots l\} \tag{2}$$

Testing the software f is defined as $c_j = f(q_j)$ where q_j is the input test data for the SUT. c_j is the specific behavior of the SUT for the test data q_j and the measurable form t_j is determined by $t_j = g(c_j)$. Determining the test result by deciding if the result is a possible security failure, is done by mapping the metrics vector t_j of the test case to the trained SOM.

The SOM is based on a set of m neurons which are represented as weight vectors w_k. It is defined as:

$$W := \{w_k | w_k \in \mathbb{R}^n, k = 1, 2, \dots m\} \tag{3}$$

The dimension n of the weight vector is given by the number of metrics which are used as input vector to the SOM. A distance function is required to calculate the matching neuron which is given by $d(x, y)$. For the presented proof of concept we used the Euclidean distance metric.

The learning algorithm is based on the following steps adapted from [24]:

Step 1: Initially, all neurons are assigned random values, preferably from the domain of the input samples.

Step 2: Following is a search for the winning neuron w_k by finding the position of the neuron with the shortest distance to behavior metric x_i used for training. When the input vector is applied to the SOM, the competitive learning rule dictates that only one neuron can be active, winning the competition. The winning neuron is also called the image of the input behavior metric vector x_i on the SOM array and is defined as the neuron whose weight vector lies closest to x_i, determined by:

$$k = \underset{k}{\operatorname{argmin}}\{d(x_i, w_k)\},\ k = 1, 2, \ldots m \qquad (4)$$

Step 3: The winning neuron and its geometric neighborhoods are stimulated by the input behavior metric vector x_i. By adapting the synaptic weight vectors of the neighborhoods the distance to the input behavior metric vector will be reduced. This continued shifting of positions of the neurons by adapting the weight vectors leads to a global ordering.

During one learning iteration Step 2 and Step 3 are performed for each training vector. For the learning process of a SOM multiple iterations are used.

For the security test case the behavior test vector t_j is used to find the winning neuron w_k as shown in Step 2. This neuron is also called Best Matching Unit (BMU).

As result of a security test case two distinct outcomes are possible: security failure and no security failure. In our approach all learning cases per definition do not trigger a security failure of the system. They are based on the functional specification and are covered in a functional test phase during the software development life cycle. We used an algorithm to cluster the behavior metrics within the SOM to define similar results. If learned behavior is located within the cluster it is defined as no security error, otherwise as security error. As algorithm in the proof of concept we used k-Means clustering. In future work, evaluation of different clustering methods is of interest.

IV. Architecture for Automated Error Detection

The architecture of the presented approach with the monitoring and analyzing engine is based on three main levels. The first level is the measurement of a single system behavior and a deduction to a metric as measurable form. The second is the combination of several individual behavior metrics to a behavior model. The third is the usage of the behavior model during security test execution to detect unexpected behavior. The following subsections will explain the architecture in more detail.

A. Architecture Overview

Figure 2 shows a simplified architecture of the components of the security test system and the SUT. At the bottom of the figure the SUT is depicted. The SUT contains various components which communicate with internal communication interfaces. Additionally, the SUT communicates with external components which are not part of the test target, the user interface, operating system and the underlying hardware.

The security testing framework contains the test execution controller as central component to manage the learning and security test execution. The two engines for the learning cases and security test cases prepare the messages with the test data. The learning and determination components are responsible for building of the system behavior model and the determination of security failures by using the model. For our approach we allow multiple analyzers to capture system behavior and convert the captured behavior to a measurable metric. Each analyzer contains a module for behavior capturing and a module for metric deduction based on the captured behavior.

To combine the single measured values from the analyzers, the analyze manager is used. It collects the results of all analyzers and performs the learning and clustering. The manager triggers the single analyzer before and after the execution of the security test/learning case. Before the security test/learning case the analyzers are informed, that they should start capturing the system behavior. After the security test/learning case the analyze manager collects the determined metrics t_j for the security test cases and x_i for learning cases from the single analyzers.

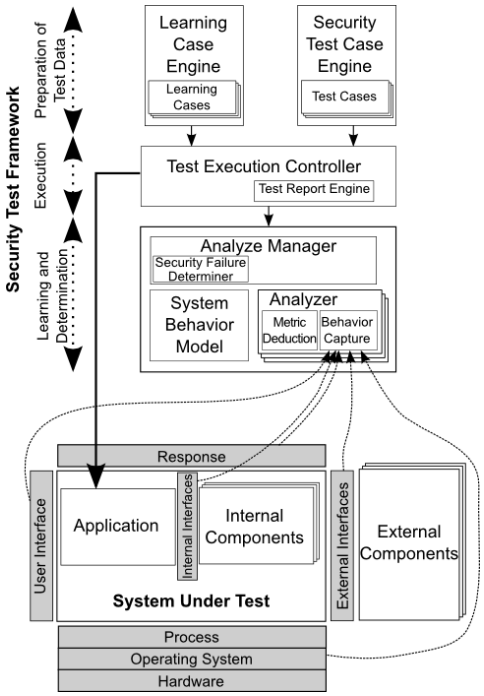


Fig. 2. Architecture for learning system behavior and determining security failures

B. Transformation of Monitoring Information to Measurable Metrics

Installing and connecting the analyzers is required before learning and security test execution. Depending on the test stage and the available access to the SUT different aspects can be monitored. If enough access rights are available during test execution, debuggers can be attached to the SUT to monitor application internals. Often, virtualization is used in system architectures. Virtualization provides additional monitoring possibilities of the encapsulated virtualized system, e.g., extending the Java Virtual Machine allows monitoring access

to system components via the Java security manager. If the SUT has a GUI, the GUI is one more example of a possible interface to monitor.

For capturing information about the system, approaches from the literature can be used. Willems *et al.* [25] present several monitoring aspects for malware analysis used in cwsandbox which can also be used for monitoring the SUT for security tests. Bossert *et al.* [26] use a monitoring approach for reverse engineering of application protocols. The presented usage of monitoring communication channels can be applied for security tests. More literature on capturing the behavior of a system is presented in Section II.

After monitoring the SUT behavior the metrics have to be converted in a measurable form. Using aspects from the representational theory of measurement the measurable metrics are derived. For the processing the five types of measurement scales have to be considered which are nominal, ordinal, interval, ratio and absolute [27].

A *nominal scale* is an unordered set of categories and the captured behavior belongs to one of the categories. An ordered nominal scale is called *ordinal scale*. This scale allows to apply comparisons such as a is greater than b. For nominal and ordinal scales no further arithmetic operations such as addition or subtraction are possible. The *interval scale* is an ordinal scale with consistent intervals between the points on the scale. Arithmetic operations addition and subtraction are possible for interval scales but not multiplication and division. Using constant intervals for an interval scale will lead to a *ratio scale*. For ratio scale all arithmetic operations are valid. This is also the case for the *absolute scale*. The absolute scale is a ratio scale but with the count of the number of occurrences.

Based on the values derived using the measurement scales further processing might be required. The reason the allowed arithmetic operations are important is because different further calculations of statistical properties are allowed for different measurement scales.

Mean, median and mode are used for central tendency statistics in data sets for a single metric. All three are allowed for interval, ratio and absolute scales. For ordinal scales only mode and median are valid whereas for nominal scale only the calculation of the mode is meaningful.

In our approach statistical methods for the measurement of central tendency and variability of the derived set of numeric values are used. This paragraph describes – based on examples – the usage of the mentioned measurement scales and statistical methods. For metrics based on the CPU usage of the SUT we use an absolute scale and a range measure of variability to detect the difference between highest and lowest CPU usage within the execution of one test case. For monitoring a crash of the system, e.g., the SUT does not answer anymore, a nominal scale with the categories system crash and no system crash is used. No further statistical methods are applied.

C. Security Test Cases: Automated Decision of Security Failures

Determination of test results in functional testing is often defined beforehand, because the expected result is known and the current result has to match the specified expected result exactly. The output of functional testing is a decision of pass/fail of the test case. For security tests, determination is often based on probabilities because the exact representation of the expected result is not known. For some software failures the detection is possible with a high probability, e.g., a crash of an application during security testing indicates a software failure.

The proposed approach uses the learned behavior model for the decision, whether a security test case is a security failure or not. For this, different metrics during security test execution are captured, converted to a measurable form and compared to the results represented in the learned model as described in Section III-C. If a similarity can be detected the current test case is interpreted as “normal behavior”. If no similarity is detected, the test case is defined as unexpected behavior and, therefore, a possible security failure.

D. Prototype Implementation

As a prototype implementation the approach was integrated into an existing security testing framework[1]. The components test execution controller and security test engine of Figure 2 are available and reused from the security testing framework to generate and send security test cases with simulated attacks to the SUT for triggering unexpected behavior of the SUT.

In the security testing framework each single analyzer decides if the current test case is a security failure or not. This often leads to false-positive results. We extended the framework with an analyze manager to combine the single analyzer results to one system behavior model based on the introduced learning approach. With the single analyzers we captured the behavior of the SUT. We implemented the metric deduction module for each analyzer to deduce behavior metrics from the captured behavior. The behavior metrics are used by the analyze manager during the learning cycle to generate the system behavior model.

For each security test case the analyze manager uses the learned system behavior model and the behavior of the SUT during execution of the security test case (deduced behavior metrics of the system behavior) to determine if the security test case leads to a security failure or not.

The learning engine was implemented using the introduced learning approach. For the generation of functional test cases different sources can be reused, e.g., available functional test cases, or specific learning cases are generated based on available interface specifications. As SOM implementation for learning and usage during security test execution of the system behavior model we integrated the Java SOMToolbox[2] into the security testing framework.

E. Observing System Under Test Behavior

Following is a list of analyzers used for the security test execution presented in Section V to observe system behavior:

Response time analyzer: Measuring the response time that was needed for the request to be handled by the SUT.

[1] http://security.inso.tuwien.ac.at/esse-projects/fuzzolution/

[2] http://www.ifs.tuwien.ac.at/dm/somtoolbox/

This means increased processing time from the simulated attack request can be detected.

Response size analyzer: This analyzer uses the size of the response which can yield some information about unexpected responses if the returned response differs from the normal behavior.

Response message analyzer: The response message analyzer uses the text within the response for further processing. It checks the content of the response for suspicious strings, e.g., error messages. To make decisions about captured text behavior, a metric has to be derived. In our prototype we used the simple approach of counting the occurrence of predefined keywords in the response. Example keywords used for the security test execution are: ”Exception”, ”bad”, ”missing”, ”fatal” or ”segmentation fault”.

Response classification analyzer: The response classification analyzer is based on methods of the field of information retrieval. It uses text indexing to build an *index* of the response data during the learning phase. In the testing phase it starts a *query* on the index for every test case by comparing the response data to the existing data. By deducing a similarity metric from this comparison, it can tell whether the current response is similar to a learned and known response or not. The advantage of this method is that the metric is deduced from the content of the response. Two responses with the same length can get assigned a different value from this analyzer.

Crash analyzer: The crash analyzer checks if a crash of the SUT occurred by trying to establish a connection to the SUT. The analyzer of the implemented prototype performs a full TCP handshake using ports of the SUT. The analyzer returns a boolean value, thus if the handshake is successful, the system is regarded as “up”, otherwise the system is considered to have “crashed”.

Keywords in log file analyzer: An analyzer can be used to inspect the log files of the SUT and determine important log entries using keywords. The analyzer uses only the entries produced during the current execution of the test case by building the delta of the log file before and after the test execution. It processes the detected log entries and based on a predefined list of weighted critical keywords returns the probability by combining the number of keywords and the associated weight.

CPU load analyzer: This analyzer monitors the CPU load of the processes in the SUT to detect heavy load during/after executing a test which could indicate a Denial of Service (DoS) attack.

Memory analyzer: This analyzer monitors the size of the allocated memory of the processes in the SUT to detect changes during/after executing a test case.

V. Discussion of Applying the Implemented Approach

As a proof of concept the proposed approach is applied and the results are discussed in this section. First, we used the approach to test a common security benchmark for web application security testing tools. This ensures a possible comparison with other security test approaches. The second test target is phpBB, a commonly used bulletin board web application with known vulnerabilities in previous versions.

This section contains illustrations (see Figure 3 and Figure 4) of the two-dimensional array of neurons of the SOM. Every square represents a neuron with its weight vector. The shaded areas describe the frequency of input behavior metric vectors, projected onto this neuron. The pie-charts on some of the squares represent the distribution of the learning and security test cases in the SOM. The numbers of learned behavior metric vectors are shown as light gray pie-charts. Dark gray pie-charts represent the number of behavior metric vectors gained by observing the system during security test execution.

A. Security Test Tool Benchmark

As a benchmark, we used wavsep[3] which was also used for a published security test tool evaluation[4]. The benchmark has several variations of SQL injection and Cross Site Scripting (XSS) vulnerabilities. Additionally, a category with web pages is available which has no vulnerabilities to measure the number of tests which generate false-positive errors.

We compared single tools used in the published evaluation with results of our approach. Several evaluated tools found all expected vulnerabilities of the presented benchmark, so did our approach. *sqlmap*[5] or *Wapiti*[6], two open source security testing tools, also found all available vulnerabilities of the benchmark. However, after a detailed analysis we found limitations of other approaches which can be overcome if the expected behavior is learned before the test execution. Figure 3(a) presents results of our approach for the first SQL injection benchmark case. The vulnerability can easily be exploited using a pattern like `' OR 'x'='x`. In the left top corner of Figure 3(a) the behavior after a successful exploit is located and this position is in the cluster of detected security errors. sqlmap and Wapiti did not detect the successful exploit using this pattern, because the expected response after a successful exploit is not known. However, both detected the error using other attack vectors. Nevertheless, if other attack vectors are filtered the vulnerability will not be detected.

In the category with no vulnerabilities of the benchmark, several false-positive detected errors are reported by our approach. However, a returned HTTP 500 code should be reported as error. Other tools like sqlmap report a warning if such a return code occurs. Both, a reported vulnerability as well as a warning have finally to be validated by a human and, therefore, cause the same manual overhead.

We extended the benchmark with a functionality which performs input validation but has a security vulnerability therein. The tools sqlmap and Wapiti did not detect the vulnerability, although according to the previously mentioned security test tool evaluation[7] both tools found all SQL injection vulnerabilities. Our presented approach did, however, find it. The representation of the SOM for this case is shown in Figure 3(b). All three test cases which injected the proper SQL statement for a successful exploit are located in the cluster (left bottom corner) with the security errors.

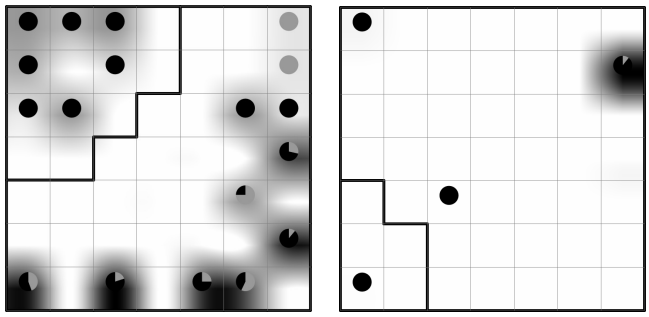
[3] http://code.google.com/p/wavsep/

[4] http://sectooladdict.blogspot.co.il/2012/07/2012-web-application-scanner-benchmark.html

[5] http://sqlmap.org/

[6] http://wapiti.sourceforge.net/

[7] See Footnote 4



(a) Applying the presented approach to the wavsep benchmark with a SQL Injection vulnerability

(b) Detection of the vulnerability of the extended benchmark

Fig. 3. System behavior model with distribution of learning and security test cases based on example execution of the prototype implementation of the approach against the described wavsep benchmark

B. Bulletin Board phpBB

Moreover, for the proof of concept, the bulletin board software *phpBB*[8] was used as SUT. Earlier versions of this tool contain known security vulnerabilities which qualify the software for automated security tests. We use it to verify the successful detection of the known vulnerabilities by the automated approach.

The known vulnerabilities of phpBB in the specific earlier versions are a SQL Injection vulnerability (see OSVDB ID: 4258[9]) and a stored XSS vulnerability (see OSVDB ID: 2532[10]). The former allows an attacker to inject arbitrary SQL code, the latter allows the storage of unverified JavaScript code.

The results of the learning and security test execution within the proof of concept test setup are visualized in Figure 4.

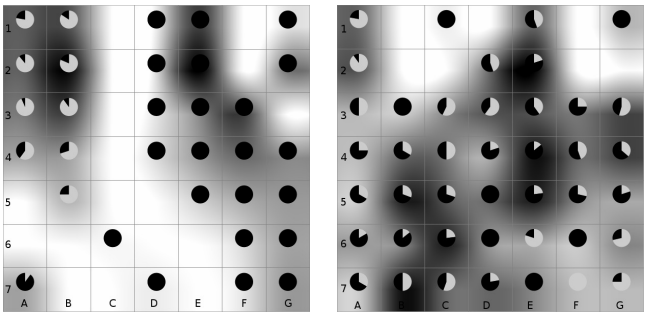
In Figure 4(a) the behavior of the system, when confronted with learning cases based on functional test data, is represented by clusters of the behavior metric vectors on the left side of the map. Even some of the security test cases yield a similar behavior as the learned behavior of the SUT. However, the majority of the security test cases lead to a behavior with a great distance to the clusters with the learned expected behavior. The vectors of behavior metrics observed using these security test cases build a cluster which is located on the right side of the map. Of interest are also the two fields (*6C* and *7D*) that contain dark pie-charts which means security failures. They could represent behavior from security test cases that are either outliers or called on an unexpected behavior on the SUT. In fact, the security test cases of both fields are successfully executed timeout-based blind injections that caused a highly increased response time.

In Figure 4(b) the system behavior model of a functionality of phpBB is presented which contains a XSS vulnerability. In field *1C* the security failure was detected, because no learning samples are assigned to this cluster. The security test cases of the assigned system behavior resulted in a successful storage of valid JavaScript code, and, therefore, a successfully detected

[8]http://www.phpbb.com/

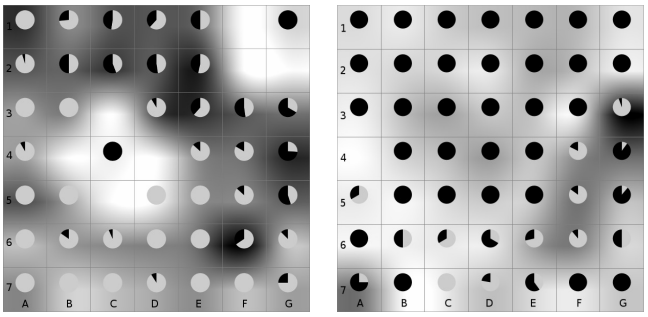
[9]http://www.osvdb.org/show/osvdb/4258

[10]http://www.osvdb.org/show/osvdb/2532



(a) Security testing phpBB SQL Injection vulnerability

(b) Security testing phpBB Cross Site Scripting vulnerability



(c) Similar to Figure 4(b) but with improved learning cycle with more learning samples

(d) Weak learning cycle for a function without security vulnerability

Fig. 4. System behavior model with distribution of learning and security test cases based on example execution of the prototype implementation of the approach against the described phpBB application

XSS vulnerability. The majority of the map is covered by the expected behavior of the SUT as can be seen by the assigned learning cases (light gray areas in the pie-charts). Some false-positive detected security failures are also located in this area. Figure 4(c) shows the same test execution but with an improved learning cycle. The false-positive detected security failures can consequently be avoided with a better learning cycle. The security failure was detected and is shown in field *4C*. The dark pie-chart at field *1G* of Figure 4(b) and in Figure 4(c) accordingly, is a false-positive result. Further investigation of the according security test cases showed that the field contains security test cases that resulted in an unexpected large response behavior, in particular a different page of the website was returned by the SUT which is not a security failure.

The quality of the model is based on a thorough learning cycle with reliable learning samples (quality and quantity of the samples). Figure 4(d) presents a result generated with a bad learning cycle by using too little and incomprehensive learning data. Many fields contain dark pie-charts which indicate possible security failures. However, the tested function did not contain vulnerabilities and, therefore, the reported security failures are false-positives. The uniform distribution of the learning and security test cases on the map is given, because no divergence in the system behavior between learning and security test cases occurs.

VI. Conclusion and Further Work

The presented approach shows the automated detection of security errors during security test execution. An automatically

generated system behavior model is used which is built using the system behavior observed during execution of functional test cases and an unsupervised learning algorithm. With the expected behavior from the system behavior model during penetration with simulated attacks, unknown and unexpected behavior of the SUT and, therefore, security failures can be detected.

We discussed the architecture for our approach and showed a prototype implementation which was used for security testing of a SUT. Several behavior metrics were used by the prototype to determine the final security test result based on the trained SOM. For the tested SUT the previously known security vulnerabilities could be automatically identified by the presented approach. We discussed further problem cases and could show the importance of thorough functional test cases to reduce the number of false-positive security failure reports.

The required effort of manual evaluation of false-positive detected security failures can be reduced by using the generated clusters of our approach. Only a single representative of a cluster has to be analyzed to verify similar SUT behavior and, therefore, different security test cases.

For the learning process the usage of supervised learning through functional test cases might be a valid approach to achieve a better trained model in future work because functional tests have defined expected behavior. However, for the usage in security testing not all useful system behavior metrics are defined for the functional test cases. For example, most of the time it is not defined in which range the CPU load is expected which is one useful metric for detecting security vulnerabilities. Evaluation of machine learning and clustering algorithms is required as future work to find the best algorithm for the presented problem.

With the presented approach automated test execution can be optimized by increasing the error detection rate and reducing the number of false-positive detected errors.

REFERENCES

[1] H. H. Thompson, "Why security testing is hard," *IEEE Security & Privacy Magazine*, vol. 1, no. 4, pp. 83–86, 2003.

[2] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[3] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*. Berkeley, CA, USA: USENIX Association, 2000.

[4] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2008, pp. 206–215.

[5] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009.*, May 2009, pp. 474–484.

[6] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, Mar. 2011, pp. 427–430.

[7] T. Tuglular, C. A. Muftuoglu, Ö. Kaya, F. Belli, and M. Linschulte, "Gui-based testing of boundary overflow vulnerability," in *COMPSAC '09: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 539–544.

[8] N. Antunes and M. Vieira, "Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services," in *IEEE International Conference on Services Computing*, 2011.

[9] Y. Song, S. J. Stolfo, and T. Jebara, "Behavior-based network traffic synthesis," in *Proc. IEEE Int Tech. for Homeland Security (HST) Conf*, 2011, pp. 338–344.

[10] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda, "Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and sql queries," *J. Comput. Secur.*, vol. 17, pp. 305–329, Aug. 2009.

[11] E. Martin, "Automated testing and response analysis of web services," Jul. 2007, pp. 647–654.

[12] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser, and T. Grechenig, "Automated security test approach for sip-based voip softphones," in *The Second International Conference on Advances in System Testing and Validation Lifecycle, August 2010, Nice, France*. IEEE Computer Society Press, Aug. 2010.

[13] W. Wang, S. Gombault, and T. Guyet, "Towards fast detecting intrusions: Using key attributes of network traffic," in *Proc. Third Int. Conf. Internet Monitoring and Protection ICIMP '08*, 2008, pp. 86–91.

[14] P. Steinbacher, F. Fankhauser, C. Schanes, and T. Grechenig, "Work in progress: Black-Box approach for testing quality of service in case of security incidents on the example of a SIP-based VoIP service," in *Principles, Systems and Applications of IP Telecommunications (IPTComm'10)*. New York, NY, USA: ACM, Aug. 2010, pp. 101–110.

[15] T. Kiziloren and E. Germen, "Network traffic classification with self organizing maps," in *Proc. 22nd International Symposium Computer and Information Sciences ISCIS 2007*, 2007, pp. 1–5.

[16] K. A. Jalil, M. H. Kamarudin, and M. N. Masrek, "Comparison of machine learning algorithms performance in detecting network intrusion," in *Proc. Int Networking and Information Technology (ICNIT) Conf*, 2010, pp. 221–226.

[17] X. Bao, T. Xu, and H. Hou, "Network intrusion detection based on support vector machine," in *Proc. Int. Conf. Management and Service Science MASS '09*, 2009, pp. 1–4.

[18] A. K. Ghosh and A. Schwartzbard, "A study in using neural networks for anomaly and misuse detection," in *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, ser. SSYM'99. Berkeley, CA, USA: USENIX Association, 1999.

[19] O. Linda, T. Vollmer, and M. Manic, "Neural network based intrusion detection system for critical infrastructures," in *Proc. Int. Joint Conf. Neural Networks IJCNN 2009*, 2009, pp. 1827–1834.

[20] I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," in *Proc. Second Int Advances in Computing, Control and Telecommunication Technologies (ACT) Conf*, 2010, pp. 201–203.

[21] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, no. 4, pp. 639–668, Dec. 2011.

[22] D. Gavrilut, M. Cimpoesu, D. Anton, and L. Ciortuz, "Malware detection using machine learning," in *Proc. Int. Multiconference Computer Science and Information Technology IMCSIT '09*, 2009, pp. 735–741.

[23] E. Alpaydin, *Introduction to machine learning*, 2nd ed. Cambridge, Mass., MIT Press, 2010.

[24] T. Kohonen, *Self-Organizing Maps*, ser. Springer Series in Information Sciences Series. Springer-Verlag GmbH, 2001.

[25] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Secur Priv*, vol. 5, no. 2, pp. 32–39, 2007.

[26] G. Bossert, G. Hiet, and T. Henin, "Modelling to simulate botnet command and control protocols for the evaluation of network intrusion detection systems," in *Conference on Network and Information Systems Security (SAR-SSI)*, May 2011, pp. 1–8.

[27] L. Laird and C. Brennan, *Software measurement and estimation: a practical approach*. John Wiley & Sons, 2006.