FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Semi-Automatic Security Testing of Web Applications with Fault Models and Properties

Matthias René Büchler

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl XXII - Software Engineering

Semi-Automatic Security Testing of Web Applications with Fault Models and Properties

Matthias René Büchler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Tobias Nipkow, Ph.D
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Alexander Pretschner
2.	Prof. Dr. Robert Hierons, Brunel University London, United Kingdom

Die Dissertation wurde am 20/07/2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 03/11/2015 angenommen.

Acknowledgments

First, I want to express my special appreciation and thanks to my supervisor Prof. Dr. Alexander Pretschner. I would like to thank him for his encouraging support during my Ph.D time. I appreciate his contributions in terms of time, encouragement, and ideas, since his comments and discussions were of tremendous help. I very much appreciated the freedom he gave me to pursue my own interests and still experience his support at the same time. This Ph.D thesis would not have been possible without the help of Alex.

I would also like to express a very special thanks to my 2nd supervisor Prof. Dr. Rob Hierons at Brunel University London, his comments and suggestions for my Ph.D thesis. I regularly met him at ICST conferences during my Ph.D time and enjoyed the discussions with him a lot.

I would like to express a special thanks to all group members at TU München. The group contributed not only in terms of my professional growing but it was a source of friendship as well.

During three years of my Ph.D, I participated in the SPaCIoS EU project. All the discussions and meetings have been very supportive, fruitful, and invaluable.

Finally, a special thanks goes to my family and friends who supported me all the times, their understanding and encouragement to consequently strive for my goal.

Abstract

Web applications are complex and face a significant amount of complex attacks, as well. The complexity makes manual testing of web applications for security issues hard and time consuming, thus, automated testing is preferable. To tackle the complexity, we propose a (semi-)automatic model-based testing approach. Using models, test cases are often generated using structural criteria. Since such test cases do not directly target security properties, this Ph.D thesis proposes fault models for generating tests for web applications. Described faults are known source code vulnerabilities that, by using respective mutation operators at the model level, are injected into models of a System Under Validation. A model-checker automatically analyses such models and potentially generates 'interesting' but abstract test cases. To find attacks on real systems this Ph.D thesis uses the help of web browsers and penetration testing techniques to operationalize abstract attack traces. Thus, we address the gap between abstract and executable test cases. Our evaluation on several web applications shows that our approach finds non-trivial multi-step XSS and SQL vulnerabilities, which other tools often missed. Furthermore, we evaluate the efficiency of our approach by comparing Syntactic Mutation Operators, Semantic Mutation Operators, first-order and higher-order mutation operators.

Zusammenfassung

Web Applikationen sind heutzutage sehr komplex und sind ebenso komplexen Angriffen ausgesetzt. Die Komplexität macht manuelles Testen sehr schwierig und zeitintensiv, weshalb man automatisches Testen bevorzugt. Der Komplexität begegnen wir mit abstrakten Modellen und model-basierten Testverfahren. Bei solchen Verfahren werden häufig strukturelle Abdeckungskriterien verwendet. Solche Kriterien zielen häufig nicht auf Verwundbarkeiten ab. Das Ziel dieser Dissertation ist daher, das Generieren von Testfällen auf der Basis von Fehlermodellen. Die beschriebenen Fehler werden in Modelle der Web Applikation injiziert. Modelchecker analysieren automatisch solche Modelle und erzeugen daraus abstrakte Testfälle. Um Verwundbarkeiten in realen Web Applikationen zu finden und damit das Abstraktionniveau zu überbrücken, verwenden wir Web Browser und Penetrationtests. Anhand von verschiedenen Web Applikationen zeigen wir, dass nicht-triviale XSS und SQL Verwundbarkeiten mithilfe unseres Ansatzes gefunden werden, die andere Tools oft nicht gefunden haben. Die Effizienz evaluieren wir anhand syntaktischer und semantischer Mutationsoperatoren auf einfacher und höhere Stufe.

Contents

1. Introduction and Motivation

1.1. Testing Web Applications

Modern IT systems are very often a composition of several subsystems that handle sensitive data. A successful attack executed on such systems may have severe consequences and therefore, such attacks have to be addressed by appropriate means. Such systems need to be tested before using them in a productive environment. This is not only true functional wise, but also security wise. Functional testing is focused on verifying whether the application behavior corresponds to the described functionality in the requirement documents. It is often restricted to those input values that are part of the description. The System Under Validation (SUV) is tested whether the behavior corresponds to the behavioral description for the specified input values. What functional testing does usually not cover is testing the SUV for input values that are not specified in the description. In most of the cases the set of non-specified input values is infinitely large and not all values can be tested. It gets even worse. Even not all values that *are* specified in the behavioral description can be tested due to restricted resources.

The purpose of security testing is primarily the task to test the system for harmful additional behavior not directly specified in the web application description [183]. Security testing is about validating given security properties against an implementation or an abstraction of a software product. It is about generating test cases that try to drive the SUV into a state where security related properties are violated. Typical security properties are confidentiality, integrity, availability, authentication, authorization, non-repudiation, robustness, safety, or trust, but also technical goals like No Cross-site Scripting (XSS) / No Structured Query Language (SQL) attacks. E.g., a security test tries to get access to sensitive data without a proper authentication beforehand. Another example is executing additional functionality that is not foreseen by the web application. If a functionality to store data in a database can be misused to delete data, it is a task of security testing to identify such problems. Such security properties can be classified into functional and non-functional properties. Although security testing functional properties is already challenging, testing non-functional properties is intrinsically hard because it is not only focused on functional testing of security mechanism, but affects all execution traces of the system.

Web applications in particular are a popular attack target because they are usually available on the Internet and everyone with access to the Internet can reach these applications. There is no physical access nor an own installation of the web application needed but the web application can be attacked from anywhere in the world. This has an enormous consequence when a web developer is designing and implementing security mechanisms since the number of potential attackers and corresponding attacks is enormous. Writing correct and secure web application is very challenging and difficult. Scott and Sharp [190] state, although a little bit provocative, that if we consider a web application connected to a database, the application will always be vulnerable.

The problem of insecure web applications is not only a theoretical threat but a very practical one [228]. Searching the Internet for reports about successful attacks on web applications, one immediately gets a massive list of articles, reports, descriptions and the like. E.g., the CERT Division at Carnegie Mellon University [2] reports a massive increase of reported vulnerabilities between the year 2000 and 2005 by 450%. Another such report that reports similar numbers is 'The Imperva Web Application Attack Report 2013' [13], which we briefly discuss now.

The Imperva Web Application Attack Report 2013

Web application security is a huge and important area and attacks are present every day. To get a feeling about the seriousness and wideness of web application attacks, the Imperva Web Application Attack Report 2013 [13] provides impressive numbers and statistics. The study reports about 70 observed web applications during six months. The authors overall conclude that those web applications on average got attacked on 12 days during the observation period. A web application is under attack at a specific day, if at least one attack happens at that day. In other words, each web application got attacked every 15th day on average. Of particular interest is the web application, among the observed ones, that was attacked the most. This web application was under attack on 176 out of 180 days. The report also evaluated how long a typical web application attack lasts. While the average attack duration is around 5 minutes, the longest attack lasted 935 minutes, or 15 hours.

In addition, the report further analyses the type of attacks that occurred on the observed web applications. The four most occurring attacks were SQL Injection, Directory Traversal, Cross-Site Scripting, and HTTP Violations. They report that a web application was attacked via an SQL injection about 10 times on average during the 6 month period, with a maximum of 207 attacks. Directory traversal attacks were observed on average 7 times (max=193) and cross-site scripting on average 7 times as well (max=85). Finally, HTTP violations were observed 11 times on average (max=2898).

The survey described in [42] used a dataset of 15000+ web applications in production and pre-production distributed over 650+ organizations, taking a variety of brands and industries into account. The authors report on the most common vulnerabilities in web applications in the year 2012. From their survey they report that 86% of all the websites that they surveyed were vulnerable to at least one attack. 55% of the observed web applications suffer from information leakage, where Cross-site scripting was present in 53% of the web applications. Interestingly, SQL injection was reported in 7% of the web applications 'only'. It is important to note that these probabilities refer to the presence of the vulnerability and not their exploitation. Furthermore, the report uses the term 'Information leakage' as a so called catch-all term. It refers to attacks that steal sensitive data in any way.

The consequences of insecure web applications are of different nature. Starting from loss of service where the application is responding very slowly or even not available anymore (called Denial of Service Attacks), to identity theft where an attacker operates in the name of someone else. An attack can also lead to a website infection such that the exploit of a vulnerability gets distributed to other web applications or client machines. Typical vulnerabilities that lead to the afore-mentioned attacks result from non-intentional information disclosure, predictable resource location, and missing or inefficient authorization and access control.

After getting an impression of the seriousness of the attack situation, the necessity of effective and preventive security mechanisms is a logical conclusion. Due to the complex interplay of operating systems, servers, development frameworks, and web application software, a variety of testing approaches exists together with corresponding attacker scenarios. Such testing approaches differ in the artifact under test and the attacker model. Therefore, web servers, development frameworks, protocols, and web applications themselves can be tested. As an example, Guo et al. [114] describe vulnerability assessment systems that work at the network level. They are focused on the communication channel between the client side browser and the server side web application. In the context of security testing, 'the' correct artifact to be tested does not exist. The choice of the considered software and hardware stack layer highly depends on the kind of vulnerabilities to be tested. E.g., there are many types of exploits that cannot be prevented at the network layer by using firewalls or intrusion detection systems. It is probably better to test for vulnerabilities dedicated to the Domain Name System (DNS) at the protocol level. At the same time, vulnerabilities according to XSS and SQL injections are very hard to impossible to be tested at protocol level and are better addressed at the application level. Therefore, the sets of vulnerabilities to be tested at specific layers differ.

In this thesis, we focus on an integral view of the web application and consider the web application infrastructure as a whole. We consider an arbitrary person with criminal energy somewhere in the world with the intension to attack the web application and its corresponding data. The set of considered attackers does not only include external unknown people or invited partners that collaborate together via the web application, but also internal people of an organization that use the web application on the Intranet. The set of attackers consists of all those people that have access to the web application over a network. The intension of these attackers can be of any sort — ranging from criminal and organized activities to bored individuals who attack a web application out of curiosity. Therefore, we assume that the attacker and the web application and its infrastructure are physically separated from the attacker so that no physical intervention or manipulation is possible. This also includes network devices that are involved in the connection between the client side browser and server side web application.

In the following subsections we want to target important issues when it comes to testing web applications. We briefly motivate, why automation is desired (Section 1.1.1), why we consider a model-based approach (Section 1.1.2), motivate the concept of a 'good' test case (Section 1.1.3) and the necessity to make Abstract Attack Trace (AAT) operational (Section 1.1.4).

1.1.1. Aiming for Automation

As in many other domains, automation is often very desirable and simplifies life. The same is true for testing web applications against security properties. There exists several degrees of automation of a testing approach. A testing approach can be categorized according to three non-exclusive categories:

- Traditionally, the security of a web application is assessed by a penetration tester. This concept was introduced by Farmer and Venema [101] where the authors proposed to test a system from the viewpoint of an attacker [101, 223]. The penetration tester is an expert that tries to break security assets based on his experience. A penetration tester tries out things that he is aware of but does not test for vulnerabilities, that he does not know. He injects vulnerability related malicious payloads into the SUV and observes the response of the SUV. To build the verdict (justify whether the vulnerability is present) the penetration tester requires the knowledge of both the desired and non-desired behavior. Finally, this approach is manual and automation is barely involved.

  Although manual penetration testing is very popular and widely used, it has several disadvantages and is difficult to perform because of the following aspects:

  - The overall success of penetration testing very much depends on the test expert and his knowledge. In particular, test cases are created ad-hoc instead of a systematic way.
  - Since penetration testing is not model-based, the tester must have a mental model of the properties, security mechanisms, attacks, and the environments of the system to be tested. Compared to a model-based approach, they never have to be written down and documented.
  - Not having an explicit specification leads very often to unstructured, not reproducible test case generation. As a consequence detailed rationales for the test design is missing.
  - The tester needs to think like an attacker [158].

- A step towards automation is to manually create the test cases but store them in an executable format. This allows a Security Analyst to repeatedly execute the test cases and reduce manual intervention in the execution phase. Still these test cases are often created in a non-systematic way and depend on the expertise of the Security Analyst.

- Still improving the degree of automation, not only the test execution, but the test generation procedure is automated. This has the big advantage compared to a manual approach, that test cases are generated in a systematic way. Furthermore, the expertise on which test cases are generated is structured and preserved in test generation tools or reusable knowledge libraries that are provided by experts in the field.

1.1.2. Benefits of a Model-based Approach

When test cases are generated automatically, the test generation procedure needs input artifacts from which test cases are generated. Such artifacts can be source code or any abstract model of the SUV. Following a manual testing approach based on the experience of the Security Analyst, test cases are very often created from an informal model that the Security Analyst has in his mind. In contrast, an automatic test generation approach is based on an explicit formal model. In this Ph.D thesis, the test generation and execution approach is a model-based approach based on formal ASLan++[1] [218] models that describe the web applications. One big advantage of using a model-based approach to generate test cases is the availability of an oracle. Especially in the context of automatic reasoning technologies like model checking, the tools reports a trace of the behavioral description that violates a specified property. Following the trace leads to a state that is usually interpreted as oracle. Therefore, the oracle does not need to be explicitly specified, but is automatically encapsulated in the security property.

1.1.3. Good Test Cases

Considering a formal behavioral description of the SUV, a model based approach proposes a strategy to generate a set of test cases out of the formal description. To reason whether such a proposed strategy makes sense, we need a definition of a '*good*' test case. In testing, it is a challenge to select an adequate test suite for a given test purpose. The corresponding interesting research question is how '*good*' test suites can be generated. Before any test generation approach can be proposed, we need to define the concept of a '*good*' test case. As an inspiration, we start with the definition of 'testing' in the book 'The art of software testing' [168], that states: *Testing is the process of executing a program with the intent of finding errors*. Pretschner [174] refines the definition as follows:

Definition 1. *A good test case is a test case that reveals potential defects[a] with good cost-effectiveness.*

[a] A defect can be a failure, an error or a fault. Detailed definitions of these terms are provided in Section 2.1.

To generate '*good*' test cases that reveal vulnerabilities using a model-based approach, different strategies can be applied. Literature very often proposes structural coverage criteria as a strategy for generating test cases out of a formal specification. They are widely used although valid criticism exists. Coverage is defined over a set of artifacts which are identified by structural characteristics. Such artifacts in a state transition system can e.g., be nodes and edges, artifacts in source code are e.g. lines of code, branch conditions, and so on. For illustrative purposes, let's consider a web application and represent this application as a state transition system, as shown in Figure 1.1. A structural coverage based approach over nodes generates a test suite so that every node is covered. Similar, a structural coverage based approach over edges generates test suites, so that every edge is contained in at least one test case. Therefore, an edge-based strategy with the requirement that every edge has to be covered at least once, would generate four different test cases as shown in Figure 1.1#b.

In terms of security testing, test suites generated based on structural coverage criteria are not necessarily '*good*' test suites, since we are not primarily interested in the nodes and edges per se. We are more interested in known vulnerabilities and the elements of the specification, that are affected by them. Furthermore, structural criteria based approaches traditionally require that the corresponding artifact is covered according to a pre-defined finite criterion. E.g., statement coverage requires that every statement s is executed at least

[1] A formal security specification language for distributed systems.

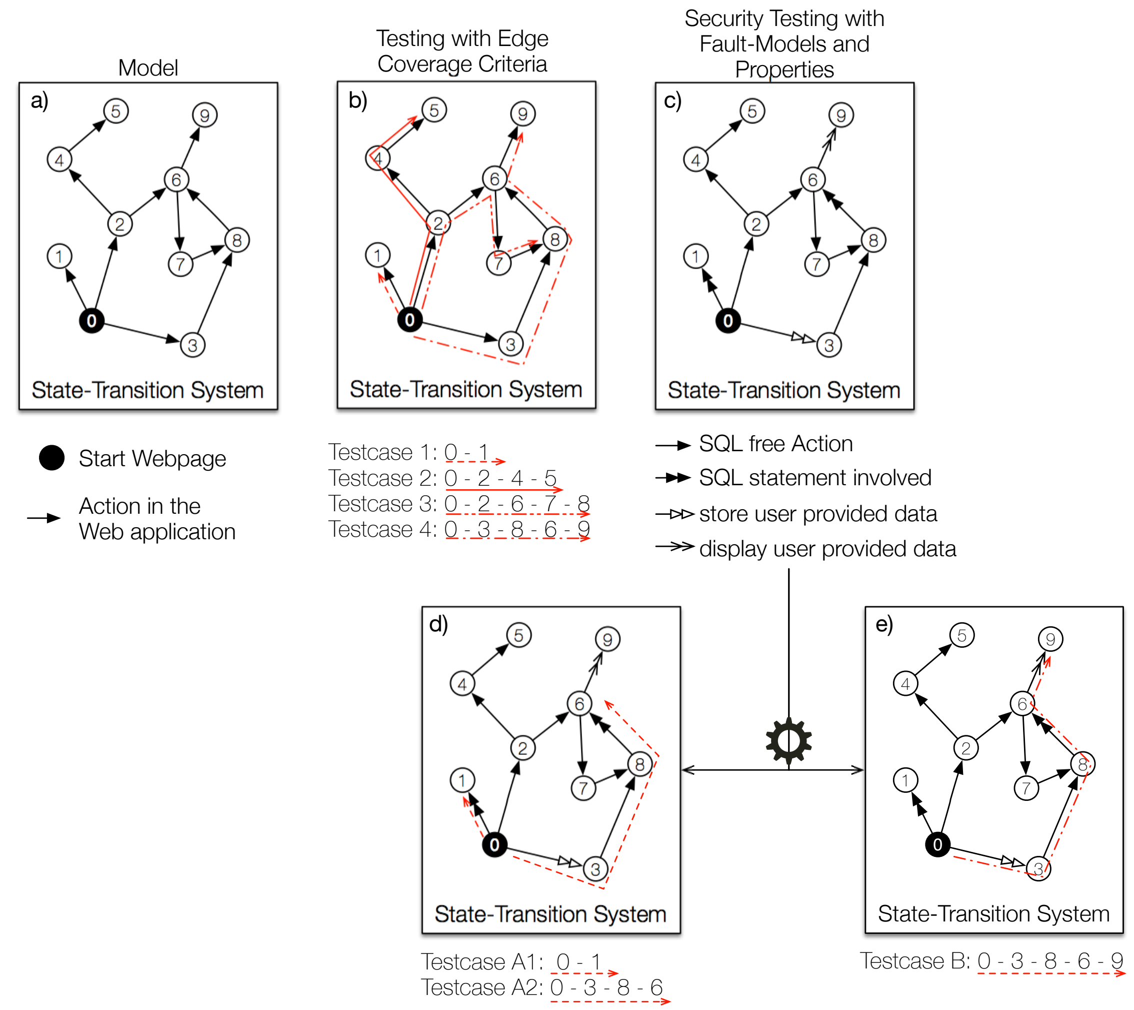

Model
Testing with Edge Coverage Criteria
Security Testing with Fault-Models and Properties
a)
b)
c)
State-Transition System
State-Transition System
State-Transition System
Start Webpage
Action in the Web application
Testcase 1: 0 - 1
Testcase 2: 0 - 2 - 4 - 5
Testcase 3: 0 - 2 - 6 - 7 - 8
Testcase 4: 0 - 3 - 8 - 6 - 9
SQL free Action
SQL statement involved
store user provided data
display user provided data
d)
e)
State-Transition System
State-Transition System
Testcase A1: 0 - 1
Testcase A2: 0 - 3 - 8 - 6
Testcase B: 0 - 3 - 8 - 6 - 9

Figure 1.1.: Why Structural Coverage Criteria Are Bad

once per test suite, branch coverage requires that every condition c is at least evaluated to true and false during the execution of the test suite. Although Martin and Xie [154] show that structural coverage correlates with faults, the correlation is not strong enough and (security) testing should consider other criteria as well. E.g., while traditional functional testing aims for the coverage of a statement s with 'well-defined' values, security testing is the evaluation, whether input parameters for a method can be passed to the web application, that statement s is executed and triggers a vulnerability. Following a traditional structural coverage based test generation approach, generated test cases would have to be combined with all possible vulnerabilities. To discover all possible vulnerabilities, one would need to try all possible improper behaviors at every interaction point in each test case. Since already traditional testing suffers from the problem of too big test suites, building the cross product of the same test suites with well-known vulnerabilities is not clever. Therefore, we argue that traditional structural coverage criteria are not sufficient and will not generate '*good*' test suites.

In terms of our example, let's assume that every edge with two filled arrows represents an action where SQL queries are involved (Figure 1.1#c). Intuitively, if the Security Analyst tests for SQL vulnerabilities, test cases that do not contain at least one edge with SQL semantics do not make sense to be generated. To cover all SQL queries, two test cases (Figure 1.1#d) are sufficient. As another example, test cases for stored XSS attacks should include edges with the semantics of storing and displaying the same data. In Figure 1.1 the abstract test case B is sufficient. Therefore, we believe that structural coverage criteria should be substituted with criteria based on semantics to generate '*good*' test cases as defined above.

1.1.4. Gap Between Abstract Traces and Executable Test Cases

A model-based approach with abstract models as input for a test generation approach has the disadvantage that the abstraction gap between the formal specification and the SUV has to be bridged. Verification tools that operate at the model level report counter examples (negative test cases) at the same abstraction level as the input model. Such AATs cannot automatically be executed on the SUV. Nevertheless, the executability of abstract AATs is crucial and important since abstract security issues found at the model level do not imply a security issue at the implementation level. Bridging this gap between an AAT and an operational test case is still an open issue.

1.2. Problem Statement

Security testing is a crucial task and required for every successful web application. Therefore, Security Analysts need effective and efficient approaches for generating security-interesting test cases for vulnerabilities dedicated for web applications. On the one hand, penetration testing is considered as the state-of-the-art security testing approach. It is a manual task and highly depends on the expertise of the Security Analyst. Good penetration testers often successfully find non-trivial vulnerabilities. On the other hand, there are many automatic black-box vulnerability scanners that are push-button tools. They find simple vulnerabilities but often fail finding more sophisticated and non-trivial vulnerabilities.

At the same time, model-based development is an emerging technology that is promising in many areas. E.g., model-based approaches together with fault-injections for test case generation are daily used in hardware industry. Holling et al. [122] use fault-models together with Simulink models. Pretschner et al. [175] propose model-based testing approaches for access control models, Martin and Xie [154] perform mutation testing for access control models, and Dadeau et al. [86] apply mutation-based testing for security protocols.

What has not been considered so far is the combination of model-based verification approaches with penetration testing techniques applied in the context of web applications. In particular, there is a gap in considering model-based approaches together with fault injections and security properties for web applications. In this Ph.D thesis, we address the problem to what extent models can be used for efficiently generating security-interesting test cases for web applications vulnerabilities. In particular, this is not trivial if the model can be considered correct with respect to security properties. To use verification techniques for test case generation on these correct models, faults can be injected. This can be achieved by using different types of mutation operators that represent different fault models. They either are motivated by the syntax of the model (called Syntactic Mutation Operators) or by its semantics (called Semantic Mutation Operators). Therefore, we elaborate on the consequences when using different types of mutation operators. The consequences will be considered both from a practical, as well as from a scientific point of view.

Along with considering a model-based approach, the challenge of bridging the abstraction gap is still an open issue. While Appelt et al. [59] already use an existing test suite and therefore start from executable test cases right from the beginning, Armando et al. [60] consider models at the protocol level and map abstract protocol messages to concrete ones. While the first approach assumes existing test cases, the latter one requires protocol level knowledge. Other approaches (like e.g., Dadeau et al. [86]) do not instantiate AATs. In this Ph.D thesis we hence also want to address the problem of automating the process of mapping abstract test cases to executable test cases without protocol level knowledge and manual API-level coding tasks.

To address these problems and evaluate the solution, this Ph.D has the following highlevel purposes that are refined in the following section:

1. Generate test cases for interesting, important, and non-trivial vulnerabilities in web applications.

2. Addressing the gap between abstract test cases, generated from the abstract model, and operationalized test cases that can be executed on the SUV.

3. Comparing the effectiveness and the efficiency of generating and operationalizing test cases generated by different fault models.

4. Supporting the Security Analyst with a complete tool chain starting from behavioral formal specifications and ending with executed test cases on the SUV for web applications.

1.3. Purpose of this Ph.D Thesis

This Ph.D has six major purposes.

- The first purpose of this Ph.D thesis is to generate test cases for interesting, important, and non-trivial vulnerabilities in web applications. The Imperva Web Application Attack Report 2013 [13] lists XSS and SQL injection vulnerabilities among the four most occurring attacks. Furthermore these two kinds of attacks are also included in the OWASP top 10 vulnerabilities [21]. Doupé et al. [92] demonstrate that automatic vulnerability scanners often fail to discover non-trivial XSS and SQL vulnerabilities.

- The second purpose of this Ph.D thesis is to generate test cases based on assets and their properties, rather than on specific functional security mechanisms. We motivate that as follows: To protect security properties of a (web) application, there are always security mechanisms involved. E.g., (i) *Anti-virus scanners* are implemented against virus injections. To be up to date, they download the newest virus-signature database every hour. (ii) *Address Space Layout Randomization (ASLR)* is a mechanisms against buffer overflows. They randomly distribute the key data areas of a process (like the stack, heap, libraries) in the memory to reduce the predictability of the addresses of those areas. (iii) *Access control mechanisms* intercept requests and decide, whether a request has to be granted and rejected respectively according to a policy to prevent un-authorized access to confidential data. (iv) *Firewalls* control ports of a system to prevent malicious data entering the system, or that intruders can access confidential data. Firewalls restrict packages sent / received via these ports based on IP, protocol, application information, etc. (v) *Intrusion Detection System (IDS)* system analyze network traffic and log files to detect unexpected events like break-in attempts.

When functional security testing is performed, mechanisms like the above mentioned ones are tested whether the mechanism behaves as they should. Nevertheless such mechanisms do not have a self-purpose. In most of the cases, it is irrelevant if a security mechanism A or B is implemented. What matters is the purpose of these security mechanisms. Therefore, a specific mechanisms is not added because it is explicitly desired by the product owner but because its purpose is to protect a higher level property. A security mechanism is in place, because the developer considers the mechanism as effective in protecting the property. More important than the mechanism itself are the assets with their properties. Therefore, the second purpose of this Ph.D thesis is to focus on assets and their properties, rather than on specific functional security mechanisms. Although a security mechanism is always in place when a property of an asset has to be protected, the security mechanism is not always dedicated to one specific component in the application. Especially in the context of web applications, sanitization of user-provided input is very likely to be distributed over the whole code basis. In contrast to a dedicated security mechanism, having it distributed makes testing much harder. Furthermore, there is no evidence that the correct behavior of a specific security mechanism indeed effectively protects the property of an asset. In a complex system, there is the risk that the security mechanism can be completely bypassed and the property of an asset can be violated following a completely different trace of the application that is not covered by the security mechanism. In such a situation, being focused on the functionality of a specific security mechanism would not be effective.

- The third purpose of this Ph.D thesis is to address the usefulness of models of web applications for test case generation. We assume that such models exist and therefore, we want to increase the benefit of model-based development approaches. Assets and their properties are given in form of a formal specification. Considering such specifications in the domain of a model-based approach using model-checkers for test case generation, the required input model can be classified into two categories. Either a model-checker reports a security issue for such an input model, or it cannot identify any issues. While an initially already ‘vulnerable’ model can directly be used for test case generation, a model without initial security issues cannot. Therefore, the third purpose of this Ph.D thesis is to address the usefulness of models without security issues by proposing an approach to make use of such models for test case generation for web applications nevertheless.

- The fourth purpose is an approach based on the knowledge of a web application end user to address the gap between abstract test cases, generated from models, and operationalized test cases that can be executed on the SUV. Due to the fact that the abstraction lacks important information of the SUV the Security Analyst is asked to provide a mapping to bridge the gap. How to bridge this gap is still an open issue.
- The fifth purpose is demonstrating one possibility how to meaningfully evaluate such security testing approach. Especially in the academic world, the evaluation is an important aspect that is very difficult and often an open issue how to do.
- Finally, this Ph.D thesis aims for tool support beyond a prototype implementation that implements the conceptual workflow. The sixth purpose is to automate as much as possible and to provide tutorials for those tasks that the Security Analyst has to perform manually.

1.4. Hypothesis

A ‘security-interesting’ test case is a test case that reveals known potential vulnerabilities to violate a security property with good cost-effectiveness. In this context, our hypothesis is:

> In a model-based context, non-trivial security vulnerabilities for web applications can be found with mutation operators and fault injections.

Mutation operators can inject faults according to different fault-models. Different kinds of mutation operators perform differently in terms of effectiveness and efficiency due to the level of automation and the requirement of manual tasks. While Syntactic Mutation Operators are fully automatic, Semantic Mutation Operators operate on manually provided semantic annotations. As a sub-hypothesis, we claim:

> Using Semantic Mutation Operators is more efficient than using Syntactic Mutation Operators.

1.5. Solution

The proposed solution for the problem formulated in Section 1.2 consists of the following:

- To generate interesting test cases for web applications where the assets and their properties are in the focus, we provide a comprehensive methodology, starting from formal specification written in ASLan++ and ending at executable test cases. The methodology is based on fault-models and security properties instead of specific security mechanisms. In correspondence with the purposes of this Ph.D thesis, the fault-models are dedicated to XSS and SQL vulnerabilities. Nevertheless the proposed methodology is not restricted to these two domains. We believe that the whole methodology is more general and can be instantiated and applied to other fault-models as well.

- To achieve a comprehensive methodology, we provide a combination of formal verification and penetration testing for web applications because automatic penetration tools are very effective in finding simple (reflected) vulnerabilities, but often fail with more complex, multi-step vulnerabilities. At the same time, formal verification applied to an abstract specification of the web application can find AATs for non-trivial security vulnerabilities. Penetration testing techniques are then applied during the execution of operationalized test cases.

- If models are used for test case generation, the literature very often proposes structural coverage criteria either to generate a test suite or to evaluate the quality of an existing test suite. Depending on the kind of model and the purpose of testing, there is often no evidence whether such coverage criteria are a '*good*' test selection criteria. Furthermore, Martin and Xie [154] claim a weak correlation only between structural criteria and faults. Therefore, in this Ph.D thesis, we provide a set of (higher-order) Semantic Mutation Operators to generate interesting test cases from correct formal specifications. They are initially not useful for test case generation because they are correct and every trace fulfills the specified security properties. Semantic Mutation Operators in contrast to structural criteria, generate test cases due to the semantics of the considered model and the exploitation of the injected vulnerabilities. Such test cases are not based on purely structural criteria.

- Considering a behavioral model, our proposed methodology in addition mitigates weaknesses that Doupé et al. [92] claim for automatic vulnerability scanners. They demonstrate that future research is needed for an effective web application vulnerability detection since automatic vulnerability scanners miss many (8 out of 16) crucial and well-known web vulnerabilities. In particular the paper concludes that a 'deep' reach into the application's resources is crucial. We provide a mutation-based approach to combine correct formal specifications with injected vulnerabilities at the model level, and penetration testing.

- To closing the gap between abstract test cases, generated from the model, and operationalized test cases, we propose a two step mapping to bridge the level of abstraction between AATs and test cases that can be executed on a SUV. The mapping is supported by an intermediate language called Web Application Abstract Language. This language allows to instantiate AATs with the help of the browser and testing frameworks like Selenium [26]. Furthermore, a given Web Application Abstract Language (WAAL) mapping for a formal specification can automatically be applied to any AAT generated from this specification. Besides the fact that this language simplifies the instantiation of AATs, this way of addressing the gap also contributes to handle complex and state-of-the-art web applications. Such application often make use of sophisticated client-side techniques like Flash, Frames, and Javascript. Doupé et al. [92] state that automatic security scanners often fail to handle them.

- This Ph.D thus does not only conceptually contribute to the identified issues, but provides a practical tool called SPaCiTE that supports the Security Analyst during modeling the web application and the generation of executable test cases in a comprehensive and systematic way. Test case generation, their execution and maintenance is a tedious task which has to be automated as much as possible. Therefore, the tool development goes far beyond a prototype implementation, and substantial parts of the contribution consists of technical work. We provide an update site for the Eclipse platform to effortlessly integrate SPaCiTE into Eclipse, an Eclipse wizard to automatically create and configure a SPaCiTE project, a modern IDE with code completing, code templates, syntax highlighting, quick fixes, etc., an automatic generation of an initial skeleton dedicated to the current formal specification, a dynamic loading procedure to integrate and apply mutation operators to a formal ASLan++ specification, and an automatic procedure to apply the WAAL mapping to AATs and to generate TestNG test cases written in Java.

1.6. Contribution

We see the contribution of this Ph.D thesis as follows:

- A comprehensive and semi-automatic methodology to use correct formal specifications with security properties and fault injections to generate security-interesting test cases for web applications. By combining formal verification techniques with penetration testing, non-trivial web application vulnerabilities can be found, if present, by executing the above generated test cases on a SUV. This contributes to the gap that automatic vulnerability scanners often miss non-trivial vulnerabilities and penetration testing is rarely automated.

- A set of Syntactic and Semantic Mutation Operators to mutate ASLan++ models that allow the use of correct formal specifications for test case generation. This addresses the gap of automatically generating test cases from ASLan++ models for web applications, after providing annotations for Semantic Mutation Operators.

- A declarative Web Application Abstract Language to automatically instantiate all generated AATs of a given formal specification. AATs consist of free-form abstract application-dependent messages. A WAAL mapping addresses the gap that Security Analyst often require source code API or protocol level message knowledge to instantiate abstract test cases. Using WAAL a Security Analyst that knows how to use the web application in the browser and does not know anything about protocol level messages can operationalize AATs.

- A tool beyond a prototype implementation for ASLan++ that allows to semi-automatically combine verification techniques and penetration testing for web applications.

- Finally, this Ph.D thesis contributes with a comparison and insights of Syntactic Mutation Operators (instance of a syntax-based fault model) and Semantic Mutation Operators for web application vulnerabilities (instance of a vulnerability-based fault model) in terms of effectiveness and efficiency.

1.7. Thesis Organization

In Chapter 2 describes our semi-automatic security testing approach with fault models and properties. We present our terminology in Section 2.1, the overall overview of the approach in Section 2.2, discuss important components like security properties (Section 2.4), correct and mutated models (Sections 2.3 and 2.5), mutation operators (Section 2.6) and the mapping of AATs to operational test cases (Section 2.7). Chapter 3 then focus on SPaCiTE, the tool that implements the approach. We provide tutorials how SPaCiTE is installed, configured and used to generate and execute test cases (Sections 3.1, 3.2 and 3.4 to 3.9). In Section 3.3 we provide guidelines for modeling SPaCiTE-conform models of web applications. In Chapter 4, we evaluate our approach by applying it to three different web applications. We discuss the evaluation strategy in Section 4.1 and show the effectiveness of SPaCiTE in Sections 4.2 to 4.4. In terms of efficiency we compare syntactic, semantic, first-order and higher-order mutation operators in Section 4.5. We discuss the results and a future large scale evaluation set-up in Chapter 5. Finally, Chapter 6 covers related work and Chapter 7 concludes.

Notation

In this Ph.D thesis we use the following notation.

- To refer to a specific part of a figure, we use the notation of the form *Figure 1#a*. It references part a of Figure 1.

- Semantic annotations have the form `%@Semantics[A,B,...]` where A and B are called semantics keywords.

- *Source code* listings are provided in boxes as shown in Listing 1.1. Please note the special symbol in the caption at the right side. It contains an ‘S’ that stands for ‘source code level’.

- *Model* listings are provided in boxes as shown in Listing 1.2. Please note the special symbol in the caption at the right side. It contains an ‘M’ that stands for ‘model level’.

- *Abstract Attack Trace* listings are provided in boxes as shown in Listing 1.3. Please note the special symbol in the caption at the right side. It contains the character ‘A’ that stands for ‘AAT’.

- *WAAL mapping* listings are provided in boxes as shown in Listing 1.4. Please note the special symbol in the caption at the right side. It contains the character ‘W’ that stands for ‘WAAL mapping’.

- *Command line inputs* listings are provided in boxes as shown in Listing 1.5. Such listings have the characters ‘CL’ in the captions that stands for ‘Command line’.

Listing 1.1: Source Code Example in Java

```
public class Test {
        public static void main(String[] args) {...}
}
```

Listing 1.2: ASLan++ Example

```
entity {
        body {...}
}
```

Listing 1.3: AAT Example

```
AAT 1: MESSAGES:
<tom> *->* webServer : login( username( tom ),INJECT( Unknown ) )
webServer *->* <tom> : VERIFY( tom )
```

Listing 1.4: WAAL Mapping Example in ASLan++

```
SIMULATED_AGENTS { "tom", "jerry" }
AGENTS_CONFIGURATIONS {...}
```

Listing 1.5: Command Line Example

```
cd /tmp
make
```

2. Semi-Automatic Security Testing of Web Applications with Fault Models and Properties

Chapter 2 introduces the approach of this Ph.D thesis to semi-automatically generate and execute security test cases on the basis of formal specification, fault models, and properties. In Section 2.1, we provide the terminology of important security related terms. Then Section 2.2 gives an overview over the whole workflow and all the components that are involved in the approach. This approach is implemented by a tool called SPaCiTE. Sections 2.3 to 2.7 provide more details for each component and each step of the workflow, including a guideline for modeling SPaCiTE conform models.

2.1. Terminology

This Ph.D thesis is based on the fact that software developers make mistakes during the implementation phase. Therefore, it is important to clarify first what 'make a mistake' means. Unfortunately there is no standard definition of the relevant terms like 'fault, 'vulnerability', etc. In addition, experts see the concept of a vulnerability at different levels of abstractions. In this section we provide our terminology borrowed from Pretschner [174].

Valid Specification / Requirements. A valid specification of a (web) application is a mental imagination of the correct application in the perfect world. It describes what the product owner wants, and includes behavior and security considerations. In particular, the behavior fulfills the desired security properties. It is a conceptual view of the (web) application and does not exist in written form but only in the head of the product owner (see Figure 2.1#a).

Behavioral Description. A behavioral description is an artifact that the developer of a (web) application creates after communicating with the product owner. Such a description can be source code, a diagram, a specification or any other artifact produced during the software development process. In general, a developer produces formal and informal behavioral descriptions. In our approach, we focus on formal specifications of a web application, that consists of a model M and a set of security properties Φ. We call such a pair '*consistent*' if every trace of M satisfies the defined security properties Φ ($M \models \Phi$), otherwise '*inconsistent*'.

Correct Behavioral Description. If for a behavioral description BD, consisting of a model M and the security properties Φ both M and Φ conform to a valid specifications, we call the pair M and Φ a '*correct*' specification. In particular, a correct specification fulfills $M \models \Phi$. No trace in M violates any of the formalized security properties in

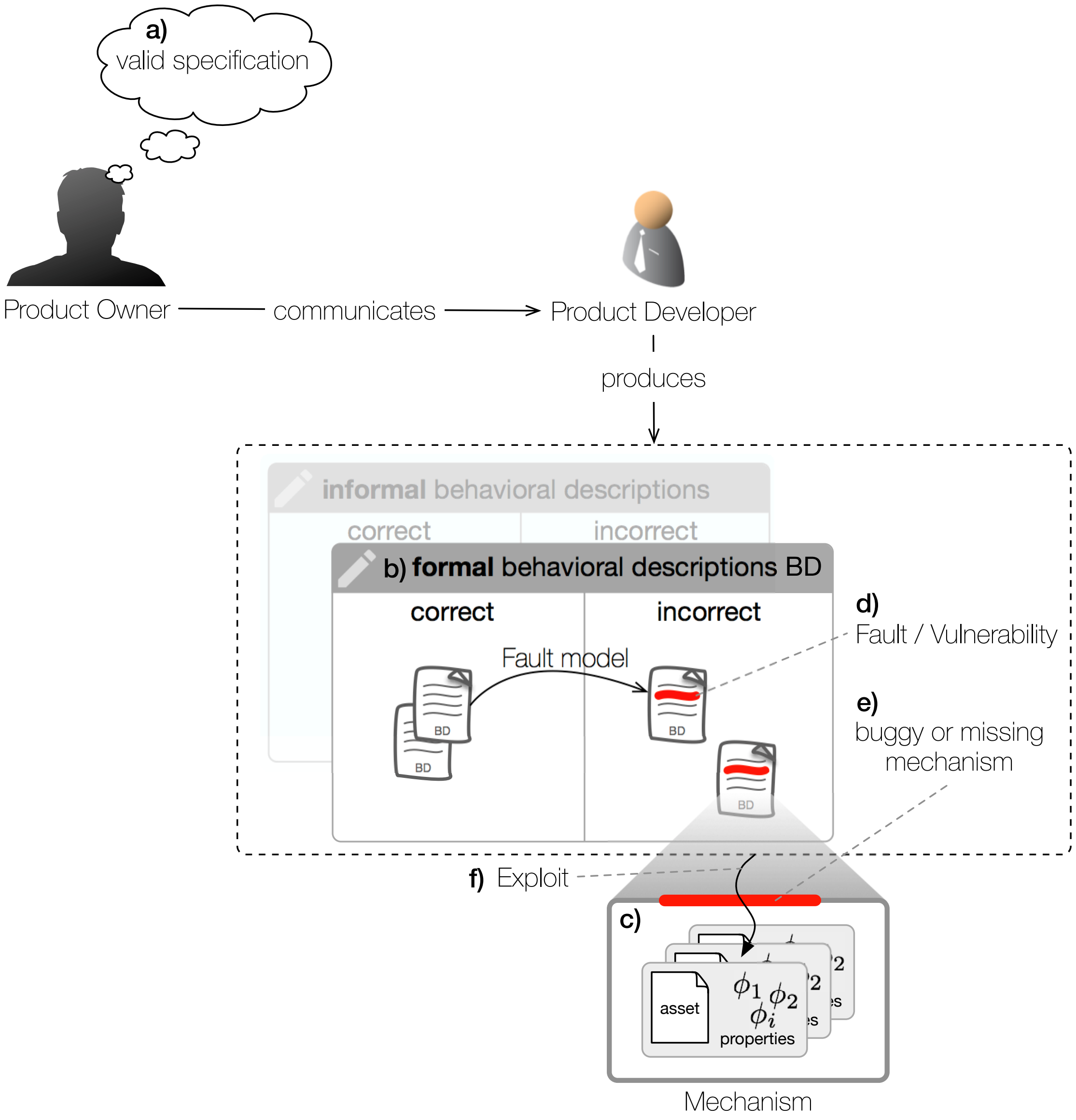


Figure 2.1.: Definition Illustration

Φ. A model checker will not find an attack for an intruder (attacker) that belongs to the considered attacker model. No specified entity in M can perform an action and no malicious input (=malicious payload) can be provided to a method so that the defined security properties are violated. In contrast, an '*incorrect*' formal specification does not match with the mental imagination of the product owner. In terms of our considered formal (web) application specifications, either M or Φ or both do not match with the valid specification. Please note, that an 'incorrect' formal specification can be consistent or inconsistent (see Figure 2.1#b).

'Long-Running' Behavioral Description. This definition is pragmatic and practically motivated. A 'long-running' behavioral description consists of a model M and a set of security properties Φ such that the Cl-Atse model checker [213] does not find any security property violations within a time frame of length t. Therefore, M is considered to be 'consistent' to Φ with respect to t. This definition is weak since t depends on many factors — the specific version of the model checker, available time for testing, computation power, possible parallelism, and so on. Therefore, the definition of a 'long-running' behavioral description is context dependent.

Asset. An asset is a data item, an artifact that has to be protected. In the security context, such an asset has one or multiple security properties assigned. The valid specification determines the situations in which the security property has to hold and must not be violated. Such a property does not necessarily need to hold at all times. As an example, a key of a VPN software like IPSec is only accepted in a specific time frame. While the confidentiality of the key is very important during the time when the key is accepted, it might not be important anymore when the key is not accepted anymore (see Figure 2.1#c).

Failure. A failure is the inability of a software system or component to perform its required function [131, 109]. Pretschner et al. [177] defines failures as follows: 'Failures [...] are observable differences between specified and actual behaviors.' A failure is a characteristics of a runtime system. The system needs to be executed in order to observe a failure.

Error. Pretschner [174] defines an error as follows. 'An error is an incorrect internal state of the program.' Observing a failure always implies that an error occurred in the system. The other direction is not true. E.g., in fault tolerant systems, errors might occur, but due to redundant components they might not lead to a failure.

Fault. Pretschner [174] defines a fault as the textual representation of what goes wrong in a behavioral description (see Figure 2.1#d). It is the incorrect part of a behavioral description that needs to be replaced to get a correct description. Since faults can occur in dead code — code that is never executed —, and because faults can be masked by further faults, a fault does not necessarily lead to an error. On the other hand, an error is always produced by a fault. A fault is not necessarily related to security properties but is the cause of errors and failures in general. E.g. a fault can be the reason why an implementation of a mathematical function does not return the correct value, or why a sorting algorithm does not sort negative values as expected.

Vulnerability. A vulnerability is a special type of fault (see Figure 2.1#d). If the fault is related to security properties, it is called a vulnerability. A vulnerability is always related to one or more assets and their corresponding security properties. An exploitation of a vulnerability attacks an asset by violating the associated security property. In practice, vulnerabilities have been distinguished into three categories (see Gegick and Williams [109]) as follows:

- **Implementation bug** — a vulnerability at the code level, such as buffer overflows, not checking return codes, and unsecured Structured Query Language (SQL) statements [159].
- **Design flaw** — a vulnerability that is related to the design of the system and can occur even if the program is well-coded. Examples of design flaws include a lack of or incorrect auditing/logging, ordering and timing faults, and improper authentication [159, 168]. As an example, the authors of the book 'Software Security: Building Security In.' [159] mention inconsistent error handling as a design flaw.
- **Operational vulnerability** — a vulnerability in the configuration, environment, or general use of the software [93].

Since vulnerabilities are always associated with the protection of an asset, the security relevant fault is usually correlated with a mechanism that protects the asset. A vulnerability either means that (1) the responsible security mechanism is completely missing, or (2) the security mechanism is in place but is implemented in a faulty way (see Figure 2.1#e).

Mistake. Gegick and Williams [109] describe a mistake as: 'A human action that produces an incorrect result. Note: The fault tolerance discipline distinguishes between the human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), [. . .]. A mistake is the human action that leads to the actual vulnerability in the software although the vulnerability may never be exploited.'

Exploit. An exploit is a concrete malicious input that makes use of the vulnerability in the System Under Validation (SUV) and violates the property of an asset (see Figure 2.1#f). We assume that an SUV always has at least one asset and each asset has at least one security property. In the easiest scenario, the SUV itself is the asset with the availability property. Since a vulnerability is not bound to one asset or one security property, it can often be exploited in different ways. One concrete exploit selects a specific asset and a specific property, and makes use of the vulnerability to violate the property for the selected asset. Furthermore, usually an asset (including its properties) is not protected by one single mechanism. E.g., to attack the confidentiality of an asset, an exploit might use an Cross-site Scripting (XSS) vulnerability to steal the asset from the client side. Another possibility is to exploit an SQL vulnerability with a malicious query to steal the same asset from a server-side database.

Attack. An attack is a sequence of steps, that an attacker performs on a SUV. Such an attack always contains at least one exploit.

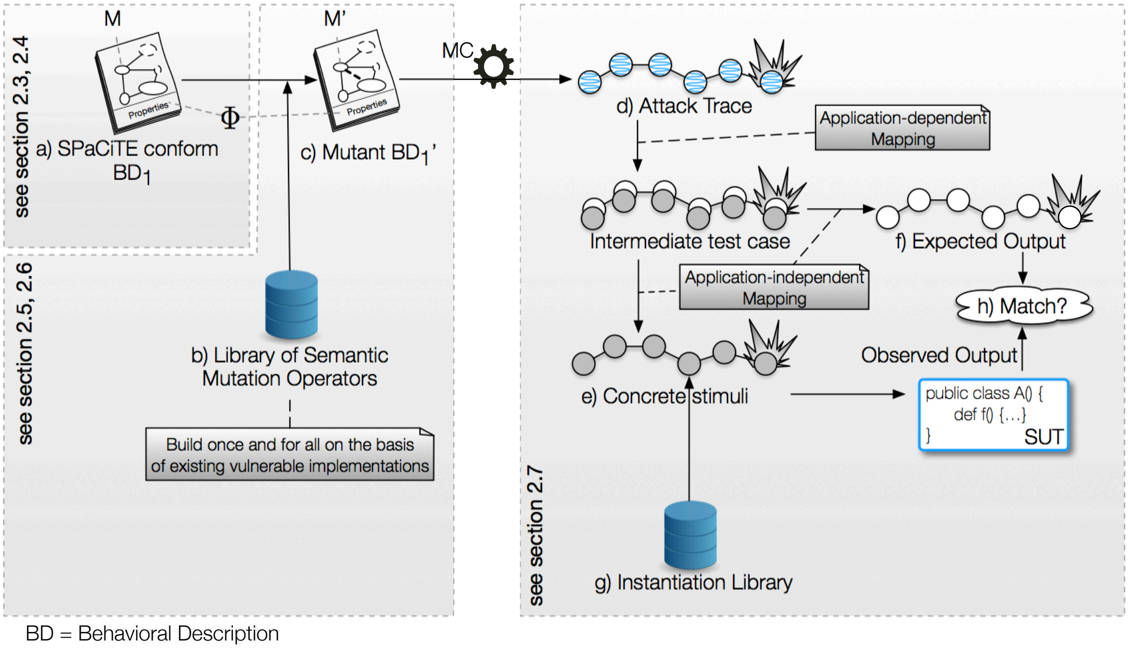


Figure 2.2.: Overall SPaCiTE Workflow

Good Test Case. In Section 1.1.3 we already discussed the concept of a 'good' test case. As a reminder, *a good test case is a test case that reveals potential defects with good cost-effectiveness*. This definition is now extended by the definition of an interesting test case.

Security-Interesting Test Case. For this Ph.D thesis, we define a security-interesting test case as a 'good test case' as defined by Pretschner [174] that is related to known vulnerabilities. An 'interesting' test case in the context of security testing exploits a potential vulnerability with the goal of violating a security property.

2.2. Overview of the SPaCiTE Workflow

As already motivated in Chapter 1, we propose a semi-automatic testing approach starting from a formal specification and end with operational test cases executed on a SUV. In this section we give an overview of the whole approach and the involved components.

Our proposed approach is a mutation-based approach that is both vulnerability-driven and property-driven. The starting point is a SPaCiTE-conform behavioral description which is either a 'correct'[1] or 'long-running' formal specification BD_1[1] of the SUV written in ASLan++, a formal security specification language for distributed systems [218] (Figure 2.2#a). Adding 'long-running' formal specification to the set of SPaCiTE-conform behavioral descriptions is motivated by practical reasons. The assumption of having access to a correct behavioral description of the web application to be tested is very strong and not

[1] According to section 2.1

always practical in daily live. Especially for bigger web applications, useful formal specifications get so big that a model checker usually cannot decide whether $M \models \Phi$ or $M \not\models \Phi$ holds within a practical period of time t. Therefore, we weaken the requirements for the input models suitable for our approach by also including 'long-running' formal specifications. Both kind of behavioral descriptions have in common, that a model checker cannot directly be used for test case generation since they will not generate counter examples that could be used for test case generation (e.g., by interpreting them as negative test cases). Therefore, one of the contribution of this Ph.D thesis is to show how such behavioral descriptions might nevertheless be used for test case generation.

Definition 2. *A SPaCiTE-conform behavioral description is either a 'correct' or a 'long-running' formal specification, written in ASLan++. A 'correct' model, given to a model checker, does terminate and the model checker reports the verdict: NO_ATTACK_FOUND. In contrast, model-checking a 'long-running' behavioral description with Cl-Atse will not lead to any security property violation within a time frame of length t.*

As defined in Section 2.1, a SPaCiTE-conform behavioral description BD consists of two parts, a model M of the web application and a set of security properties Φ (e.g., confidentiality, authentication, no XSS, no SQL injection) that the SUV has to fulfill. Furthermore, the model M of the web application contains semantic annotations provided by the Security Analyst to incorporate the meaning of different components of the specification. Because the behavioral description is assumed to be 'correct' or 'long-running', a model checker will not find, in a reasonable time period, any trace in M that could be interpreted as a test case that violates the specified security properties (and negating the properties for correct models will yield *any* trace as counterexample). Therefore, SPaCiTE applies Semantic Mutation Operators to the model M. These capture vulnerabilities commonly found in web applications and represent them at the abstract model level. The mutation operators are collected in a library that has been built once and for all by a Security Analyst (Figure 2.2#b), until new vulnerabilities are found. Nevertheless the library is easily extendable with new vulnerabilities if needed. To generate test cases using a SPaCiTE-conform behavioral description, the mutation operators are applied to the model, resulting in several mutations BD', where the model M but not the set of properties Φ is mutated (Figure 2.2#c). If for a defined property $\phi \in \Phi$ we have that $M' \not\models \phi$, automatic reasoning technologies (e.g., a model checker) yield a trace that violates ϕ in M' (Figure 2.2#d). In the context of web applications, such a trace consists of abstract messages that are exchanged between different components defined in M. The reported trace exploits a vulnerability (captured by the applied mutation operator) that yields the violation of ϕ. After concretization, this trace is a useful test case for the SUV: if the SUV is vulnerable, then the concretized trace is an exploit, otherwise the trace increases confidence that the vulnerability is not present.

A reported Abstract Attack Trace (AAT) by a model checker is expressed at the model level and has no direct relationship to the implementation. To verify whether the implementation suffers from the reported abstract security issue, SPaCiTE provides a semi-automatic execution engine for executing AATs. To make AATs operational, a mapping of the AAT to executable test cases is used. To provide such a mapping, the SUV first has to be split into simulated and non-simulated components. This splitting in turn depends on the applied Semantic Mutation Operators. Given such a splitting of the SUV, the messages of an abstract attack trace are partitioned into messages that are generated (concrete stimuli, Figure 2.2#e) and messages that are verified (expected output, Figure 2.2#f).

The mapping of AATs to operational test cases consists of two sub mappings. First, the Security Analyst defines a mapping from abstract messages of the AAT to abstract actions in a web browser expressed in the *Web Application Abstract Language (WAAL)* (e.g., click, insert text). The intuition is that after performing these actions in the browser, the desired concrete message (e.g., HTTP message), which is represented by the abstract message, is generated or verified by the browser. Using a second mapping, the abstract WAAL actions are mapped to executable API calls of a framework that can communicate with a real browser. The need for concrete malicious input during the execution of abstract attack traces is addressed by an instantiation library, whose elements describe concrete attack vectors and verification steps to check if the attack could be successfully executed (Figure 2.2#g). The selection of instantiation library elements for an attack trace is mainly driven by the applied Semantic Mutation Operator (e.g., if an SQL related vulnerability was injected, only SQL related malicious inputs from the instantiation library are used). Finally, applying the mapping results in an executable test case that generates the stimuli messages and compares the output from the SUV with the expected output given by the AAT (Figure 2.2#h).

In the following sections we will discuss the different components of our approach in more details.

2.3. SPaCiTE-Conform Behavioral Descriptions

As described in Section 2.2, our approach requires a SPaCiTE-conform behavioral model. In practice, already assuming a model in itself is a very strong assumption. It is even a stronger assumption, if the model has to be ‘consistent’ (see Section 2.1). To support the Security Analyst to initially come up with such SPaCiTE-conform behavioral descriptions, we provide a modern IDE for ASLan++. Furthermore, for practical purposes, we provide guidelines to support the Security Analyst with the modeling activity.

2.3.1. Considered Abstraction Level

[The concepts of the following paragraphs was already published in the SPaCIoS Deliverable 2.1.2 [199]].

A SPaCiTE-conform model is an abstraction of a real implementation of the SUV. Being an abstraction means that irrelevant aspects of the implementation are ignored and left out of the model. Since models used by verification tools might suffer from the state explosion problem, abstraction is very desirable. At the other side, if security-related aspects are abstracted away, the verification tool will not check for attacks based on these missing aspects.

Considering web applications as SUV one may consider the following two abstraction levels (Figure 2.3). At the technical level, the client side browser communicates with the server using protocol level messages. In the majority of the cases, these messages are HTTP messages. Modeling a web application at this level requires that the Security Analyst is familiar with the protocol and is able to express web application behavior in terms of protocol

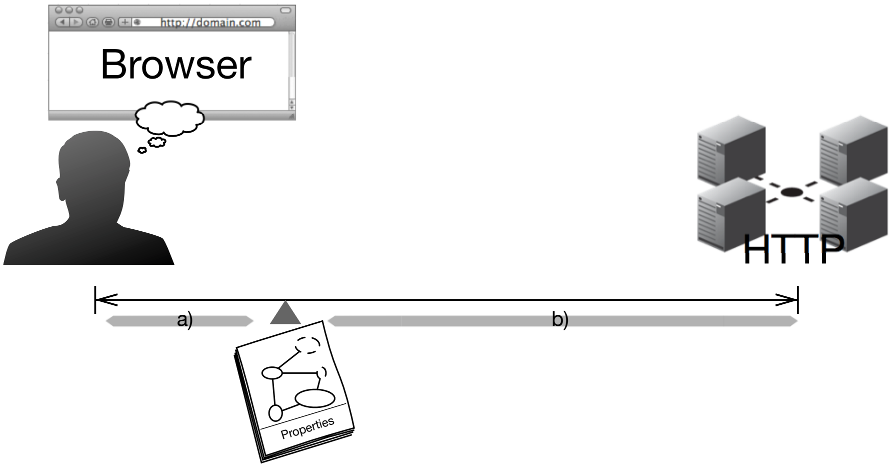


Figure 2.3.: Modeling Level

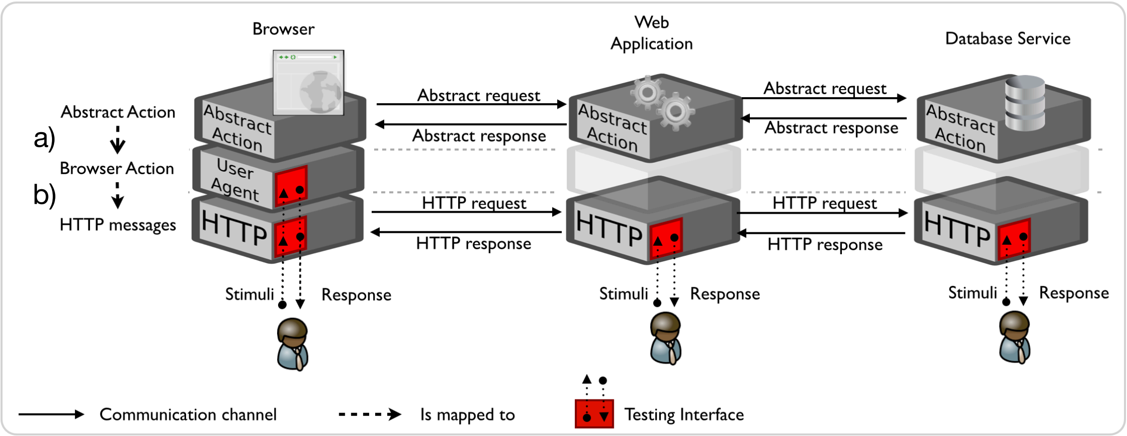


Figure 2.4.: Levels of Abstraction For Modeling Web Applications

messages. At the same time, the web application behavior can be expressed more abstractly at a conceptual level. Doing so, the behavior is formalized in terms of abstract messages that directly capture the logical action, neglecting the details of exchanging protocol-level messages (Figure 2.4). Since we will focus on abstract messages rather than protocol level messages, we show how such abstract messages may look like. Assume an example web application that allows a user to log in and view profiles of other users. Listing 2.1 shows the corresponding abstract messages to describe the communication between the browser and the web application. The `login` message requires two parameters, a username of type 'agent' and a password of type 'symmetric_key', whereas the 'viewProfileOf' message only requires one parameter of type 'agent'. Compared to modeling the same functionality at protocol level, low level details are abstracted.

Listing 2.1: Abstract Messages in ASLan++

```
login(agent, symmetric_key): message;
viewProfileOf(agent): message;
```

Modeling a web application in terms of abstract messages has several advantages but also some drawbacks with regard to modeling the same functionality at the HTTP level. The main advantages are:

- For a Security Analyst, it is more natural and easier to express the formal specification of the web application at the same level as he is using the web application. It better reflects the view-point of an end-user. In particular, the Security Analyst can benefit from the knowledge he has about how to use and navigate in the browser. In particular, the Security Analyst does not have to have knowledge about low level HTTP messages and the like. Since protocol level messages contain a lot of low level details like protocol flags and network related information, that are irrelevant for SQL and XSS attacks, they can be hidden or ignored because the browser will generate these details automatically (Figure 2.4#b). In addition, if e.g., the `login` functionality requires the exchange of several protocol level messages, they can be combined to one highlevel abstract message so that the modeling activities get (potentially) simpler.
- Expressing the communication between the client and the server in terms of abstract messages is sufficient for the type of vulnerabilities that we consider in this Ph.D thesis. XSS attacks and SQL injections are all exploited at the browser level and not at the protocol level. Therefore, it is much easier to recognize such security issues at the browser level, rather than on the protocol level.
- Executing actions in the browser and letting the browser be responsible for protocol level messages has the advantage that encryption and decryption is automatically handled by the browser. Furthermore, SPaCiTE can be augmented with a browser plug-in that intercepts HTTPS requests before they are encrypted and HTTPS responses after they are decrypted. Therefore, our approach benefits from the advantage that also encrypted communication can be easily handled.

Besides all the advantages, the disadvantages are:

- Expressing the web application's behavior in terms of browser actions neglects low level protocol details. Therefore, vulnerabilities that are exploited at the protocol level are not modeled and thus, cannot be recognized at the model level. E.g., HTTP splitting, unless modeled, cannot be automatically detected by the model-checker.
- The disadvantage of considering the browser as the abstraction level is that the mapping from abstract traces to executable traces (Figure 2.3#b) gets more complicated for the Security Analyst because potentially more information is abstracted away that the Security Analyst has to provide again. At the same time, choosing a protocol-level close abstraction level would make the mapping specification easier for the Security Analyst, although specifying the behavior of the web application (Figure 2.3#a) gets more challenging at the same time.

As Figure 2.3 shows there is no win-win situation where both the modeling and the mapping task gets easier and simpler. These two tasks contradict each other in the sense that making the modeling more abstract and therefore easier, the mapping task gets more complicated (Figure 2.3)#a,b). Nevertheless, in the special context of testing for XSS attacks and SQL injections, we focus on modeling closer to the browser level rather than protocol level. Involving the browser level, the mapping can be split into two mappings to turn AATs operational. Applying the first mapping to an AAT expresses it in terms of abstract browser actions (e.g., click on a button, follow a link, type text in an input field). Applying the second mapping to abstract browser actions, these actions are mapped to a framework API that communicates with a concrete browser. The good news is that the second mapping is tightly bound to the framework used to automate the browser and can be reused for every SUV. Therefore, the Security Analyst needs to care for the first mapping only.

Considering the two discussed modeling levels, we follow, for this Ph.D thesis, the approach of describing the web application's behavior in terms of abstract messages at the browser level. To turn reported AATs from such models operational, we ask the Security Analyst to provide a mapping from the abstract attack trace messages to abstract actions performed in the browser (Figure 2.4#a). This step requires that the Security Analyst knows the web application and is able to describe abstract messages in terms of browser actions that he performs to achieve the abstract messages. As an example, to map the message `login(username, password)` to abstract browser actions, the Security Analyst needs to describe that e.g., he first selects his username from a dropdown menu, then types his password into the textfield, and finally clicks on the button 'Sign in'. In contrast to a model at the protocol level, this approach does not require that the Security Analyst is familiar with protocol level messages. The Security Analyst can concentrate on browser actions. We believe that this is more natural and adequate not only for the Security Analyst but for the modeler as well.

2.3.2. ASLan++ in 5 Minutes

Since we focus on formal specification written in ASLan++, we provide in this section a brief introduction to the ASLan++ language. This language is used as foundation for the guidelines for modeling SPaCiTE-conform models of web applications in Section 2.3.3, as well as for the described mutation operators in Section 2.6. Listing 2.2 illustrates an example model which shows the following basic ASLan++ concepts[2]:

Symbols Section. In ASLan++ every fact, variable, constant, message, etc. needs to be declared before it can be used. SPaCiTE provides a dedicated section for these declarations as part of an entity (Lines 2 to 8). Such a section starts with the keyword `symbols` and contains a non-empty list of declarations.

Variable. A variable name has to start with an upper case character and has a type. E.g., `ProcessedText` in Line 5 is a variable of type `text`.

Actor. This is a special variable that contains a reference to the current entity. In Line 11 it is used as the receiver of the message `echo`.

[2] The interested reader can find a comprehensive introduction to ASLan++ in the AVANTSSAR Deliverable 2.3 [62].

Messages. Different entities can exchange messages. In Line 11 the current entity `Actor` is waiting for a message `echo` from an entity `Browser`. The arrow (*→*) represents the channel between the two entities. The two stars express that the channel has the authentication and secrecy property.

Question-marked Variables. A variable can be extended with a question mark as shown in Line 11. Such a variable learns the actual value during model checking. In the concrete example given in Listing 2.2 the `Browser` sends a parameter with the message `echo` and the server assigns that received parameter to variable `A`.

Assignment. A value is assigned to a variable using the assign operator (:=). In Line 12 of our example, the value stored in variable `A` is mapped to the expression `process(A)` of type `text` and this expression is stored in the variable `ProcessedText`.

Facts. ASLan++ allows the introduction of facts and the evaluation to their boolean values. A fact can be set to true and false by using it as a statement, or it can be evaluated when it is part of a condition. Once a fact is introduced (Line 13) this fact is evaluated to `true` in any state from now on. When an earlier introduced fact is retracted (Line 14), or the fact was not introduced so far, the fact is evaluated to `false` in any state from now on.

Listing 2.2: ASLan++ Example Model

```
 1 entity {
 2         symbols
 3                 Browser : agent;
 4                 A : text;
 5                 ProcessedText : text;
 6                 process(text) : text;
 7                 output(text) : fact;
 8                 echo(text) : message;
 9         body {
10                 select {
11                         on( Browser *->* Actor: echo(?A): {
12                                 ProcessedText := process(A);
13                                 output(ProcessedText);
14                                 retract output(ProcessedText);
15                         }
16                 }
17         }
18 }
```

2.3.3. Guidelines for Modeling SPaCiTE-Conform Models of Web Applications

Equipped with a basic knowledge about ASLan++, we describe in this section a possible way to model web applications so that the models are useful for our approach. It is important to stress that this section presents recommendations and experiences. It is neither complete nor does it exclude other ways of modeling. Usually, the same concept can be modeled in different ways, and still be suitable for the SPaCiTE approach. In this section we concentrate on the important high level aspects. Low level details and a formal introduction to ASLan++ is not provided as part of this Ph.D thesis. The interested reader is advised to read the ASLan++ tutorial [62].

2.3.3.1. Modeling Different Instances of the Same Value

As we will see later in this section when we discuss how to model the client side agent and the server side component, we repeatedly need a way to represent several copies of the same value. E.g., when a web application stores a value V in a database, the value is copied and stored at two different locations at the same time (e.g., in a variable at the web application and in the database). Due to this duplication and technical details of such store actions, one of the values may be sanitized, whereas the other is not. Since such sanitization properties of a value are very crucial in the context of XSS and SQL injections, we first show a way how to model such copies of the same value before we continue with the client and server side components.

Let's consider a message handler that receives a user-provided value. The task of the message handler is to store this value in a database. After this step we have two copies of the same value. In the presence of stored XSS and SQL injection vulnerabilities, different instance of the same value should be distinguished. Although the message handler has sanitized the value in the message handler variable, the same value stored in the database does not necessarily be sanitized anymore due to the way how e.g., SQL stores values in the database. As an example, to store the character sequence `admin'` in the database, the quote needs to be escaped, which can be achieved by sanitizing the character sequence. Because the sanitization is 'used' during storing the value, the stored value has 'lost' its sanitization. For modeling this behavior, there exists many ways how to do so; we present one possibility. In the models that we refer to in this Ph.D thesis, we use symbolic functions to represent different copies of a value V of type t. We use the symbolic function to bind a value to a random variable to make it unique. The Security Analyst introduces an expression of type t of the form `instance_[t]( V, text ) : t` where `[t]` is substituted by the type t and the 'text' parameter is a random value. E.g., whenever a message handler receives a username V of type `username_t`, it expects it in the form `instance_username_t(V, randomValue)`. Such an expression can now be used to store a value V in the database by creating a new random value for value V. Modeling copies of data values happens a lot and therefore, SPaCiTE supports the Security Analyst with a providing a skeleton as a content assist.

2.3.3.2. Modeling the Server Component

A web application model always consists of at least two components. The first component represents the client side browser, called 'client' entity. The second component represents the web application running on the web server, called 'server' entity. We start with modeling the server component. If the modeler has used SPaCiTE to create a new modeling project, SPaCiTE already created a skeleton with two entities. Therefore, we describe the body of the server entity in this section. A web application is usually accessible via the HTTP(S) protocol and provides a set of requests that it can handle. For each request, the server component implements a corresponding message handler. The web server has no control over the client and therefore, any request can be sent to the application at any time in any order. We model this behavior using the `select` statement inside a while loop. The `select` statement with its containing `on` statements is used to non-deterministically choose one of the `on` statements that evaluates to true (see Listing 2.3). The while loop guarantees that multiple such choices can be made. This way of modeling guarantees that the server component does not enforce a specific sequence of exchanged messages. For more details, we refer the interested reader to the following AVANTSSAR Deliverable 2.3 [62]. To demonstrate how such a message handler is modeled, assume a user needs to login first before using the web application. Therefore, we add the `login` message as an `on` statement to the `select` statement (Line 6 in Listing 2.3). The method assumes two parameters, a username of type `username_t` and a password of type `password_t`. As discussed in the previous Section 2.3.3.1, both parameters are expected in the special form `inst_[t](Value,Nonce)` where `Nonce` represents a random value.[3]

Listing 2.3: Modeling Messages at the Server Component

```
 1 entity Server(User,Actor : agent ) {
 2  body {
 3   while( true ) {
 4    select {
 5     on( ?User *->* Actor :
 6      login(inst_usr_t(?Username,?Nonce1),inst_pwd_t(?Password,?Nonce2))
 7      & inSUT(?User)
 8      & !auhenticated(?User)
 9      & db(inst_usr_t(?Username,?),inst_pwd_t(?Password,?))
10     ) : {
11       // SELECT-ON-BODY
12     }
13    }
14   }
15  }
16 }
```

Usually, receiving the `login` message is bound to some constraints. We add them to the `select-on` statement (Lines 7 to 9 in Listing 2.3). As an example, the server only accepts the `login` message if the user is defined in the specification (expressed with the fact `inSUT()`) (Line 7), if the user is not authenticated yet (Line 8), and if the provided username and password are stored in the database `db` (Line 9). The actual behavior of the server upon successfully receiving a `login` message is added between brackets right after the `select-on` statement (Line 11). In a message handler body, we e.g., introduce new facts, store expressions in variables, or send a message back to the client.

2.3.3.3. Modeling a Message Handler With Sanitization

A message handler usually reads parameter values, processes them and eventually sends back a response to the client. When it comes to security issues like XSS attacks or SQL injections, sanitizing input data plays a crucial rule. We will discuss these two types of

[3] Due to space limitations, we write `inst_[t]` instead of `instance_[t]`.

vulnerabilities in details in Section 2.6, when we introduce Semantic Mutation Operators. For now, we only provide an intuition, why sanitization is crucial and how the Security Analyst annotates corresponding modeling blocks with semantic keywords.

In Figure 2.5 we see different activities of a message handler. `param1` is a parameter that is used at the implementation level in a database query to read `param3`. To do so, user-provided data is mixed with SQL control data. To prevent a malicious input to change the semantics of the SQL query, the user-provided data needs to be sanitized against SQL before using it. `param2` is a user-provided input that is directly reflected in the response of the message handler. Similar to the SQL vulnerability, if such input is mixed with HTML or Javascript code, malicious input can change the semantics of the webpage. Therefore, the value of this parameter must be sanitized against XSS. Finally, `param3` is a value that is first stored in the database and later read and sent back to the client. Therefore, it needs to be sanitized against SQL so that a malicious input cannot harm the system during the SQL query. In addition, it needs to be sanitized against XSS as well, since it is sent back to the client. The XSS sanitization can either occur before that value is stored, or after it is read from the database but before it is sent back. Nevertheless the latter case is more advised since an already sanitized value in the database might be overwritten by a non-sanitized value after being stored.

Modeling the sanitization functionality can be done in different ways since ASLan++ does not have a pre-defined language construct for sanitization. In this section we present two different approaches:

Representing Sanitization Functionality as Facts. User-provided data is either effectively sanitized against XSS or SQL or it is not. To model this decision, ASLan++ provides facts. Facts are expressions that are evaluated to true or false. Since fact names are arbitrary, the modeler choses one specific name that he uses for sanitization. W.l.o.g. let's assume that the modeler uses the facts `sanitize_xss()` to represent a function that sanitizes against XSS. Therefore, to represent a value V of any type T that is sanitized, the fact `sanitize_xss(V)` is introduced. Facts in ASLan++ are global properties. Therefore, applying the fact to value V does not have to be stored in a variable and V can be used normally. To check, whether value V is sanitized, the fact `sanitize_xss(V)` is evaluated to its boolean value.

Representing Sanitization Functionality as Symbolic Functions. A different approach is based on symbolic functions. Let's again assume a value V of any type T. This time, the modeler represents the sanitization functionality against XSS as a symbolic function `sanitize_xss` of type $T \rightarrow T$. To sanitize a value V, the symbolic function is applied and the result is stored in a new variable (`V' := sanitize_xss(V)`). To check whether a variable V'' is sanitized, one has to check if it is of the form `sanitize_xss(?)` where ? is a placeholder for any matching value.

Finally, as we have seen for `param1`, not every parameter that is used in a SQL query is also stored in the database. E.g., they can also be used for a selective purpose by adding constraints to an SQL query. Those variables are affected by SQL injections in the same way as parameter values that are stored in the database. To enable the verification tool used by SPaCiTE to identify vulnerabilities related to SQL, we need to track which values were involved to store or read a data d from the database. There is no single solution for

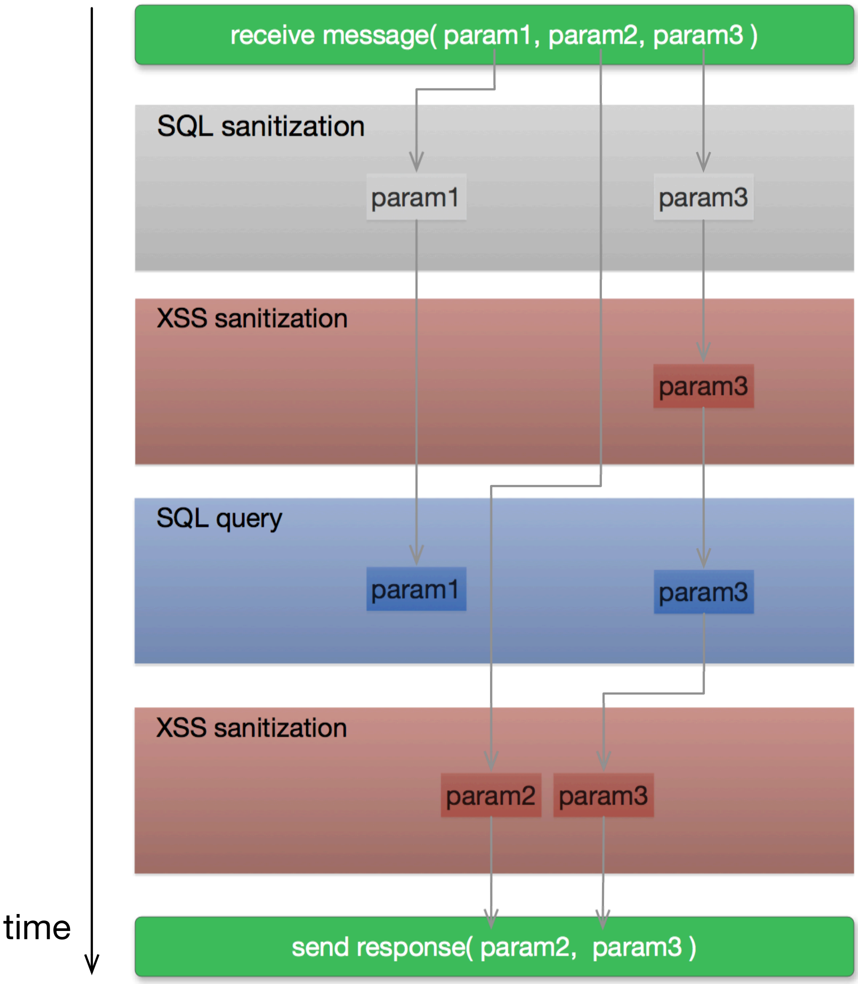


Figure 2.5.: SPaCiTE: Message Handler Modeling

this. The possibility we present here is one among many. It is the one that we used for our models. We add a fact of the form[4]:

Listing 2.4: Keeping Track Which Values Are Stored

```
writtenToDB(user-provided-value1,corresponding-db-value1);
```

We define the semantics of the parameters of the fact `writtenToDB` as follows. The first element represents the user-provided input that is used to build the SQL query to store the second parameter in the database. E.g., if the message handler receives a username tagged with a random value Y1 and stores the same username in the database tagged with a random value Z1, we introduce the fact `writtenToDB(inst_username_t(username,Y1), inst_username_t(username,Z1))`. For another example, assume that the username tagged with the random value Y2 is used to identify a profile. The profile contains a counter c of type nat (tagged with the random value Z2) that is incremented by 1. In this example, the username is used in the query to update the counter c. Therefore, we introduce the fact `writtenToDB(inst_username_t(username,Y2), inst_nat(c,Z2))`. This explicit modeling of the dependencies of different values allows the Security Analyst to specify effective security goals that check for SQL injections.

2.3.3.4. Propagating Sanitization to Different Instances of the Same Value

When user-provided values are stored in a database, the specific sanitization method applied to the values to be stored might be propagated to the values finally stored in the database. To model this propagation, we apply the sanitization also to the values stored in the database. For illustrative purposes, assume that the client has sent a value v to the server that correctly sanitized it. After the sanitization, the server component stores the sanitized data in the variable V which is then used for the SQL query. Let V_{DB} refer to the corresponding value stored in the database. Because the user-provided value is correctly sanitized, the sanitization of V is propagated to the value V_{DB} stored in the database as well. Following the modeling of unique values described in Section 2.3.3.1 we need to track that the value V_{DB} stored in the database is a copy of the user-provided value V. This link is provided by the semantic annotation `Implicit(`V`)` in Listing 2.5.

Listing 2.5: SPaCiTE: Implicit Sanitization

```
%@Semantics[ HTML, Sanitize, Implicit(V) ]
sanitize_xss( VDb );
```

2.3.3.5. Modeling the Client Side

Compared to the server side modeling, the client side modeling is much simpler. While the server-side entity is modeled with a while-loop and a select statement, we only model a single trace for the client-side entity, such that every message is covered at least once. The client behavior basically consists of sending messages to the server and receiving the

[4] A similar fact is used when data is read from the database.

response. In terms of XSS and SQL vulnerabilities, it is important that we introduce two different agent instantiations. We introduce an agent that is honest and plays the role of the client by following and respecting the specified behavior. For certain vulnerabilities, this is too restrictive. Therefore, we also introduce an agent that is dishonest. Such an agent can arbitrarily mix the messages and therefore, it can exploit vulnerabilities, that an honest client would not do. To declare an honest agent `jerry`, it is sufficient to introduce the constant `jerry` of type `agent`. To declare an agent `tom` to act dishonestly, it is important to note that each agent has access to, among other artifacts, three private keys —inv(ak), inv(ck) and inv(pk), where inv() is the inversion function applied to the public keys. Therefore, we declare the agent `tom` like an honest agent but add the fact `dishonest(tom)` to the specification and provide the agent's three private keys `inv(ak)` (used for authentication), `inv(ck)` (used for confidentiality), and `(inv(pk)` (used for generic purposes) to the intruder i (see Listing 2.6). To make knowledge available to intruder i the special fact $iknows()$ is used.

Listing 2.6: Declaring an Agent as Dishonest

```
dishonest( tom ) ;
iknows( inv( ak( tom ) ) ) ;
iknows( inv( ck( tom ) ) ) ;
iknows( inv( pk( tom ) ) ) ;
```

Security Goals Specification for XSS and SQL Injections

Security goals in ASLan++ are specified in the goal section of any entity (see AVANTSSAR Deliverable 2.3 [62]). As already discussed in Section 1.1, we consider any agent with malicious and dishonest intentions an attacker. Therefore, honest server and client entities are considered as potential victims of an attack. Since several goals section are available to be selected by the Security Analyst for specifying the security properties he needs to decide for each security property where to put them.

A Security Analyst might be interested in test cases for stored XSS and stored SQL injections where one or multiple agents are involved. In the latter case, a dishonest agent injects malicious code into the application that is later in time used by the honest victim of the attack. This includes that the security goal does not have to be violated while the dishonest user injects the malicious payload. In ASLan++, a security goal specified as part of an entity, that is played by a dishonest agent, is never violated. Since that allows an attack where a dishonest client entity and a victim client entity are involved, the `goals` section of the client side entity is a good candidate for a security goal that captures stored XSS / SQL injections. The security goal specified at the client side will allow the dishonest attacker to inject the malicious payload and allows an honest agent to be attacked. In terms of reflected SQL and XSS injections, no interplay between a dishonest and an honest agent is needed and such security goals are advised to be specified at an honest server entity.

2.4. Security Properties

Following a model-based approach in combination with model checkers, security properties build an essential part of the formal specification. The properties themselves are not the central part of this Ph.D thesis, although they are needed to pursue this approach. Therefore, we briefly discuss common security properties, that often are used in the context of web applications. The provided list of properties are not definitions, but explanations only. The set of security properties can be partitioned into two subsets. One subset covers highlevel, standard security properties like confidentiality, privacy, authentication, authorization, integrity, accountability, or availability. The other set of properties are closer to vulnerabilities. We call them technical security properties. E.g., such a property might state that all user-provided data need to be sanitized before it is processed. This applies to XSS, as well as to SQL vulnerabilities. For completeness reasons, we first give a brief overview of highlevel security properties in Section 2.4.1. We will see that only two out of the seven discussed security properties are directly supported by built-in features of ASLan++. All the others need to be expressed by the Security Analyst himself. The lack of built-in detection features in ASLan++ for XSS and SQL related attacks motivates the necessity of more vulnerability-related security properties. Therefore, after Section 2.4.1 we then focus on technical security properties in Section 2.4.2.

2.4.1. General Highlevel Security Properties

Highlevel security properties including their formalization in ASLan++ are discussed in deliverable D2.1.1 [198] of the SPaCIoS project. The purpose of the following list of properties is to provide a brief overview of different highlevel security properties and how ASLan++ supports them.

Confidentiality is one of the most important properties and one that always gets mentioned first when taking about security properties. Confidentiality expresses that a given resource is only accessible to a set of subjects known in advance. It not only means that the resource itself has to be protected from unauthorized access, but also the credentials that can be used to access the resource. In general, encryption is used as a security mechanism to assure the confidentiality property. In ASLan++, the property expresses that a protected value should not be learned by an intruder. Confidentiality can be expressed with ASLan++ built-in expressions. A confidential value can be exchanged by different agents using confidential communication channels. To guarantee confidentiality, correct authorization is necessary, and to guarantee authorization, authentication is necessary. Therefore, the three concepts are closely related to each other.

Privacy can be considered as a special form of confidentiality with the extension that privacy also talks about data ownership. ASLan++ does not offer a built-in construct to express privacy, but it can nevertheless be expressed. The confidentiality part of privacy can be expressed with the built-in ASLan++ construct. The ownership aspect can be expressed using facts (e.g., `isOwnerOf( agent, text )` with the semantics, that the 'text' is owned by the specified 'agent'). Since the fact can be introduced multiple times, each time with different parameters, this fact can be used to express that 'text' is owned by multiple 'agents'.

Authentication is closely related to the confidentiality and authorization property. Authentication is usually enforced by either a PKI mechanism or on the basis of user names and passwords. Examples how such a property can be expressed in ASLan++ can be found in the AVANTSSAR Deliverable D2.3 [62].

Integrity claims that the data generated by an entity A is not tampered with or modified by an intruder. Sending data to entity B, B can verify that the data was not modified after entity A has sent it. In particular, if a dishonest agent or an intruder is able to modify a piece of data, that is still accepted by the system, the integrity property is violated.

Authorization is very closely related to the confidentiality property. The authorization property can only be enforced if the authentication property is enforced correctly as well. Authorization is in contrast to confidentiality not a built-in construct at the model level. Therefore, the Security Analyst has to formalize this property by himself. Therefore, von Oheimb and Mödersheim [218] present an example how authorization can be modeled using ASLan++. The formalization makes use of horn clauses and predicates (see AVANTSSAR Deliverable 2.3 [62]). Temporal changes can be expressed by LTL formulas.

Accountability expresses that if an agent A performs an action, A can later not deny the fact that he triggered the action. Accountability cannot be expressed with ASLan++ built-in constructs. It has to be 'simulated' by using secrecy and authentication, as well as LTL formulas.

Availability is a liveness property and in general hard to model check. ASLan++ does not offer built-in constructs to express this property. One could try to approximate the liveness property by a safety property, expressing that at most x requests are issued within a given time slot.

2.4.2. Technical Security Properties

A conclusion of the discussion of highlevel properties in the previous section is that detecting XSS and SQL related security issues with models following the guidelines in Section 2.3.3 require more vulnerability-related security goals. In particular, since ASLan++ does not support the Security Analyst with built-in features for input and output sanitization, such desired security goals need to be formalized separately. Therefore, we will discuss such security property formalization in this section.

No XSS. Security goals for XSS attacks can be formalized in different ways. A first approach was to use security properties that expressed characteristics of the attacks and state, that this behavior must never happen. E.g., a security goal for a stored XSS attack expressed that it must never happen that a user-provided value is stored non-sanitized by one message handler, read and sent back non-sanitized to the client using a second message handler. To provide an intuition how such a security property might look like, consider the following goal specification in Listing 2.7:

A security goal section in ASLan++ starts with the keyword `goals` and is followed by a list of goal specifications. Each specification starts with a unique name. Separated by a colon (:), the goal is formalized using LTL. The following LTL formalization expresses that if `Data` is read by any `User1` in a non-sanitized way, it either had to be stored sanitized previously by any other `User2`, or it is a predefined value that was not stored by any user.

Listing 2.7: Formalization of a Stored XSS Security Property
([] = globally; <-> = past; => = implies)

```
goals
  storedXSSDetailed1:
    forall User1 User2 Data.
    [](
        (
        read(User1, Data) &
        notSanitizedByRead( Data )
        ) => (
          <->(stored(User2,Data) & sanitizedByStore(Data))
           | !stored(User2, Data)
        )
    );
```

Such security goals are non-optimal due to several reasons. First, they easily get complex and hard to understand. The discussed example in Listing 2.7 is non trivial and already suffers from missing corner cases. Second, they are dedicated to one possible attack scenario. Such a security goal can be used for a stored XSS injection, but not for other XSS attack scenarios. Third, if a Security Analyst has to specify the security goal in such great details, it is very likely that the Security Analyst has a very clear understanding what he wants to test for and how the attack looks like. If so, he most likely will not use model checking technology since revealing such attacks (traces / sequences of behavioral steps) is the task of a model checker.

Therefore, we make use of much simpler security goals in the formal specifications that we use for our evaluation. We express the technical security property: `No XSS` as an assert statement. Whenever a client agent receives data from the webserver and that data is supposed to be rendered by the browser, the corresponding value needs to be sanitized against XSS. Let's assume that a value v is sanitized, if the fact `sanitize_xss(v)` is evaluated to true[5]. The corresponding `No XSS` security goal can be expressed by the following assert statement.

```
assert XSSSanitized: sanitize_xss( v );
```

The above line is then added after each message receive statement at the client entity specification. The advantage of such a simple goal is that it covers a variety of different attacks, ranging from reflected to (multi-step) stored XSS attacks. Furthermore, it is much easier to specify and understand. In contrast to the previous security goal in

[5] This semantics of the fact sanitize_xss is given by the Security Analyst, not the ASLan++ language.

Listing 2.7, it focuses on the final end result of an attack, and not on the attack steps to violate the `No XSS` security property.

No SQL Injection. For the `No SQL injection` security property, we provide two different goals. The choice which one is selected in a specific formal specification depends on how 'sanitizing against SQL' is formalized. For the first example, we assume that 'facts' are used to model the sanitization functionality (see Section 2.3.3.3). Therefore, whenever a data element `data` is used as part of an SQL query, it has to be sanitized against SQL. To illustrate the formalization of such a security goal, let's assume a fact `writtenToDB(A,B)` with the following semantics. In order to store data element B in the database, the user-provided value A is used to construct the corresponding SQL query. Furthermore, the fact `sanitize_sql(data)` represents the property whether `data` is sanitized. In our specific models, the modeler distinguishes between queries used to store and read data from the database. Therefore, we specify the `No SQL injection` security property twice, both for reading and writing. The security goals are expressed as LTL goals that globally have to hold. For every data element `data`, whenever the fact `writtenToDB` is evaluated to true, `data` has to be sanitized against SQL, i.e., `sanitize_sql(Data)` needs to be evaluated to true as well.

```
goals
  sql1: forall Data. [](
    writtenToDB(Data,?) => sanitize_sql(Data)
  );

  sql2: forall Data. [](
    readFromDB(Data,?) => sanitize_sql(Data)
  );
```

Using facts to express the sanitization functionality is not the only possibility. As already discussed in the Section 2.3.3.3 sanitization functionality can also be expressed with symbolic functions. As an example, a value v is sanitized against SQL, if it appears in the form `sanitize(v)`. Note that the expression `sanitize` is not a fact, but a symbolic function. Furthermore, the semantics of this symbolic function is given by the Security Analyst, and not by ASLan++. A non-sanitized value v is represented by v only, i.e., the symbolic function `sanitize` is not applied. If symbolic functions are used, the `No SQL injection` security goal is expressed as the following LTL formula. It expresses that the symbolic function `sanitize()` must be applied to every data element `Data` that is part of an SQL query (captured by the fact `writtenToDB`). In more details, if the fact `writtenToDB` is true for an element `Data` and an arbitrary second argument, then the element `Data` must be of the form sanitize(?), where ? is an arbitrary value.

```
goals
    noSQL: forall Data. [](
      writtenToDB(Data,?) => (Data = sanitize(?))
    );
```

2.5. Mutate Models

[The following section was already published in the paper 'Security Testing with Fault-Models and Properties' [81]]

In the previous sections, we have discussed SPaCiTE-conform models and how a Security Analyst can come up with such models. For this section we assume to have access to such models. According to the overall SPaCiTE workflow in Figure 2.2, such SPaCiTE-conform models are not directly usable for test case generation. Therefore, we discuss in this section how to use SPaCiTE-conform models for test case generation and introduce the corresponding mutation operators in Section 2.6. These steps correspond to Figure 2.2#b-c.

As we have seen in Section 2.3, SPaCiTE-conform models (correct or long-running models) do not generate counter example that can be used for test case generation. Literature in the context of model-based testing often propose structural criteria on models for test case generation. They are on a syntactic level and with a few exceptions (e.g., Fraser and Wotawa [107]) are not related to security properties due to the lack of an obvious link between structural criteria and security properties. Therefore, we discuss in this section the problem how 'interesting' test cases can be generated that test a SUV for violations of security properties, under the assumption that such a violation is indeed present in the SUV. As defined in Section 2.1, an 'interesting' test case is a good test case that exploits a potential vulnerability to violate a security property. At the model level, model checking can verify security properties Φ and reported counter examples can be considered as abstract test cases. To make use of models that do not generate AATs for test case generation, either the specified security properties Φ or the model M needs to be mutated. If the underlying model M is correct, and therefore, $M \models \Phi$, mutating properties is not immediately useful for test case generation since all traces of the correct model M satisfy all original properties ($\forall \phi \in \Phi : M \models \phi$). If the security properties are mutated, the model checker can only report traces from the correct model that all satisfy the initial security properties. Therefore, the model M itself needs to be mutated to M' so that a trace in M' exists that violates at least one specified property ($\exists \phi \in \Phi : M' \not\models \phi$). Model checking tools might now report traces of the mutated model that violate an initial security property $\phi \in \Phi$ (① in Figure 2.6).

By mutating correct formal specification and model checking them, we combine penetration testing and model checking techniques. Penetration testing techniques provide knowledge about security properties and source code vulnerabilities that could be exploited by an attack to violate a security property. Source code vulnerabilities are captured by model-level mutation operators and are injected into a correct model. Model checking techniques are used to report AATs that violate the security property due to the injected vulnerability (see Figure 2.6). The advantage of using model checker for finding counter example is, that these tool are fully automatic and they don't require any user intervention. Since a single syntactic change might not be powerful enough to represent a source code vulnerability at the model level, they are aggregated into Semantic Mutation Operators. They are hence higher-order mutation operators that consider the semantics of the model as well. E.g. for access control policies, Martin and Xie [154] show evidence that structural coverage for test selection is far from optimal for fault-detection effectiveness (see Chapter 6). Using our approach, generated test cases are 'interesting' (① in Figure 2.6) because they test for

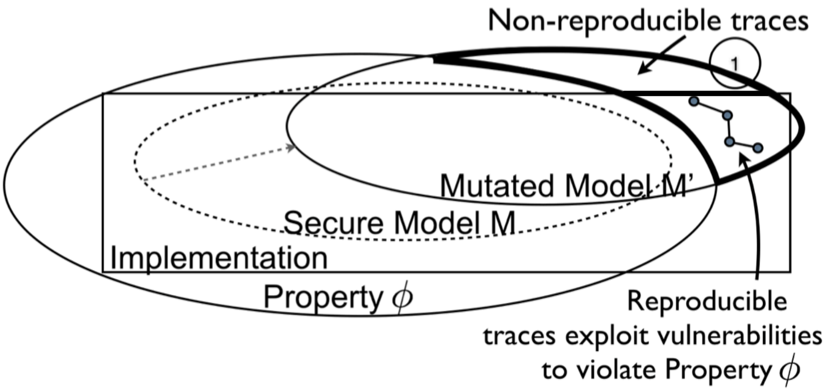


Figure 2.6.: Characteristics of Interesting Test Cases

specific vulnerabilities, acknowledge the presence if the vulnerability indeed exists, or raise confidence that the vulnerability would have been found if present.

The purpose of injecting vulnerabilities and generating test cases for the mutated model is to show conformance of the implementation to the mutated model. Furthermore a test case generated from a mutated model that could successfully be reproduced on the SUV concludes the non-conformance to the original model. Since vulnerabilities are injected to the behavioral model but not into the security properties, a successfully reproduced AAT also shows non-conformance to the security property. What remains unclear is the relation between syntactic mutations and generating test cases in the context of security testing. We believe that the set of test cases generated from Syntactic Mutation Operators is too large to represent a good and meaningful test suite. Therefore, one contribution of this Ph.D thesis is a set of mutation operators that are dedicated to vulnerabilities.

2.6. Mutation Operators

In this section we describe the mutation operators for mutating formal models written in ASLan++. Figure 2.8 shows the general hierarchy of the defined mutation operations and Figure 2.7 shows a graphical overview over all defined mutation operators including the dependencies between them and how they are categorized.

Definition 3. *Whenever a mutation m is performed on an element e of a model, e must match the requirements of the mutation. E.g., a mutation operating on statements is only applicable on elements that are statements or inherit from statements. Otherwise the mutation cannot be performed. We call an element e a* **mutation candidate for m***, if it is type correct with respect to the mutation m.*

At the top of the hierarchy are the `Syntactic` and `Semantic Mutation Operators`. A `Syntactic Mutation Operator` takes a `General Mutation Operator` m and considers every possible mutation candidate for m of the specification. This is different for `Semantic Mutation Operators`. They only apply a `General Mutation Operator`

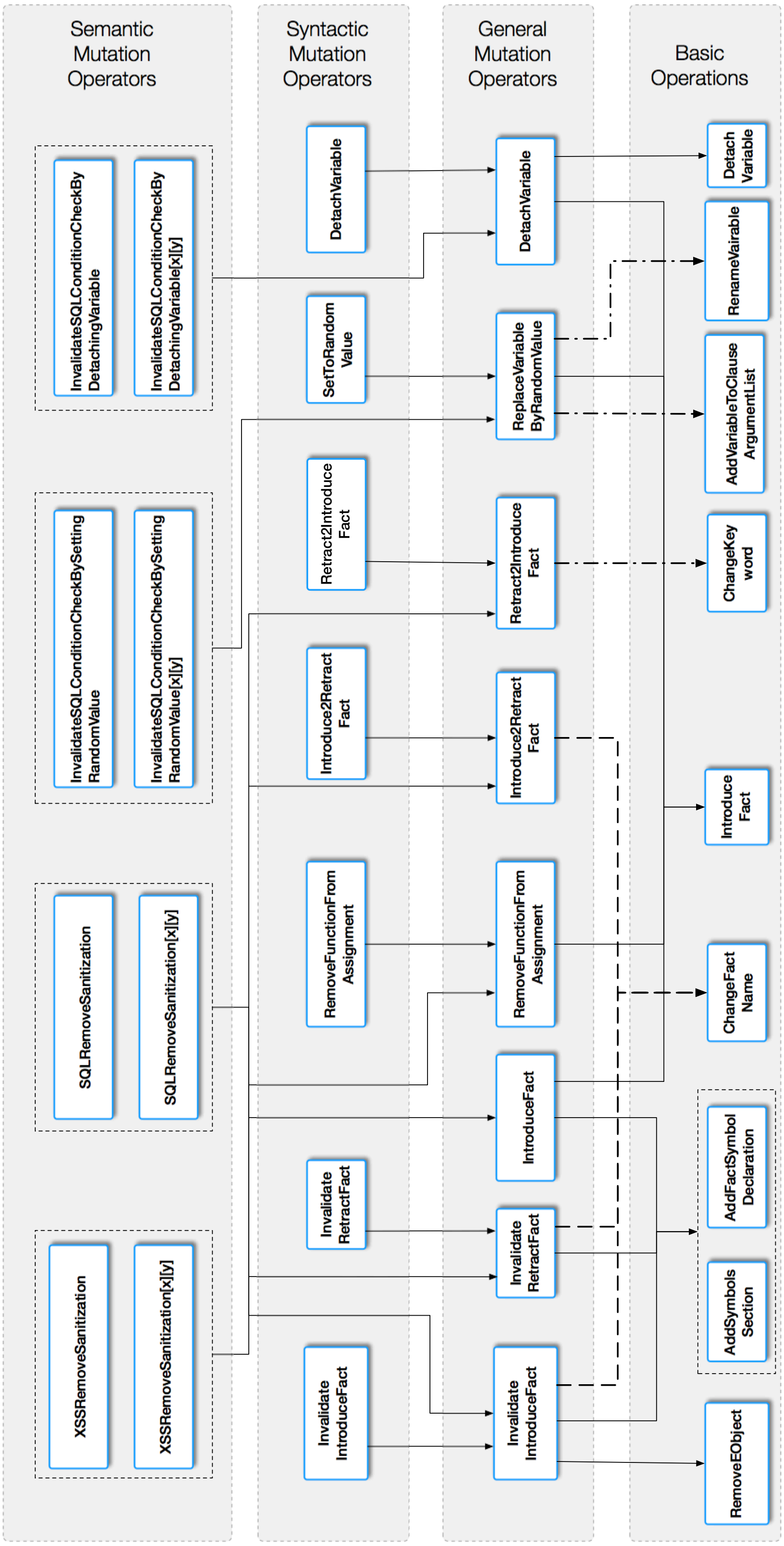

Semantic Mutation Operators
Syntactic Mutation Operators
General Mutation Operators
Basic Operations
InvalidateSQLConditionCheckBy DetachingVariable
InvalidateSQLConditionCheckBy DetachingVariable[x][y]
InvalidateSQLConditionCheckBySetting RandomValue
InvalidateSQLConditionCheckBySetting RandomValue[x][y]
SQLRemoveSanitization
SQLRemoveSanitization[x][y]
XSSRemoveSanitization
XSSRemoveSanitization[x][y]
DetachVariable
SetToRandom Value
Retract2Introduce Fact
Introduce2Retract Fact
RemoveFunctionFrom Assignment
Invalidate RetractFact
Invalidate IntroduceFact
DetachVariable
ReplaceVariable ByRandomValue
Retract2Introduce Fact
Introduce2Retract Fact
RemoveFunctionFrom Assignment
IntroduceFact
Invalidate RetractFact
Invalidate IntroduceFact
Detach Variable
RenameVairable
AddVariableToClause ArgumentList
ChangeKey word
Introduce Fact
ChangeFact Name
AddFactSymbol Declaration
AddSymbols Section
RemoveEObject

Figure 2.7.: Mutation Operators Overview

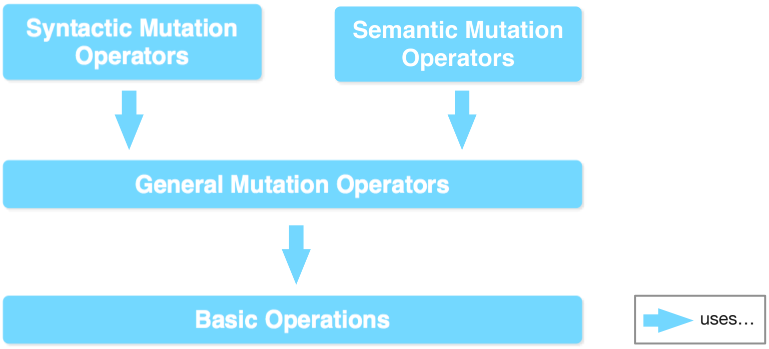

Syntactic Mutation Operators
Semantic Mutation Operators
General Mutation Operators
Basic Operations
uses...

Figure 2.8.: Mutation Operators Hierarchy

m at semantically annotated mutation candidates for m. After collecting all mutation candidates, the configuration of the `Syntactic` and `Semantic` mutation operators determines how many candidates are mutated in the same mutated model. E.g., for first-order mutation operators, only one mutation candidate is mutated per mutated model, whereas for a higher-order mutation operators several candidates are mutated. The set of `General Mutation Operators` provides the required mutation operations that the `Syntactic` and `Semantic` mutation operators perform. Since the General Mutation Operators are composed of several basic operations, the Basic Operations provide the operations from which the General Mutation Operators are constructed. Basic Operations perform a single, atomic operation on the specification. They are simple and easy to understand. One special characteristics is that upon applying such Basic Operations to a specification, the mutated specification might be type incorrect. E.g., if we add a new fact statement in the body of an agent, the mutated model is type incorrect since ASLan++ requires that every used fact has to be declared. Therefore, an additional Basic Operation to add a fact declaration has to be applied as well. This characteristic is different for the General Mutation Operators. If such a mutation operator is applied to an input specification, the mutated specification is guaranteed to be type correct.

To discuss the different mutation operators, we follow a top-down approach. We start the discussion with source code level vulnerabilities and their corresponding Semantic Mutation Operators (Section 2.6.1). For each Semantic Mutation Operator we discuss which lower level operations they require. They themselves are then discussed in the following sections. Therefore, General Mutation Operators are discussed in Section 2.6.4, and Basic Operations in Section 2.6.5. Since Syntactic Mutation Operators are used for the evaluation only, we postpone that discussion to Section 4.5.1.

2.6.1. Semantic Mutation Operators: Concept and Example Model

In this section we describe the set of Semantic Mutation Operators that SPaCiTE supports. Semantic Mutation Operators capture source code vulnerabilities and represent them at the abstract ASLan++ level. A Semantic Mutation Operator at the model level is an abstraction of vulnerabilities and abstracts away irrelevant low-level implementation details. Since in such a situation a 1:1 mapping of a Semantic Mutation Operator and a specific source code vulnerability is in general not possible, a Semantic Mutation Operator represents a specific class (commonalities) of vulnerabilities. Mutation operators in general are functions that take a formal specification and output a set of mutated specifications. Let S be the set of all ASLan++ specifications, and M the set of all mutation operators. Therefore, a mutation operator is a function $m \in M : S \rightarrow 2^S$. We call these operators ‘Semantic Mutation Operators’ since each mutation operator is based on at least one intuition what can go wrong at the implementation level. Arbitrary mutations without any understanding or relation to source code vulnerabilities are not considered as Semantic Mutation Operators. Semantic Mutation Operators are based on semantic annotations. Such annotations are manually inserted by the Security Analyst and are used to provide semantics to model level statements in terms of functionality, as well as the technology used to implement the functionality. An annotation always starts with `%@Semantics` and contains a list of semantics keywords. E.g., the annotation `%@Semantics[SQL, Sanitize]` contains the two semantics keywords `SQL` and `Sanitize`. The annotations used in our models consist of up to three semantics keywords. The first one is from the set $HTML, SQL, JAVASCRIPT$, the second optional keyword is $Sanitize$, and the third one from the set $Input, Internal, Implicit(X)$. The semantics of these keywords is explained later when they are used. An annotation can be a single expression annotation (see Line 17 in Listing 2.8) or a block annotation (see annotation at Line 8 in Listing 2.8). A single line annotation just annotates the next expression, whereas a block annotation annotates all expressions between *`%@Semantics[...]`*$\{$ and *`%@`*$\}$.

In addition to the injection of a vulnerability, mutation operators add facts to keep track of the changes they perform. This is crucial because the mutated elements usually appear in combination with variables. If the mutated model generates an AAT, the value of the mutated element plays a crucial rule. In addition, the value of the variable is determined during model checking. Such information is needed when a Test Execution Engine (TEE) generates executable test cases out of AATs. To keep track of such values, all mutation operators described in this thesis make use of ASLan++ facts. Therefore, we call such facts ‘*value-tracking facts*’.

For each Semantic Mutation Operator, we first discuss the corresponding source code level vulnerabilities, that motivate the Semantic Mutation Operator. The considered vulnerabilities themselves (SQL vulnerabilities in Section 2.6.2 and XSS vulnerabilities in Section 2.6.3) are motivated by several surveys that claim, that SQL and XSS belong to the most reported issues in web applications. They are only a subset of security related issues for web applications but a subset that is relevant in practice [13, 21].

To illustrate the Semantic Mutation Operators and their corresponding annotations, we introduce a simple and partial example model given in Listing 2.8. The model is not complete and irrelevant parts are missing. The model describes a web application that allows users to *login*, *store*, and *view* profiles like Facebook or Google+ profiles. We discuss these three different methods and show how they are modeled, annotated, and mutated.

Listing 2.8: ASLan++ Model of Simple Web Application to Store Profiles

```
1 symbols
2   Username, Password: text ;
3   sanitizeTextSQL( text ): fact ;
```

```
 4   credentials( text, text ): fact ;
 5     ...
 6 select {
 7  on( ?Client *->* Actor: login( ?Username, ?Password ) &
 8     %@Semantics[ SQL ] {
 9       credentials( ?Username, ?Password, ?AdditionalData )
10     %@}
11     & additionalCheck( ?AdditionalData ) :{
12       authenticated( Client ) ;
13     }
14  }
15  on( ?Client *->* Actor: store( ?A, ?D ) & authenticated( ?Client ) ):{
16 
17   %@Semantics[SQL, Sanitize, Input]
18   sanitizeAgentSQL( A ) ;
19 
20   %@Semantics[SQL, Sanitize, Input]
21   sanitizeTextSQL( D ) ;
22 
23   %@Semantics[HTML, Sanitize, Input]
24   sanitizeTextHTML( D ) ;
25 
26   D2 := new copy of D
27   retract stored( A, ? ) ;
28   stored( A, D2 ) ;
29 
30   %@Semantics[SQL, Sanitize, Implicit(D)]
31   sanitizeTextSQL( D2 ) ;
32 
33   %@Semantics[HTML, Sanitize, Implicit(D)]
34   sanitizeTextHTML( D2 ) ;
35 }
36  on( ?Client *->* Actor: view( ?A ) ):{
37    %@Semantics[SQL, Sanitize, Input]
38    sanitizeAgentSQL( A ) ;
39 
40    select{ on(stored( Agent, ?D ) ):{} }
41 
42    %@Semantics[HTML, Sanitize, Internal]
43    sanitizeTextHTML( D ) ;
44 
45    Actor *->* Client : response( D ) ;
46  }
47 }
```

Login. To login, the user has to specify his username (parameter `Username`) and password (parameter `Password`) (Line 7). The `login` method checks if the tuple `(Username, Password)` is stored in the credential database. This is modeled using fact of the form `credentials(u,p)` (Line 9). If the fact `credentials(u,p)` is evaluated to true for a username u and a password p, that means that the username u and the password p are registered in the credential database (Line 9). To get access to the credentials in the database, the credential check is implemented with an SQL query. Therefore, we annotate the `credentials` fact using the annotation keywords `SQL` (see Line 8). Using SQL for checking user credentials is an implementation decision. Many other alternatives are available. If e.g., a third-party login procedure is used that the Security Analyst does not have the permission to test for SQL injections, or the third-party solution does not even depend on SQL, the `credentials` check is not annotated and therefore, SPaCiTE will not generate corresponding test cases. This provides a very dynamic and easy way to exclude certain parts of the SUV from being tested. If the user has provided the correct credentials, we introduce the fact `authenticated` in Line 12 to authenticate a user. This statement is not annotated since the framework to implement a web application usually provides this functionality and performs it automatically.

Store. The method to store a profile takes two parameters: a user ID A of an agent whose profile should be stored/modified, and the data `D` that represents the data to be stored in the profile. Such a message sent by agent $Client$ is only accepted by the server if $Client$ is authenticated (Line 15). If he is, the server sanitizes the user-provided user ID A against SQL (Line 18) because A is used in an SQL query to store/update the profile (Line 28). We annotate Line 18 with the semantics keywords `SQL`, `Sanitize`, and `Input` since the value is directly provided by the client in this message handler. Furthermore, the data to be stored is sanitized against SQL and HTML since the data is part of an SQL query and data is later supposed to be embedded into HTML and displayed in the browser later (Lines 21 and 24). We represent storing the data `D` in profile of agent `A` by using facts. We define the fact `stored(a,d)` with the meaning that the profile of agent `a` stores data element `d`. To delete the data in the profile of agent `a`, we simply retract the fact `stored(a,d)`. Therefore, the `store` method first deletes a potentially already existing profile in Line 27 and stores the new profile afterwards in Line 28. Depending on the concrete specific sanitization function that is used, the stored profile data D_2 in the database *is* or *is not* sanitized against SQL and HTML. For the positive case, where D_2 *is* sanitized as well, we add the following semantics keywords to the fact that sanitizes D_2: `SQL`, `Sanitize`, and `Implicit(D)` (see Line 31). The `Implicit` annotation expresses that the sanitization property of D_2 is inherited from data `D`. If the database value also keeps the HTML sanitization, we add the semantics keywords `HTML`, `Sanitize`, and `Implicit(D)` (Line 34). In the negative case, where the data in the database does not inherit the sanitization, we simply omit Lines 31 and 34, including their annotations.

View. Finally, the message `view` models the functionality to request and display a profile. This method takes the agent whose profile is requested as a single parameter `A`. Since this data is user-provided and used in an SQL query to retrieve the profile, the input is sanitized against SQL (Line 38). We annotate that statement with the semantics keywords `SQL`, `Sanitize`, and `Input`. This SQL query is modeled as a `selectOn` statement where the data parameter `D` is appended to a question mark (Line 40). This means that the model checker tries to find a data element for `D` of type `type_of(D)` so that the fact can be evaluated to `true`. Before the retrieved profile data `D` is sent back to the agent that requested the profile, D is sanitized against HTML in Line 43. This is required since the profile data is supposed to be rendered in the browser (Line 45). Therefore, we annotate Line 43 with the semantics keywords `HTML`, `Sanitize`, and `Internal` since the data is not directly provided by the user.

2.6.2. SQL Vulnerabilities and Their Semantic Mutation Operators

In this section we propose Semantic Mutation Operators that represent SQL vulnerabilities. SQL vulnerabilities both occur and are exploited at the server-side. Since Semantic Mutation Operators are motivated by underlying source code vulnerabilities, we present typical vulnerabilities and then discuss corresponding mutation operators.

Those parts of the formal specification that are annotated with the semantic keywords `SQL` and `Sanitize` represent functionality that sanitizes user-provided input against SQL. Security-related vulnerabilities in this context are of different types — either the sanitization functionality is completely missing, or the sanitization functionality is present but faulty. A faulty sanitization function is effective for some inputs, whereas it is not for others. In the following, we list some real world vulnerabilities, that have been published at CVE websites [11].

Vulnerability: Completely Missing any Sanitization Against SQL

The vulnerability example in Listing 2.9 is based on the example provided by the PHP documentation [16] but slightly adapted for simplicity. The code listing is given in diff format, that means that the lines starting with the minus sign (-) are the vulnerable parts that need to be fixed. The lines starting with the plus sign (+) are the lines that fix the bug. The malicious version of the given code (the (-) lines instead of the (+) lines) allows an attacker to provide a malicious payload for the variable `name` so that it affects the semantics of the SQL query.

Listing 2.9: Missing MySQL Real Escape String

```
 1 <?php
 2 // create connection
 3 $link = mysql_connect('mysql_host', 'mysql_user', 'mysql_password')
 4     OR die(mysql_error());
 5
 6 // create query
 7 - $query=sprintf("SELECT * FROM profiles WHERE name='%s'",
 8 -     $_POST['name']);
 9
10 + $query=sprintf("SELECT * FROM profiles WHERE name='%s'",
11 +     mysql_real_escape_string($_POST['name']));
12
13 mysql_query($query);
14 ?>
```

To illustrate the danger of the missing user-provided input sanitization, we show two malicious payloads to modify the semantics of the SQL query.

- **Changing the Condition:** The original query (Lines 7 to 8) is supposed to return profiles which match in terms of the name with the user-provided input value. If the content of the variable `name` is not sanitized, an attacker can provide `’ OR ’1=’1` as input. The complete SQL query then looks as follows: `SELECT * FROM profiles WHERE name=’’ OR ’1=’1’`. This input changes the semantics of the query so that every profile is returned, independent of the attribute `name`. Thinking of the example web application in Listing 2.8 this kind of issue might have the following consequences. When the web application is checking the password using an SQL query, the password check might be manipulated so that the check is passed although it should fail. In addition, when the web application uses SQL queries for storing or retrieving profiles, the corresponding semantics of the queries might be changed so that a different profile is retrieved or the data is stored in a different profile than expected.

- **Appending New SQL Command:** The attacker cannot only modify the existing query, but append an additional, arbitrary SQL queries as well. E.g., the resulting query after providing the malicious payload `'; DROP * FROM profiles; --` is as follows: `SELECT * FROM profiles WHERE name=''; DROP * FROM profiles; --'`. This query not only performs the original `SELECT` query, but also deletes all profiles. Due to possible subsequent SQL query fragments, the malicious payload finishes with the comment out symbols (`--`) to increase the possibility of ending up with a syntactically correct SQL query.

Vulnerability: Using the Wrong Sanitization Method in the Betster Project

An example of a missing SQL sanitization was present in the Betster project [9]. The project web page describes the software as follows:

Betster is an OpenSource Software to create a online bet office based on PHP and MySQL. The system works with variable Quotes and a totalisator. You bet not for money but with credits. The initial amount of credits which a user gets at the registration you can choose. The admin can manage the bets, he is the bookmaker of the bet office, he manages the categories and administrate the users.

For this project, an SQL injection vulnerability was found and published [3, 8]. Conceptually, the client queries for a profile by sending an ID to the `showprofile.php` webpage. The value is sanitized against XSS attacks using the PHP function `htmlspecialchars`. The sanitized value is then forwarded to the `getUserById` method in the class `class/DbMapper.class.php`, where it is used to construct an SQL query. Since the user-provided parameter (ID) is not sanitized against SQL, it can be used for an SQL injection. The corresponding code example is shown in Listing 2.10.

CVE ID	CVE-2015-2237 `http://www.cvedetails.com/cve/CVE-2015-2237/`
Vulnerability Type	Execute Code Sql Injection
Description	Multiple SQL injection vulnerabilities in Betster (aka PHP Betoffice) 1.0.4 allow remote attackers to execute arbitrary SQL commands via the id parameter to (1) showprofile.php or (2) categoryedit.php or (3) username parameter in a login to index.php.
Location	showprofile.php, categoryedit.php, index.php

Listing 2.10: CVE-2015-2237

```
in file: showprofile.php
-------
  $id = htmlspecialchars($_GET['id']);
  $xuser = $db_mapper->getUserById($id);

in file: class/DbMapper.class.php
-------
  function getUserById($id){
    $query = "SELECT id,username,email,firstname,lastname,balance,status
      FROM user WHERE id = '".$id."'";
    $result = mysql_query ($query)  or error_catcher();
```

Vulnerability: Faulty Sanitization

An easy to understand but still complex example of a faulty sanitization method is provided in a paper published by Fu et al. [108]. The purpose of this sanitization method is to sanitize every user-provided input. Therefore, it is invoked every time the server receives data from the user, that is intended to be used part of an SQL query. The implementation of the sanitization method `sanSQL` is given in Listing 2.11. In a first step, the sanitization method checks if the user-provided string contains either the comment string (`--`) (Line 4), the SQL keyword `OR` (Line 5), or the SQL keyword `drop` (Line 6). If the user-provided input contains at least one of the above mentioned options, an application that uses the sanitization method given in Listing 2.11 is rejecting the input and it can not be used for an SQL injection on that application. Therefore, let's consider an input that does not fulfill the above mentioned criteria. It passes the check and therefore, every occurrence of a single quote is replaced by two single quotes (`''`) (Line 11). While a single quote is used to start and terminate strings in SQL, a double single quote is needed to represent a single quote in the string. Therefore, a user-provided single quote cannot be used to terminate the SQL string. In turn, terminating an SQL string is necessary to provide SQL keywords that the SQL server interprets. Finally, to reduce the risk of a successful SQL injection, the user-provided input is restricted to a maximal length of 16 chars in Line 12.

Fu et al. [108] provide the following SQL query for which the above discussed SQL sanitization method is used

```
query = "SELECT uname, pass FROM users WHERE
  uname='" + sanSQL(username) + "' AND
  pass='" + sanSQL(sPwd) + "'"
```

Although this specific sanitization method is effective for certain SQL injections like `admin' --` for the username and the empty string for the password, it is not effective e.g., for the username `123456789012345'` and the malicious password `␣O/**/R uname<>'`[6]

[6] ␣represents a space

because of the following argumentation. Providing these two values, the implementation constructs the following SQL query:

```
query = "SELECT uname, pass FROM users WHERE
  uname='123456789012345'' AND pass=' OR uname<>''"
```

Because of the escaped single quote in the username, the `where` condition checks if the username is equal to `123456789012345’’ AND pass=` or if the username is different from the empty string. The probability that a username has length ≥ 1 is quite high and therefore, the probability of being able to exploit the `sanSQL` method with the above malicious input as well.

Listing 2.11: Faulty SQL Sanitization

```
String sanSQL(String strInput) {
  //1. SQL keyword search
  if(
    strInput.IndexOf("--")!=-1
    || strInput.IndexOf("OR")!=-1
    || strInput.IndexOf("drop")!=-1
  )
     throw new Exception("Possible SQL Injection);

  //2. massage the data for single quote
  String sOut = strInput.Replace("'","''");
  sOut = sOut.Substring(0,16);
  return sOut;
}
```

2.6.2.1. Semantic Mutation Operator: SQLRemoveSanitization

Since ASLan++ does not provide dedicated language constructs to sanitize user-provided input against SQL, the Security Analyst has some freedom to model that functionality. We focus here on two different ways of modeling it, following the guideline presented in Section 2.3.3.3.

Modeling Sanitization with Facts. If the Security Analyst follows a fact-based approach to represent sanitization functionality at the model level, he models the sanitization of a data value v by introducing the fact similar to `sanitize_sql(v).` If v is sanitized, he introduces the fact, and he retracts the fact if v is not sanitized anymore. Therefore, the same syntactical representation of value v is used before and after the sanitization. To check whether a data value v is sanitized, the fact `sqlSanitized(v)` is evaluated to its boolean value. Since sanitization does not change the syntactical representation of value v, the `SQLRemoveSanitization` Semantic Mutation Operator is invalidating the introduction of the fact `sqlSanitized(v)` by applying the General Mutation Operator `InvalidateIntroduceFact`.

In Section 2.6.4 we have already seen that the dual view of introducing a fact is retracting it. Following this approach, the Security Analyst retracts a fact when the data is *not* sanitized instead of introducing a fact when data *is* sanitized. In this situation, the General Mutation Operator `InvalidateRetractFact` is applied.

Modeling Sanitization with Variable Assignments. If the Security Analyst follows a symbolic function-based approach (see Section 2.3.3.3) the Security Analyst introduces a symbol function similar to `sanitize_sql( T ) : T`. To sanitize a value v of type T, `sanitized()` is applied to v and the complete expression is assigned to a new variable. In a formal specification that makes use of this approach of sanitization, removing the whole assignment is very likely to break the described functionality in the model. Because the sanitized value is assigned to a new variable, it is very likely that other parts of the model operate on that variable from now on. Therefore, the variable assignment should not be eliminated during invalidating the sanitization. Therefore, the `SQLRemoveSanitization` Semantic Mutation Operator makes use of the `RemoveFactFromAssignment` General Mutation Operator. It basically mutates an assignment of the form `V' := sanitized_T(V)` to `V' := V` if type restrictions and the number of parameters allow it (see Section 2.6.4).

As we will see in Section 2.6.4, applying a General Mutation Operator will automatically introduce a general value-tracking fact without vulnerability correlation, since no relation to vulnerabilities or the semantics of the mutation is known. If the Semantic Mutation Operator applies a General Mutation Operator the underlying fault and semantic information is available. This information is therefore used to overwrite the general value-tracking fact with a vulnerability specific one. Thanks to the semantics annotation keywords `SQL`, `Sanitize`, and the type of data (`Input`, `Internal`, and `Implicit`), the General Mutation Operator is configured to add the specific value-tracking fact of the following form: The fact name start with the prefix `removed`, followed by the type of technology involved (`SQL`), the type of data (`Input`, `Internal`, and `Implicit`), and the semantics of the action (`Sanitize`). Since sanitizing can be performed on different types of values, and because ASLan++ requires every fact to be unique in terms of the fact name only, the signature (types) of the fact parameters is appended to the fact name as well. All these pieces of information is separated by the underscore symbol (_). To demonstrate the mutation operator we apply it to the annotated Lines 18, 21 and 38 in Listing 2.8. Listing 2.12 shows the effect of applying the mutation operator on Lines 18 and 38, and Listing 2.13 shows the mutation on Line 21.

Listing 2.12: Applying SQLRemoveSanitization

```
    %@Semantics[SQL, Sanitize, Input]
-    sqlSanitizeAgentSQL( A );
+    removed_SQL_Input_Sanitize_agent( A );
```

Listing 2.13: Applying SQLRemoveSanitization

```
    %@Semantics[SQL, Sanitize, Input]
-    sanitizeTextSQL( D );
+    removed_SQL_Input_Sanitize_text( D );
```

According to the above description, the `SQLRemoveSanitization` Semantic Mutation Operator is also applicable to the annotated Line 31. Since that Line is annotated with the semantics keyword `Implicit`, we postpone the discussion of that line to Section 2.6.2.2.

SQLRemoveSanitization in 10 Seconds

The Semantic Mutation Operator looks for annotations that contain both the labels `SQL` and `Sanitize`. If the formal specification contains such an annotation, the mutation operator performs the following actions, depending on the annotated expression.

- If the sanitization is modeled by introducing a fact, the General Mutation Operator `InvalidateIntroduceFact` is applied.
- If the sanitization is modeled by retracting a fact, the General Mutation Operator `InvalidateRetractFact` is applied.
- If the sanitization is modeled by applying a symbolic function to a variable and assign the application to a variable, the General Mutation Operator `RemoveFunctionFromAssignment` is applied.

In all three situation, the mutation operators add the fact `removed_SQL_[type of value]_Sanitized_[parameter signature](value)` to the specification.

Multiple Applicability of the SQLRemoveSanitization Semantic Mutation Operator Depending on the modeling and the complexity of vulnerabilities that are considered, generating mutated models by applying the Semantic Mutation Operators only once is potentially not enough. To let the model checker find AAT related to more complex vulnerabilities, we need to apply the Semantic Mutation Operators multiple times. Therefore, the above described mutation operators `SQLRemoveSanitization` collects all semantic annotations that contain the semantics keywords `SQL` and `Sanitize` and generates mutated models with all possible combinations. In practice this easily leads to an enormous number of mutated models that are impractical to model check and execute. Therefore, we introduce the Semantic Mutation Operator `SQLRemoveSanitizationUpToLimit` with the additional capability to limit the minimal and the maximal number of sanitizing annotations that are mutated for the same mutated model. If at least x but at most y mutations dedicated to SQL sanitization should be applied to mutate a model, the Semantic Mutation Operator `SQLRemoveSanitizationUpToLimit[x][y]` is used.

2.6.2.2. Vulnerability: Inappropriate SQL Sanitization in the Context of 2nd Order SQL Injections

The previously introduced `SQLRemoveSanitization` Semantic Mutation Operator is very powerful and configurable by using different semantics keywords in the annotation[7]. In this section we show that the same Semantic Mutation Operator is also useful in the context of 2nd order SQL injections. To demonstrate this vulnerability we extend the example application given in Listing 2.8 with the feature to *create user accounts*. Since users sometimes forget their password, the web application offers in addition a registered user the functionality to change its own password. For the 2nd order SQL vulnerability, it is a crucial requirement that both functionalities (creating user accounts, changing passwords) make use of SQL sanitization and SQL statements respectively. Listing 2.14 shows the extension at the model level. Because the web application now offers two new features, it accepts two new requests, modeled with `selectOn` statements.

Listing 2.14: Message Handler For The `createNewUser` Message

```
on( ?Us *->* Actor : createNewUser(
                              inst_username_t(?UsernameMessage,?UNounceMessage),
                              inst_password_t(?PasswordMessage,?PNounceMessage)
                     ) &
                     %@Semantics[ SQL ] {
                       !credentials( inst_username_t(?UsernameMessage,?)
                     %@}
) : {
  UsernameMsg := inst_username_t(UsernameMessage,UNounceMessage);
  PasswordMsg := inst_password_t(PasswordMessage,PNounceMessage);

  %@Semantics[ SQL, Sanitize, Input ]
  UsernameMsgSan := sanitize_SQL_username_t(UsernameMsg ) ;

  %@Semantics[ SQL, Sanitize, Input ]
  PasswordMsgSan := sanitize_SQL_password_t(PasswordMsg) ;

  Nounce_toStore := fresh();
  UsernameDB := inst_username_t(UsernameMessage, Nounce_toStore);
  PasswordDB := inst_password_t(PasswordMessage, Nounce_toStore);

  %@Semantics[ SQL, Sanitize, Implicit(UsernameMsg) ]
  UsernameDBSan := sanitize_SQL_username_t(UsernameDB) ;

  %@Semantics[ SQL, Sanitize, Implicit(PasswordMsg) ]
  PasswordDBSan := sanitize_SQL_password_t(PasswordDB) ;

  credentials(UsernameDBSan, PasswordDBSan);
  writtenToDB(UsernameDB,UsernameDBSan);
  writtenToDB(PasswordDB,PasswordDBSan);
}


```

[7] %@Semantics[keyword1, keyword2, . . .] is a semantic annotation consisting of the two semantics keywords `keyword1` and `keyword2`

```
34 on( ?Us *->* Actor : changePassword(
35   inst_username_t( ?UsernameMessage, ?Nonce1 ),
36   inst_username_t( ?OldPwdMessage, ?Nonce2 ),
37   inst_username_t( ?NewPwdMessage, ?Nonce3 )
38   ) : {
39     UsernameMsg := inst_username_t(UsernameMessage,Nonce1);
40     OldPwdMsg := inst_password_t(OldPwdMessage,Nonce2);
41     NewPwdMsg := inst_password_t(NewPwdMessage,Nonce3);
42 
43     %@Semantics[ SQL, Sanitize, Input ]
44     UsernameMsgSan := sanitize_SQL_username_t(UsernameMsg) ;
45 
46     %@Semantics[ SQL, Sanitize, Input ]
47     OldPwdMsgSan := sanitize_SQL_password_t(OldPwdMsg) ;
48 
49     %@Semantics[ SQL, Sanitize, Input ]
50     NewPwdMsgSan := sanitize_SQL_password_t(NewPwdMsg) ;
51 
52 
53     select{ on( credentials( inst_username_t(UsernameMessage,?Nonce4),
54                              inst_password_t(?PasswordDB,?Nonce5))):{} }
55     readFromDb(UsernameMsgSan, inst_username_t(UsernameMessage,Nonce4));
56 
57     if( OldPwdMessage == PasswordDB ) {
58       retract credentials( inst_username_t(UsernameMessage,?),?);
59       credentials( inst_username_t(UsernameMessage,Nonce4),NewPwdMsgSan );
60     }
61 }
```

To create a new user, the web application accepts the message `createNewUser`. It requires two arguments — the username and the password. Both arguments are stored in variables in Lines 9 and 10 and are sanitized in Lines 13 and 16. Because both user-provided inputs are stored in the database (Lines 28 to 30), the Security Analyst first creates two unique instances of these two values (Lines 18 to 20).

To change the password, the web application accepts the message `changePassword` which requires three arguments — the username, the old and the new password. After the user-provided inputs are stored in variables in Lines 39 to 41, they are sanitized against SQL in Lines 44 to 50. In Lines 53 to 55 the web application gets the corresponding record from the database, based on the provided username. If the password retrieved from the database matches with the old password provided as argument of the `changePassword` message, the new password is stored in the database in Line 59.

A possible implementation of the methods `createNewUser` and `changePassword` is given in Listing 2.15. The `createNewUser` request handler (Lines 2 to 14 in Listing 2.15) accepts two parameters `username` (Line 3) and `password` (Line 4). The content of both parameters is sanitized using the implementation level method `sanitizeImpl` in Line 6 and Line 7. The method represents any sanitization method that is motivated by the special characters in SQL strings [31] and sanitizes them by escaping. E.g., one could use the PHP method `mysql_real_escape_string` [16] or the method `escape` provided at stackoverflow.com [30]. In the SQL context, characters like single `(’)` and double `(")` quotes and the like have a special meaning. The SQL server recognizes and interprets them. If user-provided data and SQL control data is mixed — by embedding user-provided data into a predefined string SQL query skeleton (see Line 11) —, special characters as part of the user-provided data may change the semantics of the SQL query. Therefore, the `sanitizeImpl` method escapes special characters by prepending the backslash symbol so that they are not interpreted anymore (see Figure 2.9#❶). It is important to note that the `sanitizeImpl` method encodes/escapes rather than removes the special characters. Because of that, the stored value in the database still contains all information including the special values. As an example, let's assume an attacker provides the string `admin' or '1'='1` as a username. Since this value gets sanitized against SQL, it gets transformed to `admin\'or \'1\'=\'1` and does not change the semantics of the `insert` query since the quotes are not interpreted. At the same time, sanitizing has another effect that is crucial in this context. In order to store e.g., a quote in the database, it needs to be escaped. Therefore, the sanitization not only makes sure that special values do not change the semantics of the query, but also that the special value without being escaped is stored in the database. In our example, the sanitized string `admin\' or \'1\'=\'1` is finally stored as `admin' or '1'='1` in the database (see Figure 2.9#❷ and ❸).

Listing 2.15: Example: Creating User Account and Updating Password

```
1  // createNewUser request handler
2  public String execute(HttpServletRequest req, HttpServletResponse resp){
3      String username = req.getParameter("username");
4      String password = req.getParameter("password");
5
6      String usr_escaped = sanitizeImpl( username );
7      String pwd_escaped = sanitizeImpl( password );
8
9      Connection con =     ...;
10     Statement st = con.createStatement();
11     st.executeUpdate(
12         "insert into credentials( usr_field, pwd_field ) " +
13         "values('" + usr_escaped + "', '" + pwd_escaped + "')");
14 }
15
16 // changePassword request handler
17 public String execute(HttpServletRequest req, HttpServletResponse resp) {
18   String username = request.getParameter( "username" );
19   String old_pwd = request.getParameter( "old_password" );
20   String new_pwd = request.getParameter( "new_password" );
21
22   String usr_escaped = sanitizeImpl( username );
23   String new_pwd_escaped = sanitizeImpl( new_pwd );
24   String old_pwd_escaped = sanitizeImpl( old_pwd );
25
26   Connection con = JDBCUtil.getConnection();
27   Statement st = con.createStatement();
28
29   ResultSet rs = st.executeQuery(
30     "select * from users where name='" + usr_escaped + "'" );
31   if (rs.next()) {
32     String usr_db = rs.getString( "name" );
33     String old_pwd_db = rs.getString( "pass" );
34     String old_pwd_db_escaped = sanitizeImpl( old_pwd_db );
```

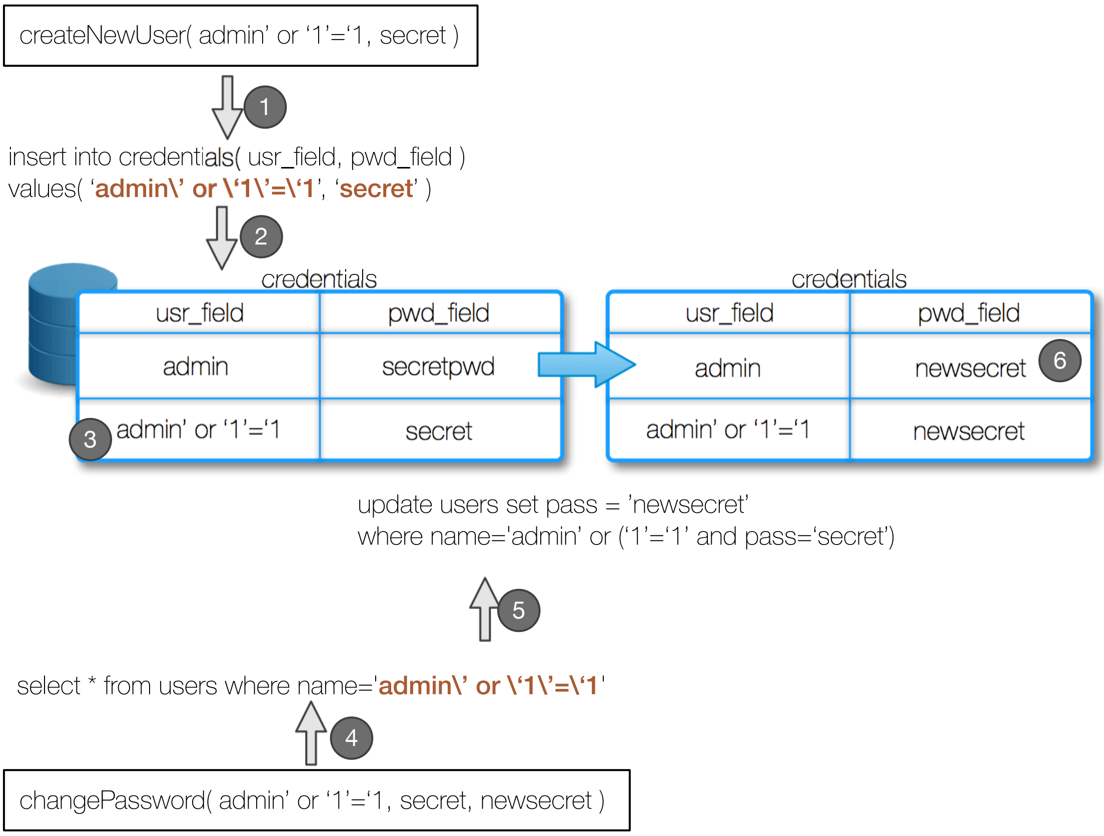

createNewUser(admin' or '1'='1, secret)
1
insert into credentials(usr_field, pwd_field)
values('admin\' or \'1\'=\'1', 'secret')
2
credentials
usr_field
pwd_field
admin
secretpwd
3
admin' or '1'='1
secret
credentials
usr_field
pwd_field
admin
newsecret
6
admin' or '1'='1
newsecret
update users set pass = 'newsecret'
where name='admin' or ('1'='1' and pass='secret')
5
select * from users where name='admin\' or \'1\'=\'1'
4
changePassword(admin' or '1'='1, secret, newsecret)

Figure 2.9.: 2nd Order SQL Injection

```
35
36      if( old_pwd_db_escaped == old_pwd_escaped ) {
37        String query = "update users set pass = '" + new_pwd_escaped + "' " +
38          " where name='" + usr_db + "' and pass='" + old_pwd_db_escaped + "'";
39
40        st.executeUpdate(query);
41        con.close();
42      }
43    }
44  }
```

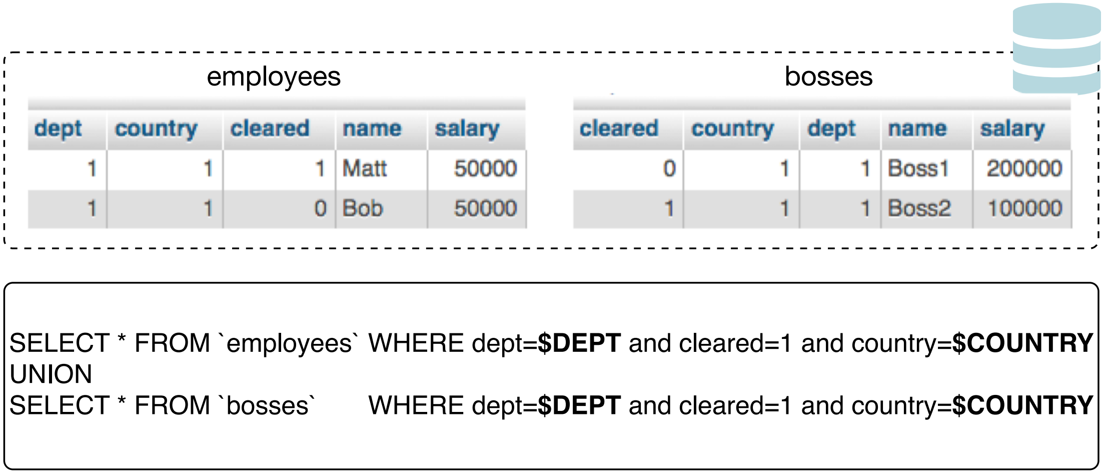
The second functionality allows a user to change his password. For this purpose, the method `ChangePassword` (Lines 17 to 44 in Listing 2.15) requires three parameters; the username (Line 18), the old password (Line 19), and the new password (Line 20). As shown in lines 22 to 24, all three inputs to this method invocation are sanitized so that special characters in these parameters cannot change the semantics of the SQL query in Line 30. The query is used in order to check, if the user is allowed to change the password. Therefore, the username and the old password are retrieved from the database in Lines 32 and 33. Important for the 2nd order SQL injection is that the software developer correctly sanitized all user-provided method parameters during the `createNewUser` and the `ChangePassword` functionality. The username provided to the `ChangePassword` method needs to be sanitized to retrieve the correct row from the database (see Figure 2.9#❹) and therefore to pass the password check. The vulnerability that allows the 2nd order SQL injection is present because the retrieved username from the database is not sanitized against SQL anymore (Line 32) and because the modeler decided to use the non-sanitized username retrieved from the database for changing the password instead of the sanitized username provided at the `changePassword` method invocation. The database value still contains all special characters during the construction of the SQL query in Line 38 and therefore, it can change the semantics of the query (see Figure 2.9#❺). In our concrete example, the username is used for the condition to update the password. Since it is not sanitized, the original condition `where name='...'  and pass='...'` changes to `where name='admin' or '1'='1' and pass=...`. Due to the precedence order in SQL[8], the new password is changed for any user with user name `admin` (independent of the password), and for users where the password matches the old password. Therefore, this attack allows to change the password of the user `admin` without knowing the password (see Figure 2.9#❻).

2.6.2.3. Semantic Mutation Operator for 2nd Order SQL Vulnerabilities

Due to the generality and the possibility to configure the `SQLRemoveSanitization` Semantic Mutation Operator, we do not need a new Semantic Mutation Operator but `SQLRemoveSanitization` can be re-used to represent the vulnerabilities related to 2nd order SQL injections. In Figure 2.9#❶, user-provided data is sanitized against SQL. The corresponding blocks in the formal specification (Lines 13 and 16 in Listing 2.14) are annotated with the semantics keywords `SQL`, `Sanitize`, and `Input`. The same annotation is used when the provided username and the two passwords of the `changePassword` message are sanitized (Lines 44 to 50). To start with a correct model that does not suffer from a 2nd order vulnerability, the Security Analyst is using a sanitization method with the property that also the stored data in the database is sanitized. This semantics is given in Line 23. The fact that the value stored in the database 'stays' sanitized is not because a separate sanitization method is applied to that value, but rather because it is a property of the sanitization method applied to the user-provided input value at Line 13. Therefore, the sanitization block at Line 23 is annotated with the semantics keywords `SQL`, `Sanitize`, and `Implicit(UsernameMsg)` since the value `UsernameDB` 'inherits' the sanitization property from the value `UsernameMsg`. Applying the `SQLRemoveSanitization` Semantic Mutation Operator to the statement at Line 23 represents the vulnerability, that the data value stored in the database is not sanitized against SQL anymore.

Applying the `SQLRemoveSanitization` Semantic Mutation Operator to Lines 13, 16 and 44 leads to the same mutation as described e.g., in Listing 2.12 or Listing 2.13. To represent 2nd order SQL vulnerabilities, the same `SQLRemoveSanitization` Semantic Mutation Operator is applied to Lines 23 and 26 as well. Since these two annotated Line are annotated with the semantics keyword `Implicit`, the mutation looks as follows (Listings 2.16 and 2.17):

[8] `and` binds stronger than `or`



employees

dept	country	cleared	name	salary
1	1	1	Matt	50000
1	1	0	Bob	50000

bosses

cleared	country	dept	name	salary
0	1	1	Boss1	200000
1	1	1	Boss2	100000

```
SELECT * FROM `employees` WHERE dept=$DEPT and cleared=1 and country=$COUNTRY
UNION
SELECT * FROM `bosses`    WHERE dept=$DEPT and cleared=1 and country=$COUNTRY
```

Figure 2.10.: Split SQL Vulnerability Example

Listing 2.16: Applying SQLRemoveSanitization

```
 %@Semantics[ SQL, Sanitize, Implicit(UsernameMsg) ]
-    UsernameDBSan := sanitize_SQL_username_t(UsernameDB) ;
+    UsernameDBSan := UsernameDB ;
+    removed_SQL_Implicit_Sanitize_username_t( UsernameMsg );
```

Listing 2.17: Applying SQLRemoveSanitization

```
%@Semantics[ SQL, Sanitize, Implicit(PasswordMsg) ]
-    PasswordDBSan := sanitize_SQL_password_t(PasswordDB) ;
+    PasswordDBSan := PasswordDB ;
+    removed_SQL_Implicit_Sanitize_password_t( D );
```

2.6.2.4. Vulnerability Leading to Split SQL Injection.

For the discussion of the split SQL injection, we do not extend the web application anymore, since it would not introduce any new concepts at the model level. Therefore, we focus on the implementation level vulnerabilities only. In our example in Listing 2.8, profiles were represented as a single string due to simplicity. For the discussion of the split SQL injections, we refine these profiles. A profile provides salary information of employees and bosses to the HR department. Data stored in profiles is split into two separate tables. The first table is called ‘employees’ and stores employee’s salaries. The second table is called ‘bosses’ and stores boss’ salaries. The webpage `getSalaries.php` accesses these two tables with the query shown in Figure 2.10. An HR employee can query the salaries of a specific department in a country by providing the department and the country ID. E.g., to get the salaries of people working for department ‘1’ in the country ‘1’, the HR employee sends the query `getSalaries.php=1&COUNTRY=1`. This creates the following SQL query and returns the first record of the table 'employees' and the second record of the table 'bosses':

```
SELECT * FROM `employees` WHERE dept=1 and cleared=1 and country=1
UNION
SELECT * FROM `bosses` WHERE dept=1 and cleared=1 and country=1
```

As Figure 2.10 shows, the tables also contains salaries that are not 'cleared', meaning they are secret and should never be revealed. In particular, let's assume not-cleared employee's salaries are secret, and not-cleared boss' salaries are top secret and therefore should not be accessible by the HR department.

To attack this security property and reveal top-secret salaries, an attacker considers to inject e.g., `1/*` as a department ID and `*/` as the country ID. Providing these two inputs, the query:

```
SELECT * FROM `employees` WHERE dept=1/* and cleared=1 and country=*/
UNION
SELECT * FROM `bosses` WHERE dept=1/* and cleared=1 and country=*/
```

is constructed. Since all text between * and *\ is interpreted as a comment, this split injection leads to the condition:

```
Select * FROM `employees` WHERE dept=1
UNION
Select * from `bosses` WHERE dept=1
```

Therefore, the query returns all salaries of employees and bosses in the department '1', independent if the salaries are cleared or not. By injecting opening and closing comment characters, parts of the query are commented out without losing the second query appended as a 'UNION'. In contrast, injecting the malicious payload `1 --` for the department ID is commenting out everything after the department ID. The query with such an injection returns all salaries of the employees but not the bosses' ones. We want to stress that the above vulnerability is even present if all user-provided inputs are sanitized against SQL with the PHP method `mysql_real_escape_string` because the slash (/) and the star (*) do not get sanitized. This is an example where a SQL sanitization mechanism is present but not effective.

This examples demonstrates that sophisticated SQL attacks might require to inject the malicious payload via different parameters to attack the web application. To find such a vulnerability in a formal specification, the Semantic Mutation Operator `SQLRemoveSaniti`

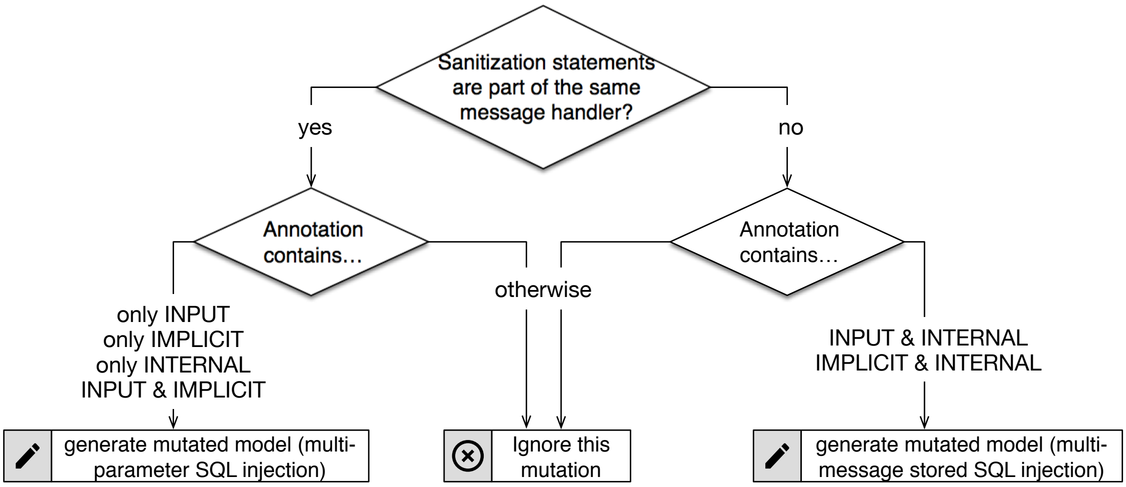

Sanitization statements are part of the same message handler?
yes
no
Annotation contains…
Annotation contains…
only INPUT
only IMPLICIT
only INTERNAL
INPUT & IMPLICIT
otherwise
INPUT & INTERNAL
IMPLICIT & INTERNAL
generate mutated model (multi-parameter SQL injection)
Ignore this mutation
generate mutated model (multi-message stored SQL injection)

Figure 2.11.: Decision Diagram for SQLRemoveSanitization

zation needs to be applied to both the department and the country ID at the same time. Collecting every possible mutation candidate and considering any possible combinations of candidates for a mutated formal specification do not always represent a real-world vulnerability. Not every possible combination makes sense from a semantic point of view. Therefore, the SQLRemoveSanitization mutation operator generates only dedicated combinations. Figure 2.11 presents an overview of the decision diagram. It shows which combinations lead to a useful mutated model. For instance, if an implicit sanitization and an internal sanitization block is mutated, that are not part of the same message handler, a mutated model is generated. This combination leads to a mutated model since it is the typical situation where a 2nd order SQL injection happens. At the other side, if a sanitization operation on an input value and a sanitization operation on an internal value are mutated, there is no well-known vulnerability and no clear understanding how this vulnerability can be exploited. Therefore, such mutated models are not generated.

Unfortunately, SQL queries are not only represented by introducing facts or symbolic functions assigned to a variable, but could also be represented as a condition check. Especially in the context of using model checkers, combining the receipt of a message with a condition check is very popular since it helps to reduce the state-space explosion problem. If the Security Analyst has chosen this way of modeling an SQL query, the above mentioned Semantic Mutation Operator(s) cannot be applied and therefore, also no SQL vulnerability can be injected. Therefore, we define two more Semantic Mutation Operators, called BindVariableToRandomValue and DetachVariableForSQL. They are both applicable, if the SQL query is modeled as a condition check. Please note that the following two Semantic Mutation Operators do not represent additional source code vulnerabilities, but define two additional ways to inject SQL vulnerabilities into the model.

2.6.2.5. Semantic Mutation Operator: DetachVariableForSQL

A formal specification usually contains conditions or guards that are used for permission checks or retrieving matching values from a data store. In our example in Listing 2.8, the server entity only accepts a `username` and a `password` during the login if they are available in the credentials database. At implementation level, this check is implemented with an SQL query. At the model level the Security Analyst models the same functionality as part of a `selectOn` statement by checking the boolean value of the fact `credentials`. Therefore, the Security Analyst annotates the `credentials` check with the semantic keyword `SQL` as shown at Line 9 in Listing 2.8.

Having an SQL vulnerability in this functionality, the username could e.g. be used to bypass the credential checks. This means that the credentials check may return true in many more cases than it actually should. To inject such a vulnerability, the `DetachVariableForSQL` Semantic Mutation Operator is applied on conditions that represent SQL queries. It replaces concrete values with an existence quantifier. In ASLan++, the question mark (?) represents a placeholder for any value of correct type. Therefore, the mutation operator performs the following mutations:

1. The mutation operator choses a question-marked variable that is used in an SQL annotated guard.
2. To determine whether the variable is an input or an internal variable, the mutation operator follows the following heuristics:
   - If the question-marked variable is part of a receiving message, it is considered as an input variable. The question-marked variable in the SQL annotated guard is mutated by the General Mutation Operator `DetachVariable` which in a nutshell, replaces the question-marked variable by a question mark only.
   - If the question-marked variable is not part of a receiving message, it is considered as an internal variable. Such a variable is used to retrieve values from other sources than the message. Therefore, it is ignored by this Semantic Mutation Operator.

Applying the mutation operator to Line 9 in Listing 2.8 will generate three mutated models, as shown in Listings 2.18 to 2.20. In the first one, the variable `Username` in Line 9 is detached. In the second one, the variable `Password` in Line 9 is detached, and in the third one, both variables are detached at the same time. Both of these variables are input variables, since they are parameters of the `login` message. All the other question-marked variables are ignored by the Semantic Mutation Operator. The `AdditionalData` variable is e.g. used to retrieve data that is stored alongside with the username and password in the credentials database. The content of that variable cannot be provided by an input parameter of the `login` message.

Listing 2.18: Applying DetachVariableForSQL

```
on( ?Client -> Actor : login( ?Username, ?Password ) &
  %@Semantics[ SQL ] {
    credentials( ?, ?Password, ?AdditionalData )
  %@} & ...
): {
   maliciousValue_SQL_input_text( Username ) ; ...
```

Listing 2.19: Applying DetachVariableForSQL

```
on( ?Client -> Actor : login( ?Username, ?Password ) &
  %@Semantics[ SQL ] {
    credentials( ?Username, ?, ?AdditionalData )
  %@} & ...
): {
   maliciousValue_SQL_input_text( Password ) ; ...
```

Listing 2.20: Applying DetachVariableForSQL

```
on( ?Client -> Actor : login( ?Username, ?Password ) &
  %@Semantics[ SQL ] {
    credentials( ?, ?, ?AdditionalData )
  %@} & ...
): {
   maliciousValue_SQL_input_text( Password ) ;
   maliciousValue_SQL_input_text( Username ) ;
   ...
```

Detaching a variable by replacing it with a question mark is a very powerful mutation operator. Without any semantic annotation, it is so powerful that it represents a set of vulnerability classes. Therefore, it is important to bind them to a specific vulnerability class. Like for any other mutation operator, we provide the semantics with the semantic keywords in the annotation of the condition. Therefore, by changing the keywords, we are able to bind the DetachVariableForSQL to a different semantics and therefore to a different source code vulnerability. We consider this as a feature of this Semantic Mutation Operator.

DetachVariableForSQL in 10 Seconds

The Semantic Mutation Operator looks for annotations that contain the label `SQL`, and are part of a condition. If such an annotation is found, the mutation operator performs the following actions:

- A question-marked variable in the annotated guard expression is replaced by a question mark only.
- The detached variable is tracked using the special fact `maliciousValue_SQL_{Internal,Input}_[type of variable]( [Variablename] )`.

While `DetachVariableForSQL` considers every possible combination of mutation candidates for generating mutated models, `DetachVariableForSQLUpToLimit[x][y]` creates mutants where the mutation is applied at least x times and at most y times.

2.6.2.6. Semantic Mutation Operator: BindVariableToRandomValue

As discussed in Section 2.6.2.5, the underlying source code vulnerability of the `Detach VariableForSQL` Semantic Mutation Operator is that a malicious user-provided data can be used to bypass the condition check. This kind of mutation is usually effective when the condition occurs in positive form and has to be evaluated to true for violating a security property (Line 9 in Listing 2.8). For the case, where the condition occurs in negative form and has to be evaluated to true to violate a security property (Line 6 in Listing 2.14), applying the `DetachVariableForSQL` Semantic Mutation Operator does not lead to the desired result in most cases.

The model expresses that the check `!database()` is implemented as an SQL query. If the user-provided data is not sanitized correctly against SQL, a malicious input can be used to bypass the database check and allows to create a profile with the same username as an existing profile. If e.g., security-relevant properties are inferred from the username, the newly created profile inherits the security properties as well and therefore, it might violate the security property.

If we would apply the `DetachVariableForSQL` Semantic Mutation Operator to represent this vulnerability, the condition `!database(inst_username_t(?),?)` in Line 6 of Listing 2.14 would always be evaluated to false if at least one username is stored in the database. Since the existence of at least one account is very likely to be true, applying the `DetachVariableForSQL` Semantic Mutation Operator does not inject the SQL vulnerability as desired. Therefore, we need the dual mutation operator of `DetachVariableForSQL` which we call `BindVariableToRandomValue`.

The Semantic Mutation Operator `BindVariableToRandomValue` in the above situation represents the incomplete or missing sanitization check against SQL by replacing the variable ?UsernameMessage in Line 6 by a random value. It does so by declaring a new non-initialized variable of the correct type. The question-marked variable is then substituted by the new variable name. The resulting part of mutated model is given in Listing 2.21 where [randomID] is a randomly generated string for every mutated model. Now the probability that the database credentials already contains a username equal to the random generated value is usually very small and therefore, the mutation represents the SQL vulnerability to bypass the security check.

Listing 2.21: Message Handler for Creating a New Profile

```
symbols
  credentials( username_t, password_t ) : fact;
  RandomValue_[randomID] : username_t;

body {
  select{
    on( ?Us *->* Actor : createNewUser(
                              inst_username_t(?UsernameMessage,?UNounceMessage),
                              inst_password_t(?PasswordMessage,?PNounceMessage)
                     ) &
                     %@Semantics[ SQL ] {
                       !credentials(inst_username_t(?RandomValue_[randomID],?))
                     %@}
      ) : {
        maliciousValue_SQL_input_text( UsernameMessage ) ;
        ...
```

BindVariableToRandomValue in 10 Seconds

The Semantic Mutation Operatoris looking for annotations that contain the label `SQL`, and are part of a condition. If such an annotation is found, the mutation operator performs the following actions:

- A question-marked variable in the annotated guard expression is replaced by a dummy value. The mutation operator achieves this by using an uninitialized variable.
- The newly introduced variable is declared in the symbols section of the entity where the mutation operator is applied.
- The replaced variable is tracked using the special fact `maliciousValue_SQL_{Internal,Input}_[type of variable]( [Variablename] )`.

As for the `DetachVariableForSQL` Semantic Mutation Operator, `BindVariableToRandomValue` considers every possible combination of mutation candidates for generating mutated models. To specify a lower bound x and an upper bound y of applying mutations, the `BindVariableToRandomValueUpToLimit[x][y]` Semantic Mutation Operator can be used.

2.6.3. XSS Vulnerabilities and Their Semantic Mutation Operators

The problem of interpreting user-provided data without first sanitizing the data is not only a problem in the context of SQL. It is the underlying fundamental problem of other vulnerabilities like XSS injections, buffer overflow, and the like as well. In this section we focus on XSS attacks. Although the vulnerability occurs at the server side, the exploit is executed at the client side, compared to a server-side exploitation of e.g, an SQL injection. Since we already discussed the problem of sanitizing user-provided input data in the previous sections, we will keep the sections about XSS vulnerabilities shorter.

Input validation vulnerabilities are very typical and widely observable in practice. A web application requires user-provided input for its functionality. Since the user is a non-trusted agent in the web application model, such input needs to be sanitized to exclude malicious parts. If the sanitization is not done properly, user-provided input can be crafted that gets interpreted by the browser and therefore can change the behavior of the web application. Such vulnerabilities can occur because the complete sanitization functionality is missing, or because there exists at least one specific input for which the sanitization functionality is inefficient. At a conceptual level, these issues also apply for SQL vulnerabilities, as already discussed in Section 2.6.2.

A common classification of XSS attacks is *reflexive*, *stored*, and *DOM-based* XSS. In this Ph.D thesis, we will focus on the first two types of XSS. Semantic Mutation Operators for *DOM-based* XSS vulnerabilities are not covered since the malicious input will never be sent to the server-side component of the web application but stays in the browser all the time. Therefore, to discover *DOM-based* XSS attacks, the client-side browser needs to be modeled in greater details than our considered formal specifications do. Therefore, we focus on *reflexive* and *stored* XSS vulnerabilities and provide a very short description of these two kinds of vulnerabilities in this section. A more comprehensive one is available e.g. at OWASP [45, 46].

Reflected XSS. This is an attack where the server accepts user-provided data that it sends back to the client. Figure 2.12 illustrates the conceptual information flow. In the first step the attacker prepares a special link that contains malicious parts, represented by `link[malicious data]`. It sends this link to the victim e.g. by email or instant message services. In the second step he waits until the victim clicks on the link so that the victim's browser will transform the link into a corresponding HTTP request. Sending this request to the server, the malicious part contained in the link is transmitted to the server as part of the HTTP request (`http-request[malicious data]`). A web application vulnerable to XSS does not sanitize data provided by the user; and therefore, the server sends back a response for the HTTP request that contains the malicious data as well (`http-response[malicious data]`). Because user-provided data is immediately sent back in the response of an HTTP request, such an attack is called **reflected** XSS. The victim's browser trusts all data from the server, therefore, it will process and render the malicious data.

Stored XSS. Figure 2.13 shows the information flow for a **stored** XSS attack. The attacker communicates with the web application directly instead of contacting the victim. He interacts with the application and exploits a vulnerability to store malicious data on the server (`http-request[malicious data]`). After storing the malicious data on the server, the attacker has to wait until the victim visits the web application. In particular the victim has to send a request to the server that triggers a response that contains the malicious data. In contrast to the reflected XSS attack, a stored attack targets victims without the need of contacting each of them individually.

In both kinds of attacks, the malicious data consists of script data, that is interpreted by the victim's browser. The XSS attack misuses the trust the browser has in the web application. The web browser considers anything that the web application sends as trusted and therefore executes/interprets it.

2.6.3.1. Vulnerability: Sanitization in the Context of Stored XSS Attacks

Compared to reflected XSS, (multi-step) stored XSS is more difficult to discover. Therefore, we skip a more detailed discussion of reflected XSS attacks but focus on stored ones. OwnCloud [24] is an open source implementation of a cloud service. It primarily allows file sharing but is extensible with a variety of plug-ins. In the past, OwnCloud suffered from many vulnerabilities. According to the *CVE Details* website [23], 80 vulnerabilities for the OwnCloud web application were reported between 2012 and July 8 2014. 35% of all reported vulnerabilities are related to XSS, but only 3.8% to SQL Injection (SQLI).

In this web application a missing input XSS sanitization that leads to a stored XSS vulnerability can be found. The web application provides a functionality to create a calendar based

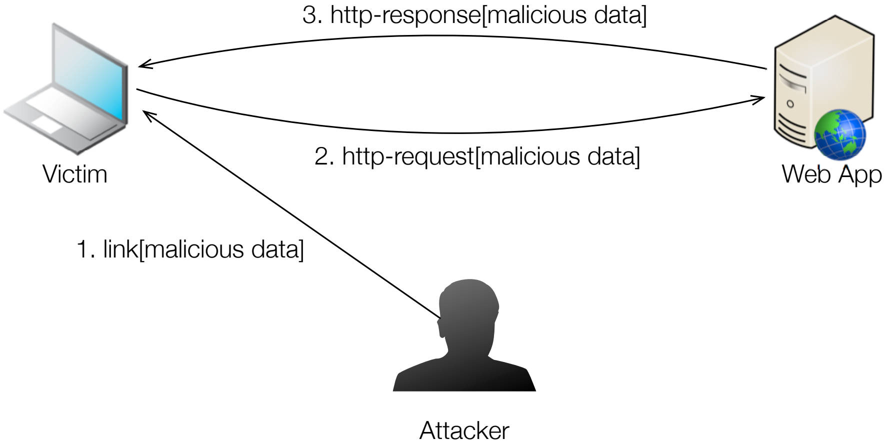

3. http-response[malicious data]
Victim
2. http-request[malicious data]
Web App
1. link[malicious data]
Attacker

Figure 2.12.: Reflected XSS: Conceptual Information Flow

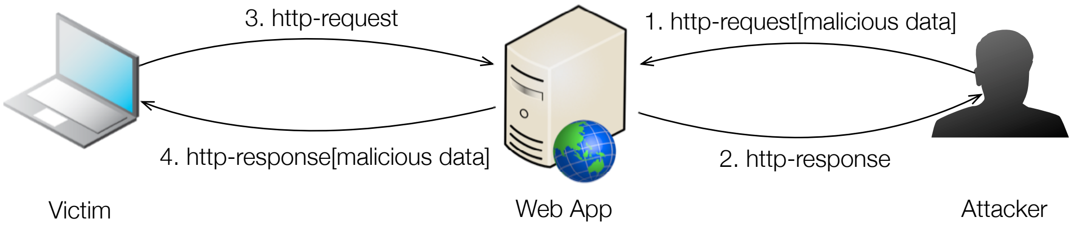

3. http-request
1. http-request[malicious data]
4. http-response[malicious data]
2. http-response
Victim
Web App
Attacker

Figure 2.13.: Stored XSS: Conceptual Information Flow

on data that is sent by the user as part of an HTTP request (Line 3 in Listing 2.22). None of the parameters title (Line 18), location (Line 19), and description (Line 31) is sanitized against XSS. The vulnerability is part of the updateVCalendarFromRequest method, that is called from within the createVCalendarFromRequest method (Line 7). After the calendar has been created, it is added to the database by invoking the method add of the object OC_Calendar_Object (Line 41). This method parses the data (Line 42), serializes it (Line 46), and finally stores the non-sanitized user-provided data using an SQL query in Lines 53 to 55. Upon reading this data later and render it in the browser, the missing sanitization against XSS allows an attacker to inject malicious code. The developer of OwnCloud fixed this concrete vulnerability by invoking the strip_tags function call for each parameter (Lines 20, 21 and 32 in Listing 2.22).

Product	Owncloud 4.0.1
CVE ID	CVE-2012-4396 http://www.cvedetails.com/cve/CVE-2012-4396/
Vulnerability Type	Cross Site Scripting
Location	apps/calendar/lib/object.php

Listing 2.22: CVE-2012-4396

```
1  %% owncloud-4.0.1/apps/calendar/ajax/event/new.php
2  $cal = $_POST['calendar'];
3  $vcalendar = OC_Calendar_Object::createVCalendarFromRequest($_POST);
4  $result = OC_Calendar_Object::add($cal, $vcalendar->serialize());
5
6  %% owncloud-4.0.1/apps/calendar/lib/object.php
7  public static function createVCalendarFromRequest($request)
8  {
9    ...
10   return self::updateVCalendarFromRequest($request, $vcalendar);
11 }
12
13
14 %% owncloud-4.0.1/apps/calendar/lib/object.php
15 @@ -600,8 +600,8 @@ public static function createVCalendarFromRequest($request)
16 public static function updateVCalendarFromRequest($request,$vcalendar)
17 {
18 - $title = $request["title"];
19 - $location = $request["location"];
20 + $title = strip_tags($request["title"]);
21 + $location = strip_tags($request["location"]);
22   $categories = $request["categories"];
23   $allday = isset($request["allday"]);
24   $from = $request["from"];
25
26 @@ -611,7 +611,7
27 @@ public static function updateVCalendarFromRequest($request, $vcalendar)
28     $totime = $request['totime'];
29   }
30   $vevent = $vcalendar->VEVENT;
31 - $description = $request["description"];
32 + $description = strip_tags($request["description"]);
```

```
  $repeat = $request["repeat"];
  if($repeat != 'doesnotrepeat'){
    $rrule = '';


%% owncloud-4.0.1/apps/calendar/lib/object.php
class OC_Calendar_Object{
  ...
  public static function add($id,$data){
      $object = OC_VObject::parse($data);

      ...

      $data = $object->serialize();

      ...

      $stmt = OCP\DB::prepare( 'INSERT INTO *PREFIX*calendar_objects (
           calendarid,objecttype,startdate,enddate,repeating,
           summary,calendardata,uri,lastmodified )
        VALUES(?,?,?,?,?,?,?,?,?)' );

      $stmt->execute(array(
        $id,$type,$startdate,$enddate,$repeating,$summary,$data,$uri,time()
      ));
  ...
  }
}
```

2.6.3.2. Semantic Mutation Operator: XSSRemoveSanitization and XSSRemoveSanitizationUpToLimit

The same kind of problems occur in our example in Listing 2.8. In Line 15, the web application accepts the `store` message that is used to write profile data into the database. Later, these profile data can be requested using the `view` message (Line 36). To prevent XSS attacks where the malicious data first is sent to the web application before returned to the victim's browser, the web application needs to sanitize the user-provided data. In the same way as the Security Analyst annotates an ASLan++ block for SQL sanitization, he annotates a block that represents the sanitization of data against XSS using the semantics keyword `Sanitize`. Depending on the semantics of the sanitization the Security Analyst further provides one of the following semantics keywords `HTML`, or `Javascript`, dependent on in which context the user-provided data is used and which kind of sanitization function is used.

- If the user-provided data is suppose to be inserted into an HTML code fragment, sanitization functions like *strip tags*, *htmlspecialchars*, or *htmlentities* are used. In this situation, the Security Analyst uses the `HTML` semantics keyword for the annotation.

- Sometimes, a user-provided value is assigned to a JavaScript variable, like e.g., `<script>var myVar = '<?php echo $variable; ?>';</script>`. In this situation, the developer needs to use JavaScript dedicated sanitization function. E.g., WordPress [43] offers the function `wp_localize_script()` to perform this kind of sanitization. Since the user-provided input is embedded into JavaScript code the Security Analyst uses the `JAVASCRIPT` semantics keyword.

This context information is later used for selecting appropriate malicious input. Finally, the Security Analyst annotates a sanitization block with either the semantic keyword `Input`, `Implicit` or `Internal`. Sanitizing `Input` data is performed by the web application before it further processes user-provided data. E.g, in Line 24 in Listing 2.8 the user-provided input profile data is sanitized against `HTML`. Applying the `XSSRemoveSanitization` Semantic Mutation Operator leads to the mutated partial model shown in Listing 2.23. `Implicit` is used to express that the sanitization property is propagated to copies of the data value (Line 34). Such an annotation leads to the mutated partial model given in Listing 2.25. Finally, sanitizing `Internal` data is performed on data that the web application processes from a location other than a request directly, e.g., data read from a database. This functionality is used as part of the `view` message handler in Line 43. Applying `XSSRemoveSanitization` leads to the mutated partial model given in Listing 2.24.

Vulnerabilities that refer to reflected and (multi-step) stored XSS attacks can be represented using the two Semantic Mutation Operators `XSSRemoveSanitization` and `XSSRemoveSanitizationUpToLimit`. Interestingly, these two mutation operators internally operate exactly in the same way as the Semantic Mutation Operators `SQLRemoveSanitization` and `SQLRemoveSanitizationUpToLimit` respectively. The semantics upon applying them to a formal specification is given by the semantics keywords as described in the previous paragraph.

Listing 2.23: Applying XSSRemoveSanitization

```
%@Semantics[ HTML, Sanitize, Input ]
-   sanitizeTextHTML( D ) ;
+   removedXSSSanitization_HTML_Sanitize_Input_text( D ) ;
```

Listing 2.24: Applying XSSRemoveSanitization

```
%@Semantics[HTML, SanitizeByEscaping, Implicit(D)]
-   sanitizeTextHTML( D2 ) ;
+   removedXSSSanitization_HTML_Sanitize_Implicit_text( D2 ) ;
+   removedXSSSanitization_HTML_Sanitize_Implicit_text( D ) ;
```

Listing 2.25: Applying XSSRemoveSanitization

```
%@Semantics[HTML, SanitizeByEscaping, Internal]
-   sanitizeTextHTML( D ) ;
+   removedXSSSanitization_HTML_Sanitize_Internal_text( D ) ;
```

XSSRemoveSanitization in 10 Seconds

The Semantic Mutation Operator is looking for annotations that contain one of the language labels `HTML`, or `Javascript` and the label `Sanitize`. If the formal specification contains such an annotation, the mutation operator performs the following actions, depending on the annotated expression.

- If the sanitization is modeled by introducing a fact, the General Mutation Operator `InvalidateIntroduceFact` is applied.
- If the sanitization is modeled by retracting a fact, the General Mutation Operator `InvalidateRetractFact` is applied.
- If the sanitization is modeled by applying a symbolic function to a variable and assign the application to a variable, the General Mutation Operator `RemoveFunctionFromAssignment` is applied.

In all three situation, the mutation operators add the fact `removed_[language label]_[type of value]_Sanitized_[parameter signature](value)` to the specification.

Depending on the modeling and the complexity of vulnerabilities that are considered, generating mutated models by applying the Semantic Mutation Operators only once is potentially not enough. To let the model checker find AATs related to more complex vulnerabilities, we need to apply the Semantic Mutation Operators multiple times. Therefore, the above described mutation operators `XSSRemoveSanitization` considers any combination of sanitization blocks against XSS. Furthermore, the Semantic Mutation Operator `XSSRemoveSanitizationUpToLimit[x][y]` can be configured by specifying the minimal x and maximal number y of considered sanitization block per mutated model. The semantics is equivalent as for the `SQLRemoveSanitization` mutation operator. Therefore, more details can be found in Section 2.6.2.1.

2.6.3.3. Vulnerability in the Context of Split XSS

An example of a split XSS vulnerability is given in a blog post called ‘Bypassing Chrome’s Anti-XSS filter’ [29]. The author created a minimal example application that demonstrates the essence of the vulnerability. The webpage reads two parameters `a` and `b`, that are part of the HTTP request `http://securitee.tk/files/chrome_xss.php?a=INPUT1&b=INPUT2`. If malicious XSS code like `<script>alert(1);</script>` is injected solely using parameter `a`, the anti XSS filter will recognize the malicious input and filter it out. The same behavior can be observed using parameter `b` for the injection. Although the filter mechanism is effective on each parameter separately, it is not effective if the injection is using both parameters at the same time. Therefore, the malicious input is split into two parts. Such a partial input is not recognized by Chrome’s anti XSS filter and therefore, the content of the parameter is sent back to the client in a non-sanitized form. Using two different parameters for the injection, it is very likely that the content of the parameters are displayed in different parts of the webpage. At the same time, the content of both parameters must be glued together in order to represent valid code that can be executed. A concrete pair of inputs that eliminates the 'gluing' problem is presented in the blog post. The first part of the malicious input terminates with the HTML comment sign `/*` for starting a multi-line comment. Similar the second part start with the HTML comment sign `*/` to end a multi-line comment. This attack behavior is very similar to the split SQL injection described in section 2.6.2.4.

2.6.4. General Mutation Operators

In the previous sections we have introduced the Semantic Mutation Operators for two important and in practice relevant security vulnerabilities. To implement them several General Mutation Operators are needed that we discuss in this section. The set of General Mutation Operators is not complete in general, but is sufficient to realize the Syntactic and Semantic Mutation Operators considered in this Ph.D thesis. In a nutshell, General Mutation Operators take a set of Basic Operations and apply them all to the same specification. Applying a General Mutation Operator to a formal specification preserves the type correctness of the model. In particular, a model checker can process such a mutated model without syntactic and validation errors. E.g., a formal specification that is syntactically correct but contains a variable that is not declared, generates a validation error. Similar to Semantic Mutation Operators, General Mutation Operators keep track of the changes that they perform on the formal specification.

InvalidateIntroduceFact, InvalidateRetractFact. In order to mutate facts appearing as statements, SPaCiTE introduces two different General Mutation Operators. The mutation operator `InvalidateIntroduceFact` invalidates a fact that was introduced. Similar, the mutation operator `InvalidateRetractFact` invalidates a statement that retracts a fact[9]. The two mutation operators work as follows: Both of them require the following Basic Operations: *ChangeFactName*, *RemoveEObject*, *AddSymbolsSection*, and *AddFactSymbolDeclaration*. We will discuss them in more details in Section 2.6.5.

The `InvalidateIntroduceFact` General Mutation Operator is applicable on statements that introduce a fact, whereas the mutation operator `InvalidateRetractFact` is applicable on statements that retract a fact. Conceptually, both mutation operators remove the statement from the specification and add a value-tracking fact instead. For the `InvalidateIntroduceFact` mutation operator, SPaCiTE combines these two actions by changing the fact name instead of removing and adding a new fact. This optimization is not implemented for the `InvalidateRetractFact` since 'the optimization' already requires two actions, — the removal of the keyword `retract`, and changing the fact name. The new fact name is the concatenation of the word '`removed`' and the initial fact name. To keep the specification model verifiable, both General Mutation Operators add the corresponding new fact declaration to the

[9] Please note that `retract` is an operation and not a fact. Therefore, one cannot check whether the action `retract` was indeed performed. In particular, if a fact is evaluated to `false`, it does not imply that it was retracted. It could also be the case that it has never been introduced so far.

symbols section. In terms of finding attack traces in the specification, in the vast majority of cases renaming a fact name and removing a fact are equivalent under the assumption that the renamed fact is not used for any further operations during model checking. Intuitively, since facts cannot be stored in variables, the new fact would influence other parts of the specification if the new fact name is referenced directly 'by name'. Even if a security goal does not depend on fact names directly but indirectly, e.g., via the number of elements in a set, adding the new fact to a set would require to reference the new fact 'by name'. The above assumption excludes such operations.

Introduce2RetractFact[10]. There are traces in which removing/renaming a fact-introducing statement is not sufficient. If a fact is introduced several times, the corresponding fact is evaluated to true, independent on how many times it is introduced (but introduced at least once). If the evaluation of the fact needs to be set to false, applying InvalidateIntroduceFact to any subset of the fact-introducing statements is not sufficient. It would require to apply it to every fact-introducing statement. More conveniently is the action retract. Since introducing the fact with the same data element d several times is semantically equivalent to introducing the fact only once, the multiple times introduced fact can be invalidated at once by retracting it rather than 'not introducing' it. Therefore, applying the General Mutation Operator Introduce2RetractFact to a fact introducing statement s performs the following operation.

- The statement s that introduces a fact F is replaced by a statement that retracts the same fact F.

In contrast e.g., to the InvalidateIntroduceFact mutation operator, the fact is already declared in the symbol section. Otherwise, the provided input specification would not be model-checkable. Therefore, the Introduce2RetractFact mutation operator does not need to modify the symbols section. In terms of the value-tracking fact, this mutation operator performs exactly the same operations as the mutation operators InvalidateIntroduceFact and InvalidateRetractFact.

Retract2IntroduceFact[10]. For the Retract2IntroduceFact General Mutation Operator the same argumentation as for the Introduce2RetractFact can also be applied. It is considered as the complement of the Introduce2RetractFact mutation operator. It operates on statements that retract a fact. When a fact F is retracted, evaluation F returns false from now on, independent how many times the same fact was introduced before. The mutation operator Retract2IntroduceFact performs the following operation.

- A statement that retracts a fact F is replaced by a statement that introduces the same fact F.

The RetractFact2IntroduceFact mutation operator takes as input a statement that retracts a fact. In contrast e.g., to the InvalidateIntroduceFact mutation operator, the fact is already declared in the symbol section. Otherwise, the

[10] The two General Mutation Operators Introduce2Retract and Retract2Introduce are listed here for completeness reasons. For the models used in our evaluation, these two General Mutation Operators are not needed and therefore, they are disabled for the evaluation.

provided input specification would not have been model-checkable. Therefore, the Retract2IntroduceFact mutation operator does not need to modify the symbols section. In terms of the value-tracking fact, it inherits the behavior from the InvalidateIntroduceFact General Mutation Operator.

RemoveFunctionFromAssignment. In ASLan++, values and symbolic functions can be assigned to variables. To apply a symbolic function with the name mySymbolicFunction to a variable V of any type T, the Security Analyst declares the function as mySymbolicFunction(T) : T;. To apply this function to value V and assigning it to a variable V', the Security Analyst writes:

```
  V' := mySymbolicFunction(V);
```

Applying the General Mutation Operator RemoveFunctionFromAssignment removes the symbolic function mySymbolicFunction and directly assigns the value of V to variable V'. Since V and V' do not necessarily have to have the same type, and the symbolic function might require more than one argument, the mutation operator is only applied if V has the same type or is a subtype of the type of V, and if the function only requires one argument. In addition to removing the symbolic function, an additional value-tracking fact is added. In the above context, the mutation operator adds the specific fact removedmySymbolicFunction_T(V) to tell the TEE, that the content of V potentially contains malicious data.

DetachVariable. The general DetachVariable mutation operator takes as parameter a question-marked variable and removes the variable name. To illustrate the functionality of the mutation operator, let's have a look at the example given in Listing 2.26: The condition fact0(?A, ?B) & fact1(?A) & fact2(?B) is used in three different contexts — as part of a while-condition (Line 2), if-condition (Line 3), and selectOn- condition (Line 5). We assume that variable A is of type text. The General Mutation Operator requires a reference to a question-marked variable as a parameter. For this example, we provide a reference to the variable A that is part of the fact fact0 to the General Mutation Operator.

Since variables A and B are question-marked, the model checker tries to find values for A and B to violate any of the defined security goals in the same specification. Because both variables A and B are part of different facts in the condition, different restrictions for possible values for A and B have to be fulfilled. E.g., A is affected by fact0 and fact1, whereas B is affected by fact1 and fact2. To make the overall evaluation of the guard independent of the variable A in fact0, one idea is to remove fact0 from the overall guard. Since fact0 has multiple parameters, it not only restricts possible values for variable A, but for others as well. Therefore, removing fact0 is not advisable. To avoid such side effects, we detach variable A in fact0 by replacing it with a pure question mark. Please note, that the value the model checker assigns to the question mark in an AAT is not accessible in the model anymore, since it represents a so called don't care placeholder or an anonymous variable.

Listing 2.26: DetachVariable Example

```
1 body {
2   while( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
3     if( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
4       Select{
5         on( fact0( ?A, ?B ) & fact( ?A ) & fact2( ?B ) ) : {}
6       }
7     }
8   }
9 }
```

In addition detaching the variable, the General Mutation Operator DetachVariable keeps track which variable was mutated. It inserts the value-tracking fact at different locations, depending on the guard type.

In the following listings we show the effect of applying the General Mutation Operator. The listings are given in diff format, that means that the lines starting with the minus sign (-) correspond to the original formal specification. Lines starting with the plus sign (+) show the result after applying the mutation operator(s).

In Listing 2.27 we show the effect of applying the General Mutation Operator to the first parameter of the fact fact0 in Line 1. The mutated variable is part of a while guard, therefore, the value-tracking fact is inserted as the first statement of the while-body (Line 3) and as the first statement after the complete while-body (Line 10). The General Mutation Operator inserts both value-tracking facts because it is not guaranteed that the while-body is indeed entered and if it is, if the trace continues after the while. It might be the case that the trace is blocked inside the while-body so that the statement right after the while-body is never reached. Therefore, both value-tracking facts are added.

Listing 2.28 shows the application of the mutation operator to the first parameter of fact0 in Line 2. The mutated variable is part of a branch guard, the value-tracking fact is inserted as the first statement of the if-statement block (Line 4) and as the first statement after the else-statement block (Line 10). The second one is even added if the else branch is missing. In this case, the mutation operator adds the complete else branch to the model. The value-tracking fact is not added after the complete if-then-else block because it is not guaranteed that the complete block is always part of a reported AAT. Due to the mutation, a specified security property might be violated at any statement inside the if or else branch. To have the value-tracking fact part of the AAT in any case, it is added to both the if and the else branch.

Finally in Listing 2.26 the mutation operator is applied to the first parameter of the fact fact0 in Line 4. The mutated variable is part of a selectOn guard, the value-tracking fact is inserted as the first statement of the selectOn body (Line 6). Since a selectOn guard does not have a similar concept like the else branch for an if-then-else branch, the mutation operator adds the value-tracking fact to the corresponding selectOn block only.

Listing 2.27: DetachVariable Example 1

```
-  while( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
+  while( fact0( ?, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
+    maliciousValue_Unknown_text ( A ) ;
     if( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
       Select{
         on( fact0( ?A, ?B ) & fact( ?A ) & fact2( ?B ) ) : {}
       }
     }
   }
+  maliciousValue_Unknown_text ( A ) ;
```

Listing 2.28: DetachVariable Example 2

```
while( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
-  if( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
+  if( fact0( ?, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
+    maliciousValue_Unknown_text ( A ) ;
     Select{
       on( fact0( ?A, ?B ) & fact( ?A ) & fact2( ?B ) ) : {}
     }
   }
+  else {
+    maliciousValue_Unknown_text ( A ) ;
+  }
}
```

Listing 2.29: DetachVariable Example 3

```
while( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
  if( fact0( ?A, ?B ) & fact1( ?A ) & fact2( ?B ) ) {
    Select{
-     on( fact0( ?A, ?B ) & fact( ?A ) & fact2( ?B ) ) : {
+     on( fact0( ?, ?B ) & fact( ?A ) & fact2( ?B ) ) : {
+       maliciousValue_Unknown_text ( A ) ;
      }
    }
  }
}
```

Besides the location of the value-tracking statement, the mutation operator automatically determines the specially crafted fact name of the form `maliciousValue_[vulnerability type]_[type]([variable name])`. The variable name is the name of the question-marked variable that the mutation operator removes and `[type]` is its type. Since a General Mutation Operator is not motivated by security related vulnerabilities, the mutation operator does not know the underlying source code fault and therefore, it sets the vulnerability type to `Unknown`.

ReplaceVariableByRandomValue. This General Mutation Operator is the dual view of the previous mutation operator. Where DetachVariable gives the model checker more freedom to choose a value for a parameter, the ReplaceVariableByRandomValue sets it to a fixed random value. Internally, the mutation operator makes use of the RenameVariable basic operations because a variable is not substituted by the random value directly, but with a variable whose value is random. Since both the DetachVariable and ReplaceVariableByRandomValue mutation operator operate on the same artifact of the model, adding the value-tracking fact works exactly the same for both of them.

To fulfill the requirement that the mutated model is still model-checkable without errors after applying a General Mutation Operator, this General Mutation Operator handles a special case — an anonymous variable that appears in a clause. A clause has the form clausename(argument list) : head :- body. If an anonymous variable in the body is replaced by a random value as described above, the random value variable must either appear in the head or the argument list of the clause. Since anonymous variables do not appear in the head, and changing the head signature requires multiple changes in the specification (declaration and every use of the head), the ReplaceVariableByRandomValue General Mutation Operator adds the random-value variable to the argument list. As a short illustration, Listing 2.30 shows the application of the mutation operator to the second argument of the instance_username_t expression.

Listing 2.30: ReplaceVariableByRandomValue in Clauses

```
clauses
-    label_listStaffParameterData( Response, ListStaffData ) :
-      getUsername( Response, ListStaffData ) :-
-        Response = instance_username_t( ListStaffData, ? );

+    label_listStaffParameterData( RandomValue, Response, ListStaffData ):
+      getUsername( Response, ListStaffData ) :-
+        Response = instance_username_t( ListStaffData, RandomValue );
```

IntroduceFact. Finally, this General Mutation Operator introduces a fact to the model and makes sure that the fact name is correctly declared. As input parameters, it takes the fact to be introduced, the signature of the fact, and the position where the new fact has to be added.

2.6.5. Basic Operations

In the previous section we have presented a set of General Mutation Operators that are sufficient to realize the Syntactic and Semantic Mutation Operators considered in this Ph.D thesis. In this section we present a set of Basic Operations. This set is not complete in the sense that it represents every possible operation that can be performed on ASLan++ neither. Nevertheless the set of Basic Operations is sufficient to implement the discussed General Mutation Operators.

AddFactSymbolDeclaration. Each fact, variable, constant, etc. needs to be declared. `AddFactSymbolDeclaration` is an operation that adds a new fact declaration to the symbol section of an entity. This Basic Operation needs two parameters — the signature of the fact symbol and the entity that declares this new symbol. Since `AddFactSymbolDeclaration` is a basic action it requires that the specified entity which should declare this new symbol, already contains a symbol section.[11] Therefore, trying to add a new symbol declaration (using the `AddFactSymbolDeclaration` Basic Operation) to an entity that does not already have a symbol section will fail. Nevertheless a dedicated Basic Operation is available to create such a symbol section (see `AddSymbolsSection`).

AddSymbolsSection. To add new symbol declarations to a model a `symbols` section must be present. Since this section is optional in an entity (not every entity needs its own `symbols` section), the Basic Operation `AddSymbolsSection` will add such a section.

ChangeFactName. An instance of the fact `myFact(...)` is renamed to ‘newName’ by applying the Basic Operation `changeFactName(‘‘newName’’)` on the fact `myFact`. Please note that this Basic Operation is designed to work on a per-usage level. This means that changing the name of a fact does not influence the declaration of the original fact name, nor the use of the fact name at other places in the specification.

ChangeKeyword. This Basic Operation takes an arbitrary keyword in the specification and replaces it by a given string. As a special use case, this operation can e.g., be used to overwrite the keyword `retract` (see Line 14 in Listing 2.2) by the empty string.

IntroduceFact. A mutation operator uses the `IntroduceFact` Basic Operation to add a new fact statement to the model. In addition to the pure new fact expression, the location where this fact is introduced is identified by two parameters: an element e of the Abstract Syntax Tree (AST) of the original model and a position information consisting of either `add_before`, `add_after` (relative to the element e), or `replace_element`.

DetachVariable. The Basic Operation `DetachVariable` considers variables prepended with a question mark (?), typically used in guards (conditions, horn clauses, security goals, etc. (see AVANTSSAR Deliverable 2.3 [62])). The semantics of such a variable is that the model checker is trying to find a value for that variable to finally violate a specified security goal. Since a question-marked variable has a name[12] it can be used in several expressions of the same guard at the same time. The more often a question-marked variable appears, the more constraints have to be fulfilled by a value for that variable. The Basic Operation `DetachVariable` substitutes a question-marked variable by a question mark only. That means that the number of constraints for possible values of the question-marked variable is decreased.

[11] See the ASLan++ grammar for syntactic details of a symbol section in an ASLan++ specification.

[12] In particular, it is not an anonymous variable.

Listing 2.31: Example AAT

```
<client> *->*  server  : register(INJECT(SQL),firstname1,lastname1,password1)
 server  *->* <client> : ack_register
<client> *->*  server  : loginAdmin(username_admin,password_admin)
 server  *->* <client> : listPendingAccounts(username1,firstname1,lastname1)
<client> *->*  server  : activateClient(username1)
 server  *->* <client> : VERIFY(client)
```

RenameVariable. This Basic Operation renames a variable to a given name. E.g., mutation operators use this Basic Operation to rename a variable to a variable name that is declared but not initialized. Such a variable gets a random/dummy value assigned.

RemoveEObject. The `RemoveEObject` Basic Operation removes the referenced node of the AST from the specification. Other related artifacts like possible declarations of the object to be removed are not deleted.

AddVariableToClauseArgumentList. Variable names that appear in the body of a clause have to appear in the head or the argument list. This Basic Operation adds a variable to the argument list.

2.7. Mapping AATs to Operational Test Cases

[The basic underlying concepts of the following section has already been published by Büchler et al. [83], but has been extended.]

In the previous section we have shown what SPaCiTE-conform models are and how different Semantic Mutation Operators modify such models. In this section we assume that SPaCiTE was used to model check mutated formal specifications and that the model checker reports a set of AATs. Therefore, we discuss how such reported AATs are transformed to executable attack traces. These steps correspond to Figure 2.2#d-h.

Listing 2.31 shows an example of an AAT. An Abstract Attack Trace (AAT) consists of a sequence of abstract messages exchanged between different entities. These messages may contain parameters where some of them are used for malicious inputs. Such parameters are indicated with the special keyword `INJECT(SQL)`, `INJECT(HTML)`, etc. to know what type of malicious payload has to be injected. Similar, the AAT can contain the special message `VERIFY` to indicate after which sequence of messages the victim is attacked.

To operationalize such AATs they need to be instantiated. Since web applications are usually accessed with the help of a browser, we consider the browser level in between the AAT and the SUV. Executing test cases requires the help of the Security Analyst since the abstraction gap has to be bridged between the AAT and the SUV. We believe that asking the Security Analyst for a mapping of an AAT to a sequence of actions performed in the browser is easier and more convenient than providing a mapping directly to protocol level messages. In addition the mapping of AATs to executable source code consists of application-dependent and application-independent information. Therefore, we split the process of making AATs operational into two different steps by adding an additional intermediate level (❷ in Figure 2.14) in between the AAT layer (❶ in Figure 2.14) and the implementation layer (❸ in Figure 2.14). These three layers have different purposes. Layer ❶ describes the AAT as a sequence of abstract messages as given by the output of the model checker. Therefore, the first mapping takes as input an AAT and maps it to a sequence of abstract browser actions, expressed in the WAAL language. WAAL is a language that we developed for this Ph.D thesis to describe how exchanged messages between agents can be generated and verified in terms of actions a user performs in a web browser (Layer ❷). Providing this mapping is a manual task but supported with a Domain Specific Language (DSL). The second step is mapping WAAL actions to executable API calls using a specific framework. In our case we make use of the Selenium framework [26] and provide a mapping from WAAL to selenium API calls. Once such a mapping is defined, it can be automatically applied and reused for any test case. Therefore, layer ❸ describes the instantiated attack trace in terms of source code.

In the following sections, we first describe WAAL in Section 2.7.1. In Section 2.7.2 we present the first mapping from application-dependent messages to the intermediate level. In Section 2.7.3, we present the second mapping, from the intermediate level to executable test cases. Both mappings are illustrated with concrete examples. Finally, the TEE is described in Section 2.7.4.

2.7.1. Web Application Abstract Language (WAAL)

The discussed vulnerabilities in Section 2.6 are vulnerabilities that are conveniently exploited via the browser. The operationalization of the reported AATs can benefit from the fact that the browser automatically generates protocol level messages based on actions performed in the browser (click, type, select, etc.). To benefit from these advantages, we contribute with an abstract language that expresses actions an end user performs in the browser, called Web Application Abstract Language (WAAL). It is an abstract language for web application actions at the browser level. The purpose of this language is to define browser actions that an end user can perform to either send messages to a web server or check its responses. Thus, WAAL actions are split into two sets: Generation Actions (GAs) and Verification Actions (VAs).

WAAL actions operate on objects which we summarize in Table 2.1. The first column shows the elements expressed in WAAL, the second column the same element in HTML, if available. These elements are used by both GAs and VAs. Most of the elements are self explanatory, for the others, a short description is given.

Generation Actions (listed in Figure 2.15) represent a set of atomic actions that a user can perform when he uses a web application (e.g., *follow* a link, *click* on a button, *type* text into a text field). More complex actions can be described by a combination of such atomic actions. For example, log in via a form may correspond to the sequence: *select* the name from a menu, *type* the password into a text field, and *click* on the login button. Since it works at the browser level, GAs are similar to API methods from Selenium, a Web application testing framework. However, GAs are not API methods at source code level but abstract browser actions and therefore, they are technology independent. Because the EBNF of the GAs and VAs is provided in Appendix C.1, and the different WAAL actions are quite intuitive, we only briefly describe the actions:

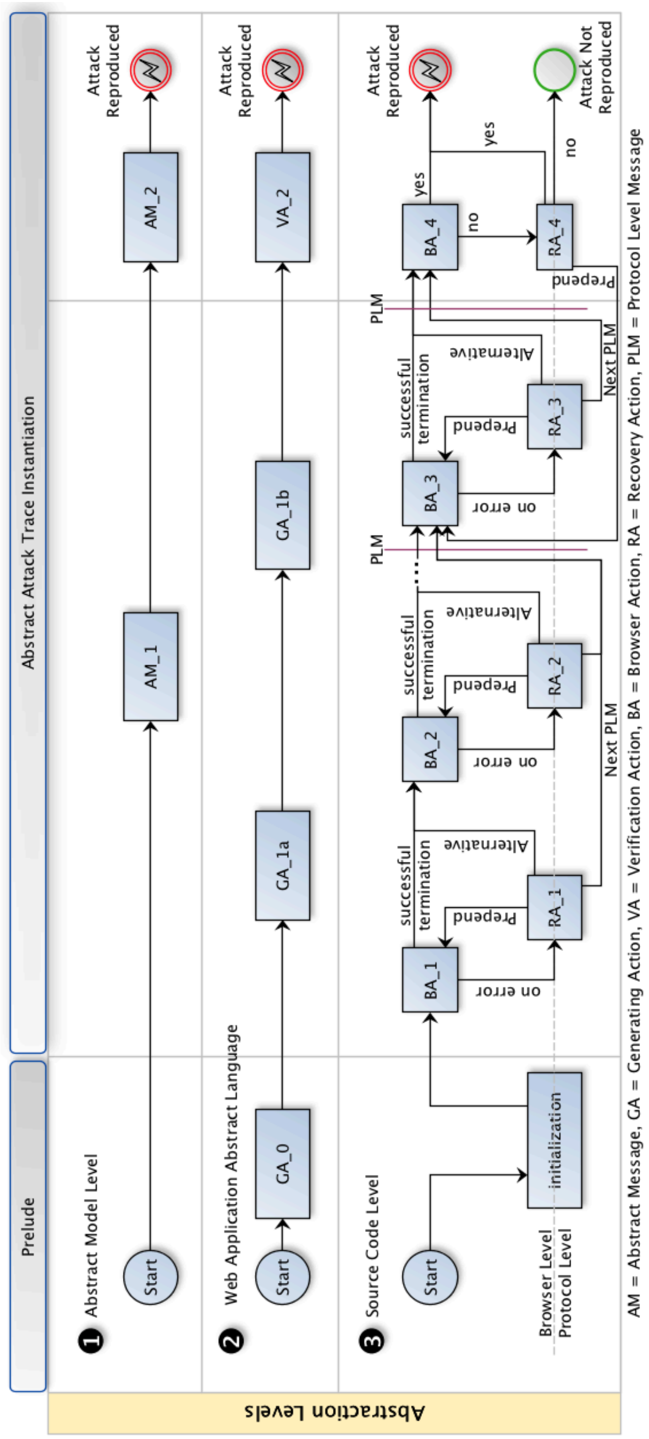


Figure 2.14.: Instantiation and Execution Methodology

WAAL Element	HTML Element	Remarks
Browser		This element represents the browser itself and is e.g., used for the action `goToURL`.
WebpageContent		The webpage content refers to the content that the browserWindow displays.
Button	<button>	
InputButton	<input>	
Textfield	<input>	
Dropdown	<select>	
Link	<a>	
SelectableList	<select>	
Div	<div>	
Span	<span>	
Image	<img>	
TextArea	<textarea>	
GenericElement		This element can be used to reference elements not listed above. In this situation, the complete XPath / CSS selector expression [27] has to be provided to uniquely identify the element.

Table 2.1.: Browser Elements Available in WAAL

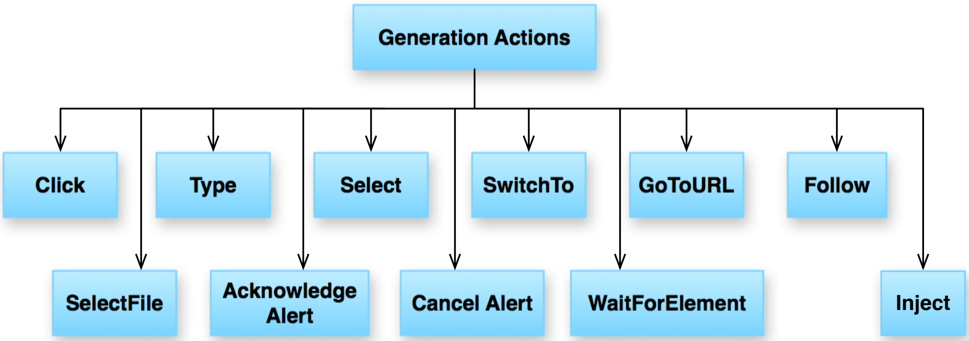


Figure 2.15.: Generation Actions (GAs)

GAs always start with the keyword `Generate::`, followed by the keyword `Element:` and the {`element`} on which the action is performed (see Table 2.1). The action is specified with the keyword `Action:` followed by the {`action`} to be executed and the required parameters.

```
Generate::
   Element: {element}
   Action: {action} [additional parameters]
```

Click. The click action takes an element identifier and performs a click action on this element. If the click action programmatically reads values from other elements, they can be specified with the optional parameters `ImplicitElement`, `ImplicitAttribute`, and `ImplicitValue`.

Type. The type actions takes two arguments. The first one identifies the textbox and is given by the `{element}` parameter. The second argument specifies the text to be typed into that textbox.

Select. The select action is used to select an item from a dropdown or list element. It takes two arguments — the identifier of the dropdown or list element (`Element`), and the identifier of the element to be selected.

SwitchTo. Depending on the used Selenium version, one needs first to switch to the correct frame before an action on an element in that frame can be executed. Such a switch is performed by the *switchTo* action. The frame to switch to is specified as the `Element` parameter.

GoToURL. To start using a web application, the *GoToURL* action is used to navigate to the homepage of the web application. The element this cation operates on is the `{Browser}`. The single additional argument of that action is the URL.

Follow. Before the web application is displayed in the browser, the user usually navigates to different pages by following links. This action is performed by the *follow* action which takes as a single argument the link identifier, given as the `Element` parameter.

SelectFile. To upload a file to the web application, the user usually opens a file selection dialog provided by the Operating System. Since this workflow temporarily leaves the browser context, WAAL specifies the action *SelectFile*. It expects two arguments — the element on the webpage to select the file (usually a button) (`Element`), and the file location on the filesystem as an additional parameter.

Acknowledge Alert / Cancel Alert. Upon executing a browser action, the web application might display alert windows that have to be acknowledged or canceled. Such an alert window is usually disrupting the workflow, i.e., it first has to be acknowledged or canceled before any further action can be performed. Therefore, the corresponding actions do not require an argument and acts on the currently displayed alert window.

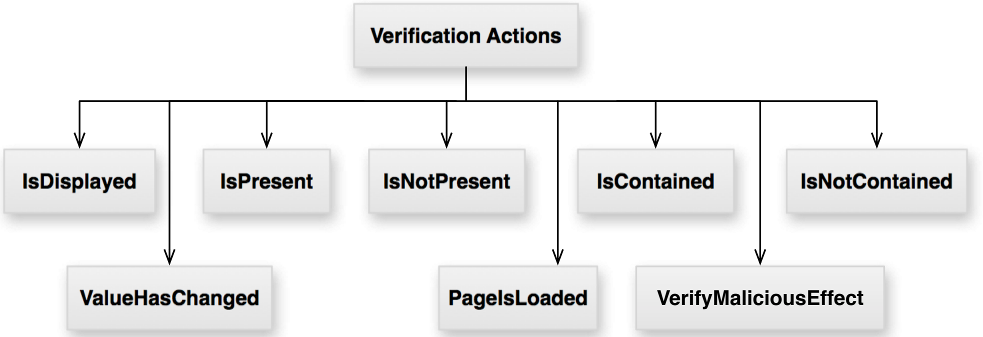


Figure 2.16.: Verification Actions (VAs)

WaitForElement. After performing an action, the next element for performing the next action might not immediately be available. To increase the robustness of a test case, the *waitForElement* action can be used to pause the execution until the desired element is available. The argument of that action is the identifier of the element (`Element` parameter) to wait for.

Inject. Finally, a security test case depends on malicious inputs that are injected to the web application. The argument of that action is the type of input (SQL, HTML, JAVASCRIPT), and optionally an integer to specify the order of injections in the context of a split attack. It is used to select corresponding malicious inputs from the instantiation library.

Verification Actions (listed in Figure 2.16) are used to verify whether an observed response matches with an expected one. The verification is performed according to a user provided criterion. WAAL supports the following criteria:

VAs always start with the keyword `Verify::`, followed by the keyword `Element:` and the {`element`} on which the verification action is performed. The condition to be checked is specified with the keyword `Condition:` followed by one of the below described conditions and the necessary parameters.

```
Verify::
  Element: {element}
  Condition: {condition} [additional parameters]
```

IsDisplayed/IsNotDisplayed. An element on a webpage is either displayed or not. To check this property of an element, WAAL offers the actions *IsDisplayed* and *IsNotDisplayed*. As an argument, they expect the identifier of the corresponding element (`Element` parameter).

IsPresent/IsNotPresent. An element might not be displayed, but it can still be present in a webpage. This property is checked using the WAAL actions *IsPresent* and *IsNot Present*.

TextIsContained/TextIsNotContained. Whereas the previous two criteria are applied to elements of the webpage, the actions *TextIsContained* and *TextisNotContained* checks the availability of a text string inside a web page displayed in the browser. Therefore, this action operates on the `WebpageContent` element.

ValueHasChanged. A lot of web applications are programmed in an asynchronous way. That means that values of web page elements can change without reloading the page. WAAL offers the action *ValueHasChanged* to verify if a specific value has changed according to a reference value provided as an additional parameter.

PageIsLoaded. To check whether a requested page has already been loaded, WAAL provides the corresponding action *PageIsLoaded*. It does not require any additional parameters, since the check is performed in the single active tab of the browser, given by the `Browser` element.

VerifyMaliciousEffect. This verification action corresponds to the *inject* action. As described above, the *inject* action requests an object from the instantiation library and asks that object for the malicious input. The action *VerifyMaliciousEffect* executes the verification action on the same object as used for the injection and checks the condition using the browser object (`Element` parameter). Therefore, it does not require any additional parameters.

The above described GA and VA sets define the foundations for WAAL:

$$WAAL = (GA^* \times VA^*)^*$$

A valid word in WAAL is a sequence of actions that either produce (GA^*) or verify (VA^*) exchanged messages. Note that we consider sequences only (and not trees) because this language is intended to represent the AAT at the browser level. In particular, WAAL does not support branches and the like. Since a trace from a model-checker is an abstract message sequence, a sequence of actions at browser level is sufficient to represent such traces.

2.7.2. Mapping From Abstract Model Level to Browser Level

2.7.2.1. General Principle

The output of the model checker is an AAT that consists of a sequence of exchanged messages. Each message m has a sender agent $\mathcal{S}$, a receiver agent $\mathcal{R}$ and a channel $\mathcal{C}$. Thus, the input $\mathcal{L}_1$ for the mapping to WAAL is defined as follows:

$$\mathcal{L}_1 = (A \times C \times A \times M)^*$$

where A is the set of agents, C is the type of channels used (confidential, authentic, or both), and M is the set of abstract messages exchanged between two agents. The mapping τ_1 maps each message m (together with its sender, receiver and channel) to a pair of sequences such that the first component of the pair generates m and the second component of the pair verifies m:

$$\tau_1 : (A \times C \times A \times M) \rightarrow (GA^* \times VA^*)$$

The actual mapping depends on the sender and the receiver. Each agent described in the model is either part of the SUV — the TEE can observe his behavior — or is simulated (stubbed) by the TEE. The former kind of agent is denoted by the set A_o, for observed agents, while the latter is denoted by the set A_s, for simulated agents. Partitioning the agent set A into A_o and A_s is the responsibility of the Security Analyst.

Given these two sets, the sequence of GAs ($\equiv \tau_1()|_G$ where $|_G$ returns the first component of the output pair of τ_1) for a sender $\mathcal{S}$, a channel $\mathcal{C}$, a receiver $\mathcal{R}$ and a message $\mathcal{M}$ is constructed as follows:

$$\tau_1(\mathcal{S}, \mathcal{C}, \mathcal{R}, \mathcal{M})|_G := \begin{cases} (ga_1, ga_2, \ldots, ga_n), & \text{if } \mathcal{S} \in A_s \\ (), & \text{if } \mathcal{S} \in A_o \end{cases}$$

where $n \in \mathbb{N}$ and $ga_i \in GA$ for all $1 \leq i \leq n$. Thus, if the sender $\mathcal{S}$ is a simulated agent, the message m is mapped to a sequence of GAs such that the message m is generated by a web browser after executing this sequence. If the sender is an observed agent, the TEE does not need to generate anything.

The sequence of VAs not only depends on A_o and A_s, but also on an assumption about the channel, namely whether sent messages can be assumed to be delivered unmodified. This assumption is called *integrity assumption*. If a channel $\mathcal{C}$ is integer, we write $integrity(\mathcal{C})$. The integrity of a channel $\mathcal{C}$ is helpful if such a channel is used by a simulated agent. Due to the integrity property the receipts of the message does not have to be verified. Therefore, the sequence of VAs ($\equiv \tau_1()|_V$ where $|_V$ returns the second component of the output pair of τ_1) for a sender $\mathcal{S}$, a channel $\mathcal{C}$, a receiver $\mathcal{R}$ and a message $\mathcal{M}$ is constructed as follows:

$$\tau_1(\mathcal{S}, \mathcal{C}, \mathcal{R}, \mathcal{M})|_V := \begin{cases} (), & \text{if } \mathcal{S} \in A_s \wedge \text{integrity}(\mathcal{C}) \\ (va_1, va_2, \ldots, va_n), & \text{otherwise} \end{cases}$$

where $n \in \mathbb{N}$ and $va_i \in VA$ for all $1 \leq i \leq n$. Thus, a message m is mapped to a sequence of VAs such that a browser can verify the received message m by executing this sequence. The only case where the TEE does not need to verify m is when m has been sent over an integrity channel by a simulated agent.

In addition to mapping every message from the attack trace to sequences of actions in WAAL, the Security Analyst must also provide an initialization block (*GA_0* in Figure 2.14) to prepare the execution of the attack trace. This initialization block is also expressed as actions in WAAL. The syntax definition of such an initialization block is given in Appendix C.1 and examples can be found at the end of Section 2.7.2.2 and in Appendices D.1 and D.2. Let us now give a concrete example of this mapping, by using WebGoat.

2.7.2.2. Example: Application to WebGoat

To illustrate the mapping τ_1 we consider the WebGoat application [39]. The application is explained in detail in Section 4.2. For now, the following information about WebGoat is sufficient. WebGoat is a simple web application to store and view user profiles[13]. A user can send a `login` message to authenticate and gets back the response `listStaffOf` from the web application. This message contains an ID of a profile the authenticated user is

[13] like a Google+ or Facebook profile

authorized to view. To request the profile, the user sends the message `viewProfileOf` to the web application and gets back the response `profile`.

For our example, let's consider an agent named `webServer` in A_o and an agent named `jerry` in A_s that must be controlled by the TEE. Furthermore, we assume the integrity of messages sent over the channels. Thus, messages generated by simulated agents do not have to be verified.

Applying our mutation-based approach to the formal specification of the WebGoat application leads to an AAT given in Listing 2.32. The AAT consists of four messages (`login`, `listStaffOf`, `viewProfileOf`, `profile`). `login` and `viewProfileOf` represent requests that are sent to the web application. `listStaffOf` represents the response sent to the client upon the `login` message. Finally, `profile` is the response sent to the client upon the `viewProfileOf` request. Listing 2.33 shows the mapping τ_1 of these four messages to sequences of actions in WAAL; Only the GAs and VAs relevant to the attack trace are shown. Left out is e.g., an initialization block (*GA_0*) that provides a way to put the system into a state suitable to run the attack trace.

Listing 2.32: Example AAT
(wS:=webServer)

```
<jerry> *->* wS   : login(username(jerry),password(jerry))
 wS *->* <jerry> : listStaff(username(tom))
<jerry> *->* wS   : viewProfile(username(tom))
 wS *->* <jerry> : profile(username(tom),tom_profile)
```

Definition 4. *During the specification of the WAAL mapping, the Security Analyst can use the special variables* `$1`*,* `$2`*,* `$3`*, etc. They refer to the following information:*

`$1` *refers to the actual sender of the message.*

`$2` *refers to the actual channel used to send/receive the message.*

`$3` *refers to the actual receiver of the message.*

`$4`*,* `$5`*, etc. refer to the actual first, second, etc. parameter values of the message.*

These special variables allow that the WAAL mapping can be reused for multiple AATs.

The mapping provided in Listing 2.33 translates to the following description:

login(usr,pwd). Whenever the abstract message `login` is exchanged, the TEE executes the following three generating actions (Lines 1 to 13 in Listing 2.33). In the considered testing environment, the integrity assumption for the channel `*->*` holds and therefore, the generated protocol level messages do not have to be verified. Therefore, the abstract message is only mapped to GAs (see Section 2.7.2).

1. In the browser that is used by `jerry` to communicate with the `webServer`, the TEE selects the element given by the first parameter of the 'login' message in the dropdown list with the name 'employee_id'. Note that the first parameter of the message is referenced by $4 (see Definition 4).

2. Using the same browser, the TEE types the password given by the second parameter of the 'login' message into the textfield with the name 'password'. Note that the second parameter of the message is referenced by $5 (see Definition 4).
3. Finally, the TEE clicks on the button with the attribute 'value=Login'.

listStaffOf. Whenever the client (`jerry`) receives a 'listStaffOf' message from the server, the TEE performs the following verification action (Lines 15 to 19 in Listing 2.33). Note, that no GA is required.

1. In the browser that receives responses for user jerry sent by the webServer, the TEE checks the webpage content if the text string 'Credit Card' is contained.

viewProfileOf. To make the abstract message 'viewProfileOf' operational, the TEE performs two generating actions. In the considered testing environment, the integrity assumption holds and therefore, the generated protocol level messages do not have to be verified:

1. The TEE chooses the browser used by jerry to communicate with the webServer and selects the element provided by the argument of the 'viewProfileOf' message in the list with the name 'employee_id'. Note that the first parameter of the message is referenced by $4.
2. After selecting the profile, the TEE performs a click on the button with the attribute 'value=ViewPofile'.

profileOf. Finally, the 'profileOf' message is mapped to one action that the TEE performs in the browser where the webServer's response for user jerry is sent to.

1. The TEE checks the webpage content if the text string 'View Profile Page'

Listing 2.33: Mapping of WebGoat Abstract Actions to WAAL

```
 1 τ1(login(usr, pwd)) = ((
 2   Generate:: "jerry" -> "webServer"
 3   Element: Dropdown: ByName("employee_id")
 4   Action: Select "$4" // $4 = user name of login message
 5
 6   Generate:: "jerry" -> "webServer"
 7   Element: Textfield: ByName("password")
 8   Action: Type "$5" DefaultValue "Default" // $5 = password of login message
 9
10   Generate:: "jerry" -> "webServer"
11   Element: InputButton: ByAttribute(Attribute: "value", Value: "Login")
12   Action: Click
13 ), ())
14
15 τ1(listStaffOf(usr)) = ((), (
16   Verify:: "webServer" -> "jerry"
17   Element: WebpageContent
18   Condition: TextIsContained:"Staff Listing Page"
19   )
```

```

τ1(viewProfileOf(usr)) = ((
  Generate:: "jerry" -> "webServer"
  Element: SelectableList: ByName("employee_id")
  Action: Select "$4" // $4 = profile ID of viewProfileOf message

  Generate:: "jerry" -> "webServer"
  Element: InputButton: ByAttribute(Attribute: "value", Value: "ViewProfile")
  Action: Click
), ())

τ1(profileOf(usr)) = ((), (
  Verify:: "webServer" -> "jerry"
  Element: WebpageContent
  Condition: TextIsContained: "Credit Card"
  )
```

Finally, WAAL allows the specification of mapping functions. As we have seen in Listing 2.33, the WAAL mapping of e.g., the `login` message takes the first argument of the `login` message to select the username form the dropdown menu, and the second argument is typed into the password textfield. Considering the AAT in Listing 2.32, the first argument is the abstract expression `username(jerry)` and the second argument is `password(jerry)`. For an executable test case, we need to map this abstract expressions to concrete ones. Such mappings of abstract values to concrete values are specified in the `MAPPINGS` section of the WAAL mapping specification. In Listing 2.34, we define two functions — *username* and *password*. The expression `username(tom)` is mapped to the value `Tom Cat (employee)`, whereas the expression `username(jerry)` is mapped to the value `Jerry Mouse (hr)` (Lines 2 to 4). The second function *password* is defined accordingly (Lines 6 to 8).

Listing 2.34: WAAL Mapping: Function Definition

```
1 MAPPINGS {
2   username:
3   abstract:" tom " concrete:"\"Tom Cat (employee)\""
4   abstract:" jerry " concrete:"\"Jerry Mouse (hr)\""
5
6   password:
7   abstract:" tom " concrete:"\"tom\""
8   abstract:" jerry " concrete:"\"jerry\""
9 }
```

Initialization Block

As mentioned before, an executable test case needs to start with an initialization block. Its structure is given in Listing 2.35 and consists of the following components:

Simulated_agents. Using this keyword, a list of agents is specified that need to be simulated by the TEE.

Listing 2.35: WAAL Mapping: Initialization

```
SIMULATED_AGENTS { "jerry" }

AGENTS_CONFIGURATIONS {
  Agent: "webBrowser" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
      Init: {
        Generate::
        Element: Browser
        Action: GoToURL "http://172.16.1.21"
      }
    }
}
```

Agents_configurations. This keyword is used to provide the configuration of each agent. Such a configuration consists of the following information.

- Username and password for a potential basic authentication.
- The IP (browserIP) and port (browserPort) of the Selenium Hub.
- Agents, with which the current agent shares a session.
- The driver used to access the web application (For now, RemoteWebDriver is the only supported driver).
- The platform on which the browser for accessing the web application should be executed (Linux, Mac OS X, Windows, Android, etc.) [14].
- The type of browser and the version that should be used (Firefox, Chrome).
- Finally, the `Init` block specifies initial actions to bring the browser into the start state.

2.7.3. Mapping from WAAL to Executable Source Code

Once the attack trace is translated into WAAL actions, the remaining step to be able to execute the test case is to map these WAAL actions into executable statements. In contrast to the first mapping τ_1 (from abstract messages to WAAL actions) that is application dependent, the second mapping τ_2 (from WAAL to source code) is done once and for all, unless the technologies used by the TEE changes.

[14] Whether the platform, browser, and browser version requirements can be met depends on the configuration of the TEE (see Section 2.7.4.1).

2.7.3.1. General Principle

In this section, we discuss the general principle of how abstract browser actions are mapped to executable API calls. Conceptually, two API interfaces are used in cooperation, even though they operate on different abstraction levels. The first API works at the browser level and is thus close to WAAL, which makes the translation of WAAL actions to this API easier. The second API works directly at the protocol level and is thus close to the Web application communication protocol. The second API is needed only if an action cannot be performed by the first API. In that case, the TEE may request the help of a test expert for providing the corresponding protocol-level message.

In Figure 2.14, there are two kinds of blocks at the source code level: Browser Actions (BA), and Recovery Actions (RA). A BA corresponds to an action performed in a browser. A RA corresponds to a recovery action performed after a failure from a BA. A failure in a BA is either a runtime exception (e.g. a browser element (link, button, dropdown menu, etc.) where an action should be invoked does not exist) or the response of the BA corresponds to a runtime exception triggered at the server side. Depending on the failure that triggers those exceptions, a RA either belongs to the browser or to the protocol level. Furthermore, such a RA may ask a Security Analyst to provide additional information and therefore, it interrupts the automatic execution.

The mapping $\tau_2 : \mathit{GA} \cup \mathit{VA} \rightarrow (\mathit{BA} \times \mathit{RA})^*$ maps each WAAL action $a \in GA \cup VA$ to a sequence of BA and RA with $\tau_2(a) \in (BA \times RA)^*$.

If the TEE can successfully execute every BA, which is done in a fully automatic way, then the verdict is determined as follows: if the actual reactions of the SUV conform to the expected reactions of the test cases — this verification is done by the BA blocks related to VA actions —, the attack has been reproduced and therefore, the test has failed; otherwise, the test has passed.

Runtime Failures During Browser Actions. A BA may fail due to several reasons of different nature. For example, an input element may be disabled, in read only mode, or its `maxLength` attribute may be set to a value smaller than the size of the text to type in. Another example is a button that is disabled or totally missing, and therefore, the BA cannot click on it. For some failures of this kind, it might be possible to execute a recovering action and continue the test execution.

If an error occurs when executing a BA, the TEE changes its operational mode and a RA is executed in order to recover from this error. There are three different ways of recovering after a BA has failed: (i) prepend missing information to the BA and execute it again; (ii) find an alternative way to execute the BA and resume just after it; (iii) move to the protocol level, provide the corresponding message, and resume after the next protocol-level message (which may be after several BA blocks). For the following examples of such recovering methods, the browser level represents HTML elements including their actions, and the protocol level is HTTP.

As an example for the *prepend* case, let BA be an action to check the content of a webpage. This action may fail because the user is not authenticated and first has to provide credentials. In the case of basic access authentication, this request for credentials can be automatically detected and therefore, it is not necessary to provide this step as WAAL actions. Thus, a possible action in RA could be that the TEE asks the test expert for the credentials or that they are read from a configuration file. Then, the TEE reconfigures the used component by adding the credentials and requests the website again. Requesting a website and adding credentials can both be performed at the browser level.

For the *alternative* case, let us consider an HTML button element that triggers an event if the user clicks on it and this event is the execution of a defined JavaScript function. If the HTML button is disabled, the click event can not be triggered by the BA. A possible alternative action, performed by the corresponding RA, is to execute the JavaScript function directly, by using a different API call. Another example is submitting a form to the server. Usually, the end user clicks on the button `Submit`. Therefore, the Security Analyst can specify exactly that action in WAAL. If the specified Submit button is missing, Selenium allows to call the `submit()` method on any element in the form. Therefore, there are alternative ways to submit the form data to the server even if the initial Submit button is missing.

A conceptual example where the TEE has to switch to the protocol (HTTP) level is the following one. Assume that a BA tries to select an element from a list and sends this value to the server by clicking on a button. This action may fail because the element is not present in the list. In that case, the TEE presents some sample HTTP messages to the test expert (e.g., by generating the HTTP messages corresponding to choosing another element from the list). Then, the TEE asks the test expert to provide the correct HTTP message. This message is sent and the BA that follows this HTTP message is executed afterwards. It is worth noting that the underlying assumption when a RA creates some HTTP samples is that the agent state may be restored afterwards. Thus, as soon as the TEE intercepts and drops the HTTP requests, the RA can generate as many samples as possible. During the SPaCIoS EU project, we implemented this functionality as a prototype. In this Ph.D. thesis, we do not further elaborate on that feature since it is not needed for our evaluation examples.

2.7.4. Test Execution Engine (TEE)

The Test Execution Engine is responsible for executing test cases on the SUV. As a reminder, a test case consists of inputs and expected outputs (Figure 2.17). Inputs correspond to stimuli, represented by abstract messages part of the AAT. Therefore, the TEE needs to make *stimuli* operational, called *controlled messages*. At the same time, the TEE needs to *observe* the reaction of the SUV, called *observed messages*, and compare them to the *expected reaction*, represented by *expected messages* in the AAT. Building a *verdict* means to compare the observed messages to the expected reactions.

Passing / Failing a Test. In the literature, a test *passes* if the expected output of the test case can be observed at the SUV. It means that the SUV behaves as the test case specifies the behavior. Contrary, a test case *fails* if the observed output of the SUV does not conform with the expected reaction. To follow the notation in the literature, we assume the following notation:

When the TEE observes that the abstraction of the observed messages conforms with the expected reaction we say that this test case *fails*. Stimuli and expected reactions represent counter examples (negative test cases) since they violate a security property. If such a negative test case can be reproduced on the SUV it means that the SUV does not defend

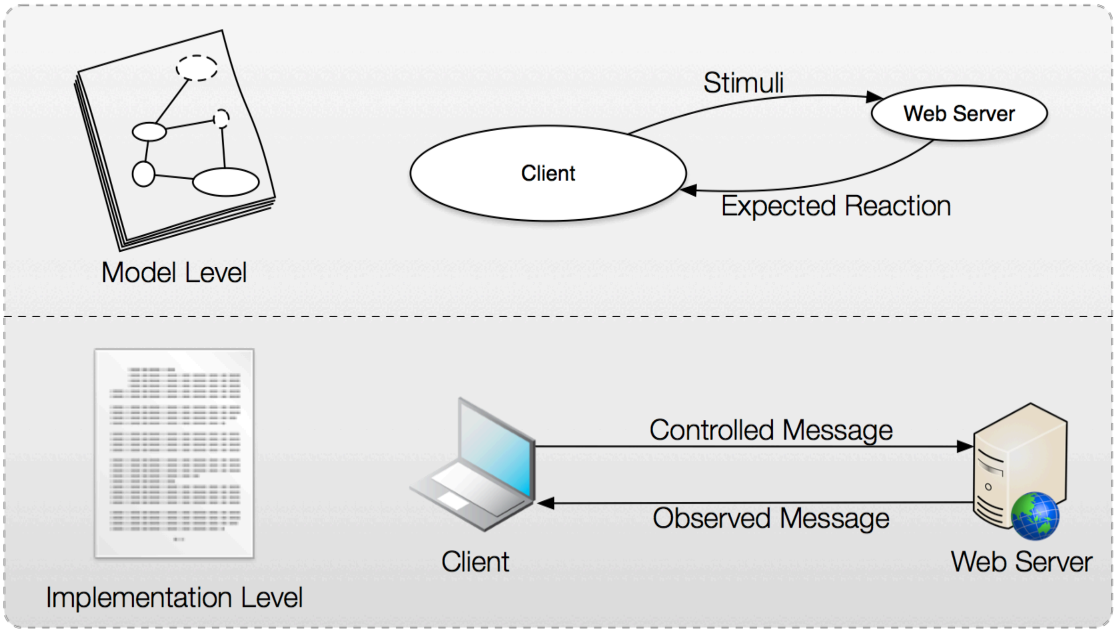


Figure 2.17.: Stimuli and Reactions

against this attack and therefore, the SUV is vulnerable. A failed test case usually attracts the attention of the Security Analyst which is desirable in this situation since the SUV needs to be fixed. Contrary, when the TEE observes that the observed messages does not conform with the expected reaction we say that this test case *passes*. Unfortunately the value of such passed test cases is limited because the Security Analyst cannot learn anything from such test cases. It is unclear why such test case could not successfully be reproduced and the reasons can be manifold. Since testing can only prove the existence but never the absence of a vulnerability the notation conforms with the fact that a passed test case usually does not attract the Security Analyst's attention too much.

2.7.4.1. TestNG and Selenium Grid

The TEE is implemented with TestNG [32] and the Selenium framework [26]. TestNG is a testing framework inspired by JUnit [14] and NUnit [19] and introduces some new functionalities that make it more powerful and easier to use. To make stimuli operational and get access to observed messages sent back by the web application, SPaCiTE makes use of the selenium framework [26]. Selenium is a collection of tools to automate browsers. It consists of Selenium IDE, Selenium Server, Selenium WebDriver, and Selenium Grid. Selenium IDE is a Firefox Browser plugin that allows an end user to record and replay browser actions. It is a complete IDE and therefore, the scriptability is very limited. Selenium IDE is not appropriate for the integration in a automated testing tool. The selenium remote control consists of two parts — the selenium-server and the client. The selenium-server is responsible for automatically launching and stopping browsers. The client libraries are used to control the server. The libraries are available for many different programming languages, like Java, Ruby, Python, Perl, PHP, and so on. Selenium WebDriver integrates the WebDriver API into Selenium. According to the website, *WebDriver is designed to provide a simpler, more concise programming interface in addressing some limitations in the selenium-RC API* [28]. In contrast to Selenium RC, WebDrivers make use of the browser's own native support for automation. Selenium RC uses the same functionality for each supported browser. It achieves this by injecting Javascript into the browser. An advantage of WebDrivers is that they can be used with a local browser, or it communicates via a Selenium Server to a remote browser. Finally Selenium Grid brings Selenium RC to the next level. It is focused on executing test cases in parallel on multiple browsers, by managing multiple environments from a central point. To keep the test case execution environment as flexible and scalable as possible, we make use of the Grid 2 feature of Selenium. This decision is supported by the following arguments:

- In the context of security testing, it is desirable to execute an abstract test case multiple times. This is the case when there exists multiple possibilities to make the abstract test case operational. In addition, an abstract test case describes the logical steps that have to be performed on the SUV but the concrete payload to be used is not part of the abstract test case. The payload consists of the malicious part of the test case. Therefore, the same abstract test case should be executable multiple times with different concrete malicious payloads. Selenium Grid 2 supports parallel execution of test cases by letting clients register with the Selenium Grid Server and act as worker threads. Such clients fetch and execute test cases that are waiting at the Selenium Grid Server. TestNG supports this situation by providing the concept of Data Providers[15] and the possibility to execute test cases in parallel[16].

- Web applications are accessed from many different devices and browsers. Selenium Grid offers the possibility to attach different browsers executed on different operating systems very easily. Therefore, the effort to add a browser of a new vendor, a different browser version, or adding browsers executed on mobile devices is very low. The operational test cases need to be parameterized by three options that specify the desired browser, browser version, and the operating system the browser should be executed.

Therefore, the test execution engine of SPaCiTE integrates Selenium Grid as an underlying execution framework and offers the necessary configuration possibilities to administer Selenium Grid.

A typical Selenium Grid 2.0 architecture is given in Figure 2.18. The Selenium hub is at the center of the architecture and brings test case generation and text case execution together. It orchestrates the test execution on all the available Selenium nodes. Each node can have separate characteristics. On a high level, a node registers itself at the hub with information about the operating system, browser vendor, and browser version. Using Selenium Grid, a test case specifies on which platform it has to be executed. Thus, the selenium hub is responsible for forwarding the test case to a matching selenium node and collecting the verdict.

[15] http://testng.org/doc/documentation-main.html#parameters-dataproviders

[16] http://testng.org/doc/documentation-main.html#parallel-running

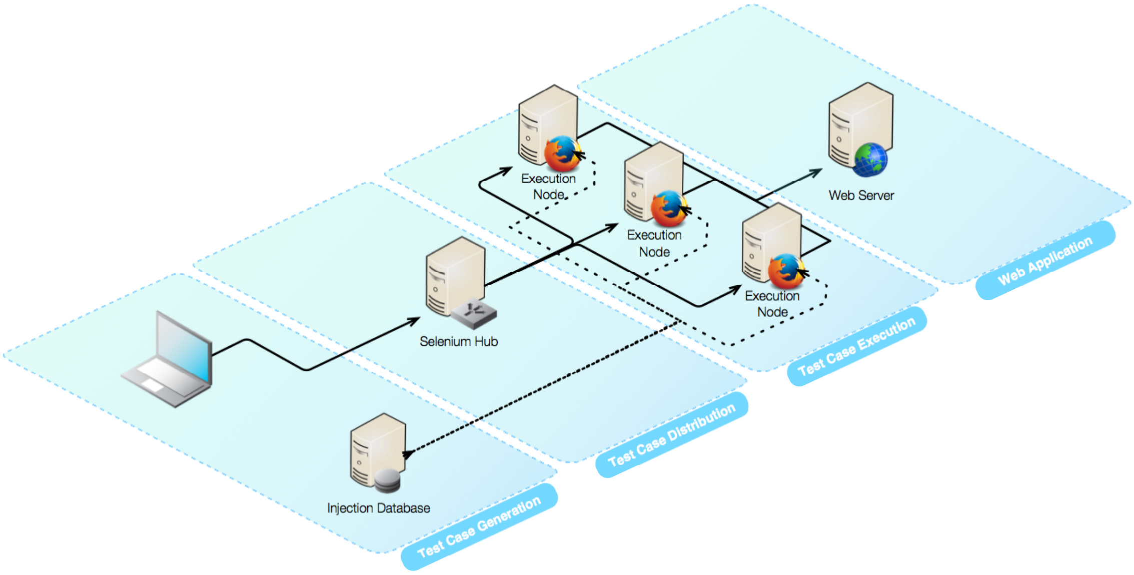


Figure 2.18.: Selenium Grid 2.0 Architecture

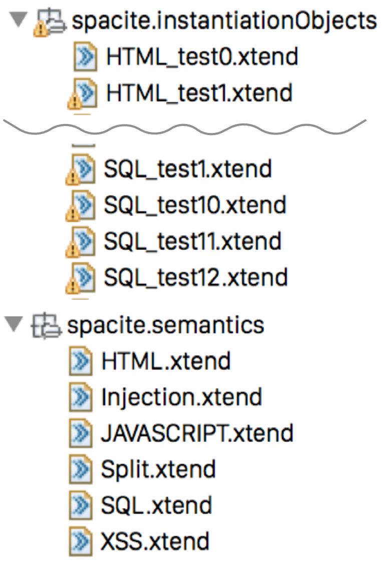
2.7.4.2. Instantiation Library

To execute operational test cases, we need concrete malicious inputs and verification mechanisms. Since these low level details are not present in the abstract model, SPaCiTE makes use of an instantiation library that provides such low level information. Abstractly speaking, the instantiation library is a set of objects, where each object provides malicious inputs and a verification method.

The elements of the instantiation library are implemented as Java Class files stored on a filesystem and belong to the package `instantiationObjects` (see Figure 2.19). Each such object needs to inherit from the abstract class `Injection` that requires the presence of two methods `getInputValues()` and `verify()`, given in Listing 2.36. To query the element for the concrete malicious input, the method `getInputValues` is invoked. Similar, to check if the injected malicious code is successfully executed by the web application, the method `verify` is invoked on the same instantiation element as used to retrieve the malicious input.

The Semantic Mutation Operator that was used to generate an AAT represents a class of source code vulnerabilities and therefore, it determines which instantiation elements have to be used during the execution of the attack trace. To allow SPaCiTE to automatically determine the semantics of the malicious input, several semantic keywords are defined in the package `semantics`. Technically, they represent interfaces so that an element can implement several of them. E.g., if a malicious input is dedicated to a split XSS attack, the corresponding instantiation element implements the interfaces `Split` and `XSS`.

Whenever SPaCiTE executes a test case, the AAT indicates the type of concrete malicious input that are needed. Since the instantiation library follows a flat structure, all instantiation elements are stored in the same package `instantionObjects`. Therefore, SPaCiTE filters the available elements with respect to the interfaces that they implement.



```
▼ spacite.instantiationObjects
      HTML_test0.xtend
      HTML_test1.xtend
      SQL_test1.xtend
      SQL_test10.xtend
      SQL_test11.xtend
      SQL_test12.xtend
▼ spacite.semantics
      HTML.xtend
      Injection.xtend
      JAVASCRIPT.xtend
      Split.xtend
      SQL.xtend
      XSS.xtend
```

Figure 2.19.: Instantiation Library: Filesystem Layout

SPaCiTE implements this filter functionality dynamically with the help of Java reflection mechanisms. This decision allows the Security Analyst to easily and dynamically add new instantiation elements to the library without changing the implementation of SPaCiTE.

After SPaCiTE identified all matching instantiation elements for a given AAT, SPaCiTE automatically adds them to a TestNG data provider[15]. If a test case is executed together with a data provider, TestNG executes the corresponding test case with each data element of the data provider.

Listing 2.36: Instantiation Object Interface

```
package interfaces

abstract class IInjection {
        // This method returns the malicious inputs
        public def Object[] getInputValues();

        // This method verifies if the effect of the malicious input is observe
        public def boolean verify();
}
```

Concrete Malicious Input Values. Concrete malicious input values can be collected from different sources. OWASP published a XSS Filter Evasion Cheat Sheet [22] that consists of a massive list of malicious XSS inputs. Conceptually, every entry in this list needs to be converted into an instantiation library element once and for all. The transformation requires two steps — (1) implementing the `getInputValues` method with the malicious input from the list, and (2) the verification method `verify` that checks the effect of a successful injection. While the first part can be automated, the second step is manual. The elements in such an instantiation library are motivated by the vulnerabilities discussed in Sections 2.6.2 and 2.6.3. In the following we show an exemplary element in our instantiation library. Listing 2.37 represents an instantiation element that is used whenever SPaCiTE needs to inject malicious XSS code. This semantics is provided to SPaCiTE by the two interfaces that this Java class implements. SPaCiTE calls the method `getInputValues.get(0)` to get the malicious code at Lines 5 to 10 in Listing 2.37. Whenever this malicious code is executed, it creates a new `div` element that contains the text `THISISANXSSATTACK`. Furthermore, this new `div` element is appended to the `body` element of the webpage. SPaCiTE verifies the success of the attack by invoking the method `verify` on the same object as used for the injection. Line 15 verifies if indeed such a `div` element exists on the webpage.

Listing 2.37: XSS Instantiation Library Element

```
1  class HTML_test0 extends Injection implements XSS, HTML {
2
3    override getInputValues() {
4      #[
5        "<SCRIPT>
6        var el=document.createElement('div');
7        el.setAttribute('id', 'THISISANXSSATTACK');
8        el.innerHTML='THISISANXSSATTACK';
9        document.getElementsByTagName('body')[0].parentNode.appendChild(el);
10       </script>"
11     ]
12   }
13
14   override verify( Connection conn ) {
15     return conn.elementIsPresentByXPath("//div[@id='THISISANXSSATTACK']")
16   }
17 }
```

3. Tool Support

SPaCiTE is the tool that implements the conceptual work. During the SPaCIoS EU project we developed a first version of SPaCiTE and significantly improved it after the project. SPaCiTE goes far beyond a prototype implementation and therefore, this Ph.D thesis not only contributes in terms of conceptual work but also with a tool usable in practice. Besides Selenium and TestNG, it is mainly based on the Xtext [48] framework, a modern way for implementing Domain Specific Languages (DSLs). For both the ASLan++ mutation operators, as well as for the Web Application Abstract Language (WAAL) mapping language, we used Xtext to develop a modern IDE for these languages. For the code generation part, we used the tightly integrated Xtend [47] language. It provides template expressions [44] that make code / model generation very straight forward.

To put our tool in context, we will discuss related tools in Chapter 6 and provide a high level summary here. At the ASLan++ level, we are not aware of any other tool that mutates ASLan++ formal specifications. For the High Level Protocol Specification Language (HLPSL), Dadeau et al. [86] developed a tool that focuses on mutating such specification but lacks a Test Execution Engine (TEE) component including the possibility to map AATs to executable test cases. Lebeau et al. [144] provide a tool chain based on UML diagrams based on an existing MBT software of the company Smartesting called CertifyIt [66, 73]. Blome et al. [70] developed VERA, a flexible model-based vulnerability testing tool. It is based on attacker models and abstracts from low-level implementation details like HTTP requests. At the same time, there exists a huge list of both black-box and white-box vulnerability scanners for web applications. While white-box scanners operate on source- or byte code, black-box scanner contain a crawler to discover the System Under Validation (SUV). As examples, tools in this category are SAGE: Whitebox Fuzzing for Security Testing [112], the Ardilla tool [141], the Apollo tool [61], Acunetix [5], AppScan [6], Burp [10], Grendel-Scan [12], Milescan [15], N-Stalker [17], NTOSpider [18], Paros [25], W3af [34], etc. SPaCiTE differs to SAGE, Ardilla, and Apollo because they all need access to source code of the web application, whereas SPaCiTE does not. Furthermore, source-code based tools are often dedicated to specific programming languages, like Ardilla is to PHP applications. SPaCiTE in contrast, is programming language independent. SPaCiTE differs to black-box vulnerability tools because the behavioral description of the web application is provided to SPaCiTE in form of a formal specification, whereas black-box scanners need to learn the behavior by crawling the web application. While black-box scanners only test parts of the web application that are discovered by the crawler, SPaCiTE only tests those parts that are modeled. Thus, extending a formal specification to cover missed parts is probably easier than bug fixing and extending a web crawler.

In this section, we focus on the use of SPaCiTE and do not discuss implementation details. We describe a step-by-step tutorial, how a Security Analyst uses SPaCiTE in practice to generate and execute security test cases for web applications. SPaCiTE supports the Security Analyst with the following tasks:

Easy Installation and Configuration Procedure. We contribute with an update site for the Eclipse platform to effortlessly integrate SPaCiTE into Eclipse. SPaCiTE preferences are integrated into the standard Eclipse preference framework.

Creating SPaCiTE Projects. SPaCiTE contributes with an Eclipse wizard to automatically create and configure a SPaCiTE project. All dependencies to external libraries e.g., for test case execution are automatically resolved.

Modeling Web Application. To support the Security Analyst during the modeling phase, SPaCiTE contributes with a modern IDE with code completing, code templates, syntax highlighting, quick fixes, and so on. In particular, the IDE helps the Security Analyst during semantically annotating a specification.

Model Checking. Instead of switching the context to model check an ASLan++ specification, the model checker can be started using a single click in SPaCiTE.

WAAL Mappings. Since the WAAL mapping is tightly bound to a formal specification, SPaCiTE contributes with an automatic generation of an initial skeleton dedicated to the current formal specification.

Applying Mutation Operators. Although we do not provide a dedicated language to develop mutation operators, SPaCiTE contributes with a dynamic loading procedure to integrate and apply mutation operators to a formal specification. Newly created mutation operators are automatically recognized by SPaCiTE and can be applied without changing or recompiling SPaCiTE.

Instantiation and Executing Test Cases. SPaCiTE contributes with an automatic procedure to apply the WAAL mapping to AATs and generate TestNG test cases written in Java. In a SPaCiTE project, such test cases can easily be executed with one click.

3.1. Installing and Configuring SPaCiTE

Before a Security Analyst can use SPaCiTE it needs to be installed an configured. SPaCiTE is provided as an Eclipse 4.4 plug-in and can easily be installed using an update site. The installation is straight forward by performing the following steps:

1. Start Eclipse Luna (Version 4.4), click on **Help** ⟶ **Install New Software...** → **Add...** and add the following updateSites:
   - TestNG framework, available at `http://beust.com/eclipse`
   - SPaCiTE, available at `http://updatesite.spacite.matt-buechler.com`
2. After restarting Eclipse, you need to configure SPaCiTE first. Therefore, perform the following actions:
   - Select **SPaCiTE Editor** → **Install external tools** to install external dependencies. It will download the latest version of the Cl-Atse model checker and the ASLan++ Connector Binary[1].

[1] For this Ph.D thesis, we used Cl-Atse in version 2.5-21 and the ASLan++ Connector Binary in version 1.4.9.

- Select **Eclipse** → **Preferences** → **SPaCiTE** → **SPaCiTE preferences** and specify the following values:

  Java Location. Usually, the Java binary is located in /usr/bin. For this Ph.D thesis, we used Java version 1.8.

  M4 Executable. Usually, the m4 executable is available at /usr/bin/m4.

  Directory for Custom Mutation Operators. This refers to the directory where custom mutation operators are stored. If the Security Analyst does not develop own mutation operators, the entry may point to any directory.

  ASLan++ Connector Binary. This Java Jar file was downloaded during the command *Install external tools* and is available at the directory where the external tools are installed. The needed binary has the a filename of the form `aslanpp-connector-[version].jar`.

  Cl-Atse Binary. This executable is available at the directory where the external tools are installed. For the Mac OS X, the binary is called `cl-atse_x86_64-mac`.

  Cl-Atse Options. For the models we consider in this Ph.D thesis, the following options are used: `--nb 2 --not_hc --free --lvl 2 --short`.

  Directory Name for Mutated Models. This directory name is used by SPaCiTE to store the mutated models. In general, there is no need to change the default value.

  Delete Intermediate Files. During the process of model checking, intermediate files are created. If this option is enabled, they are deleted afterwards.

  Debug. If enabled, SPaCiTE outputs debug information in the log view.

  Max Number of Threads. Certain tasks are parallelized. The maximum number of threads is an upper limit of concurrent threads used by SPaCiTE.

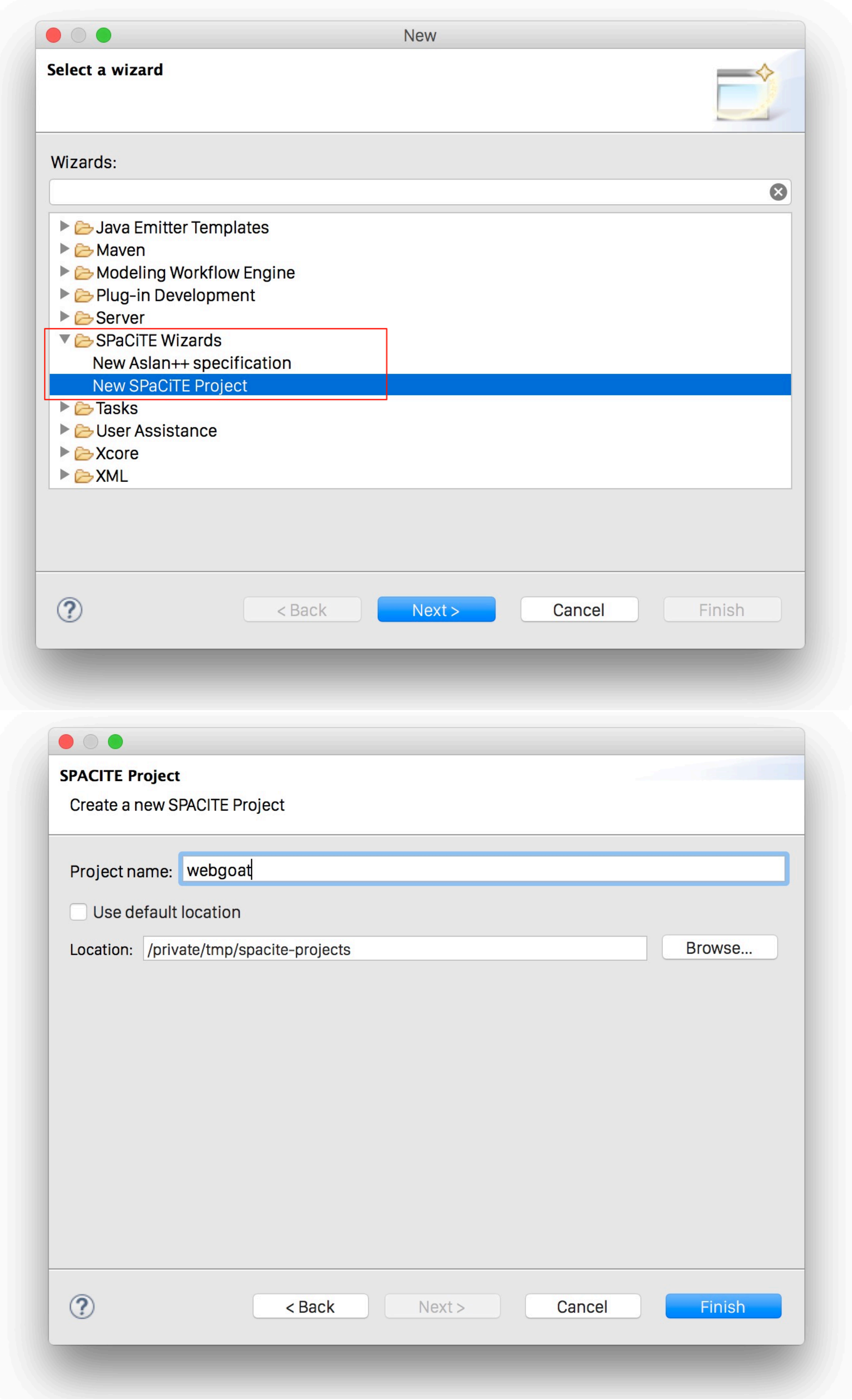
  Modelchecking Timeout in Seconds. A mutated model might be a long-running model. To stop the model checking after a certain amount of time, the length of the model checking time is specified here.

3.2. Creating a SPaCiTE Project

As a first step after the installation and configuration of SPaCiTE (see Section 3.1) the Security Analyst needs to create a SPaCiTE project. It is created by the command **File** → **New** → **Other...** → **SPaCiTE Wizards** → **New SPaCiTE Project**. After providing a name and a location for the new project, SPaCiTE automatically creates and configures the project, including a skeleton template for the formal specification (see Figure 3.1).

The newly created model is incomplete and therefore, several error messages are shown. We address two of them as follows. The formal specification has a name which follows after the keyword **specification** in line 1 in Figure 3.2. For the channel model[2], we have several options available. By placing the cursor after the keyword **channel_model** and pressing the

[2] See Avantssar Deliverable D2.3 (update) [62] for details.

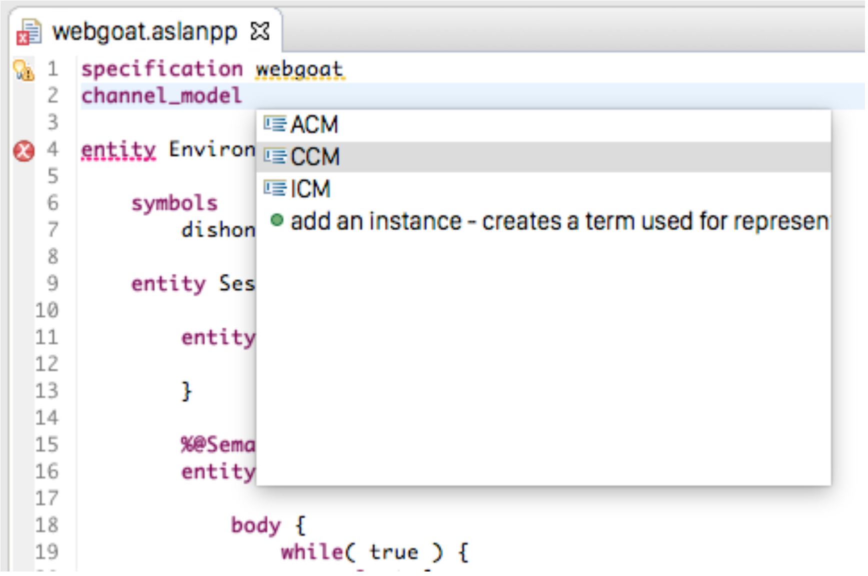


Project name: webgoat

☐ Use default location

Location: /private/tmp/spacite-projects Browse...

Figure 3.1.: SPaCiTE: Create a New Project



```
specification webgoat
channel_model

entity Environ

    symbols
        dishon

    entity Ses

        entity

        }

        %@Sema
        entity

            body {
                while( true ) {
```

Figure 3.2.: SPaCiTE: Select a Channel Model

autocompletion shortcut (usually `ctrl-space`), we get a list of alternative values. For all models considered in this Ph.D thesis, we choose **CCM** (see Figure 3.2).

3.3. Modeling the Web Application

Upon creating a SPaCiTE project, an empty ASLan++ model is created and opened. In the newly created file the Security Analyst specifies the formal specification of the web application using the ASLan++ language. This language was developed during the AVANTSSAR project and von Oheimb and Mödersheim [218] published the formal specification. An extended description and a ASLan++ tutorial can be found in the AVANTSSAR deliverable 2.3 [62].

SPaCiTE has a built-in ASLan++ editor that supports the Security Analyst modeling the web application. As described in Section 3.2 a basic skeleton is automatically created when a new SPaCiTE project is created. Furthermore, SPaCiTE provides the following specific features:

Auto-Completion. SPaCiTE provides the usual auto-completion feature, known for many other language.

Code-templates. Since modeling web applications very often requires the same code fragments, basic skeletons for these language constructs can be added by using the auto-completion feature. SPaCiTE provides templates for so called `SelectOn` statements, and a fact called `instance_[type]` that is used to uniquely create data elements (see Section 2.3.3). Both of these templates consist of parts that the Security Analyst needs to adapt. Using `TAB`, the Security Analyst automatically navigates through the missing variables in the code template.

Semantic Highlighting. Since transmitted messages are core elements when modeling a web application, `Transmissions` are highlighted so that they are better identifiable for the Security Analyst.

For modeling details, please refer to Section 2.3.3, where we provided a tutorial how to model web applications so that the models are useful in terms of SPaCiTE.

3.4. Model Checking the Formal Specification

When the Security Analyst has finished the formal specification of the web application, he has to make sure that the model is 'consistent'. This step can be performed with SPaCiTE by selecting **SPaCiTE Editor → Model check specification**.

3.5. Binding the Model to a WAAL Mapping

Upon having a SPaCiTE-conform model, the Security Analyst has to provide a mapping of the abstract messages to abstract browser actions. If this link is missing in a formal specification, the SPaCiTE editor will issue a warning, as shown in Figure 3.3. To resolve the issued warning, the SPaCiTE editor provides quickfixes. The Security Analyst therefore clicks on the warning sign at the beginning of the line (see red circle in Figure 3.4) and the SPaCiTE editor suggest context specific actions as quickfixes. In Figure 3.3 several warnings are issued and therefore, also several quickfixes are suggested. In order to address the warning that the specification is not bound to a WAAL mapping, the Security Analyst selects the quickfix[3] `Add WAAL mapping annotation`. The consequence of this step is that a link is added to the formal specification (see Figure 3.5).

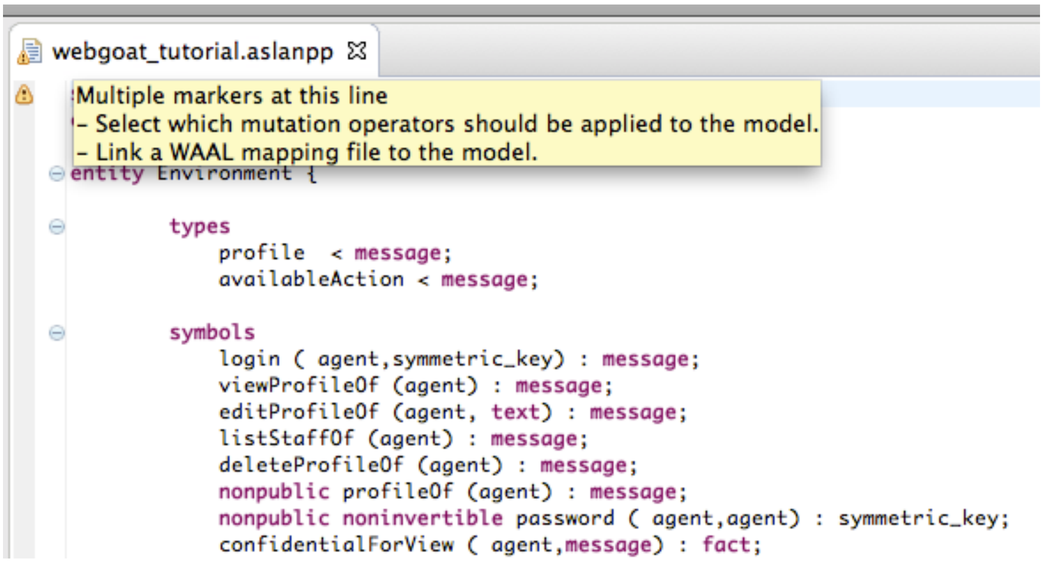
In addition, a skeleton of a WAAL mapping file is created and opened. An example of a WAAL mapping skeleton file is shown in Figure 3.6. The skeleton consists of several parts. The first one specifies the simulated agents and is automatically specified according to the `%@Semantics[Test]` annotation in the SPaCiTE-conform model. The second part is the configuration of the different agents. The skeleton is syntactically incomplete and the editor marks the locations that have to be completed by the Security Analyst. In particular, the Security Analyst has to provide the following information:

- The IP and the port of the browser, SPaCiTE can use to execute the test cases.
- The sessions a particular agent is involved with.

The other specifications are default values and might be updated by the Security Analyst if needed.

The third part consists of the mappings of abstract messages to browser actions. This block consists of a set of `Message@Actions` specifications (see Figure 3.6). Each such specification is a triple, consisting of the `abstract` action, the `generation` block, and the `verification` block. The skeleton of such a triple is automatically generated by

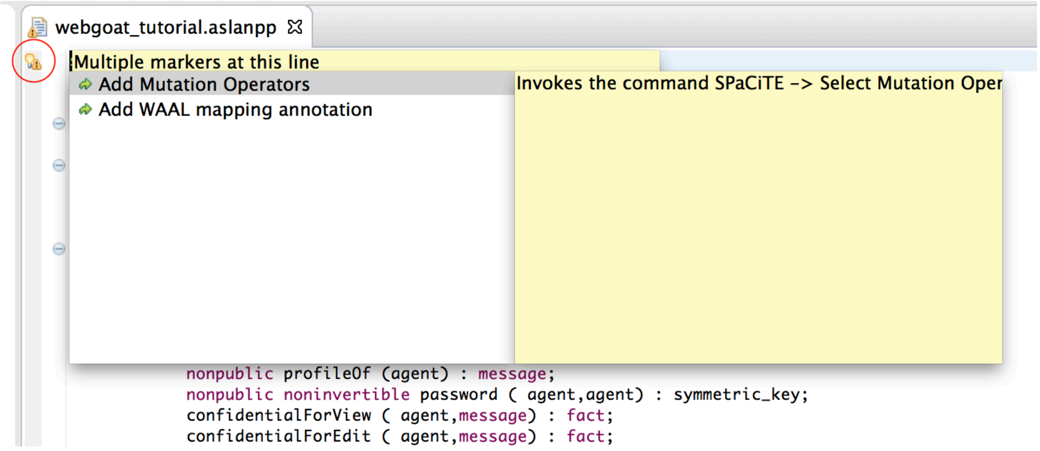
[3] Alternatively the Security Analyst makes sure that the specification he wants to provide a mapping for, is currently opened. He then selects **SPaCiTE Editor → Link model to waalmapping**



Multiple markers at this line
- Select which mutation operators should be applied to the model.
- Link a WAAL mapping file to the model.

```
entity Environment {

        types
            profile  < message;
            availableAction < message;

        symbols
            login ( agent,symmetric_key) : message;
            viewProfileOf (agent) : message;
            editProfileOf (agent, text) : message;
            listStaffOf (agent) : message;
            deleteProfileOf (agent) : message;
            nonpublic profileOf (agent) : message;
            nonpublic noninvertible password ( agent,agent) : symmetric_key;
            confidentialForView ( agent,message) : fact;
```

Figure 3.3.: SPaCiTE Editor: Validation That Linking to a WAAL Mapping File Is Present

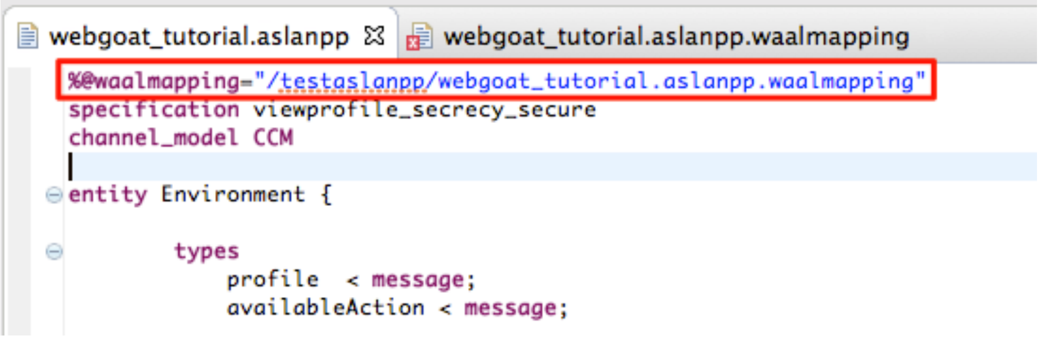


Multiple markers at this line

Invokes the command SPaCiTE -> Select Mutation Oper

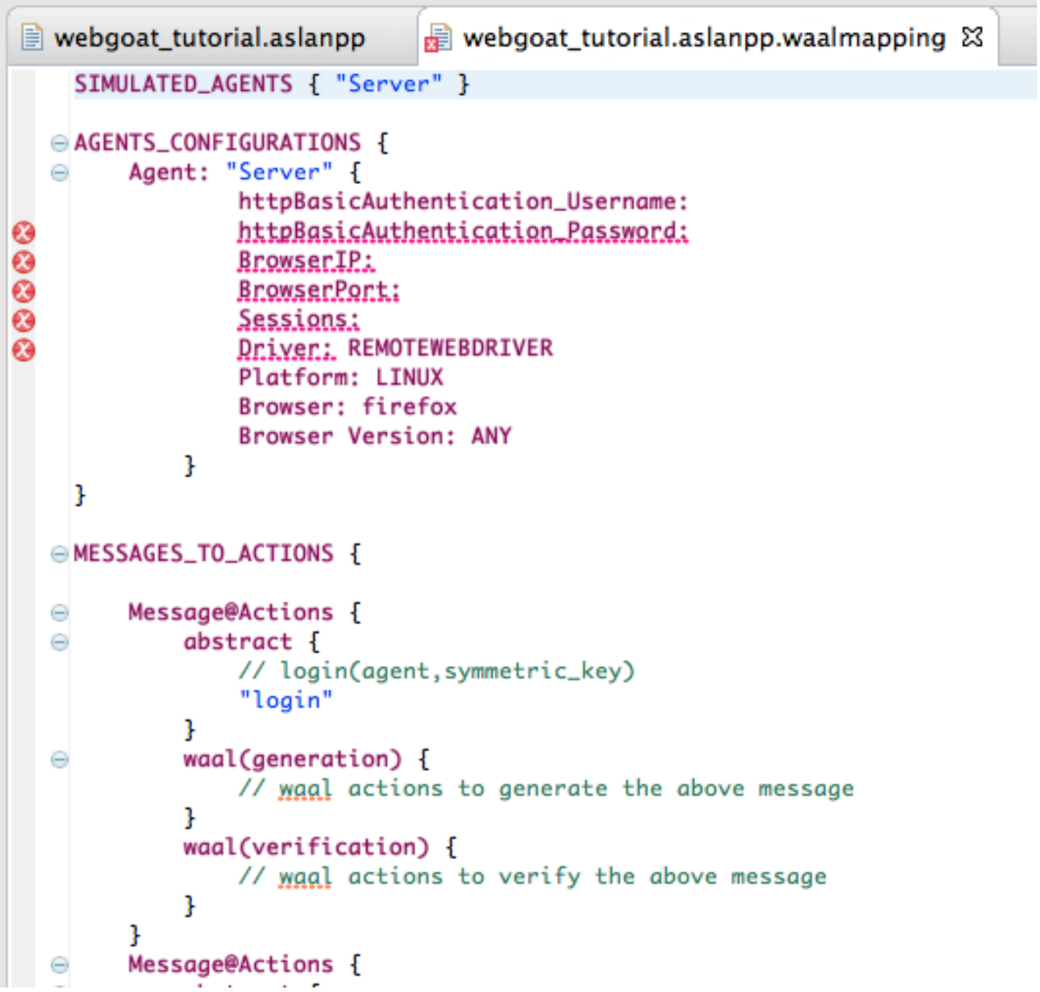
```
            nonpublic profileOf (agent) : message;
            nonpublic noninvertible password ( agent,agent) : symmetric_key;
            confidentialForView ( agent,message) : fact;
            confidentialForEdit ( agent,message) : fact;
```

Figure 3.4.: SPaCiTE Editor: Quickfixes For Issued Warnings



```
%@waalmapping="/testaslanpp/webgoat_tutorial.aslanpp.waalmapping"
specification viewprofile_secrecy_secure
channel_model CCM

entity Environment {

        types
            profile  < message;
            availableAction < message;
```

Figure 3.5.: SPaCiTE Editor: Linking Model to a WAAL Mapping File



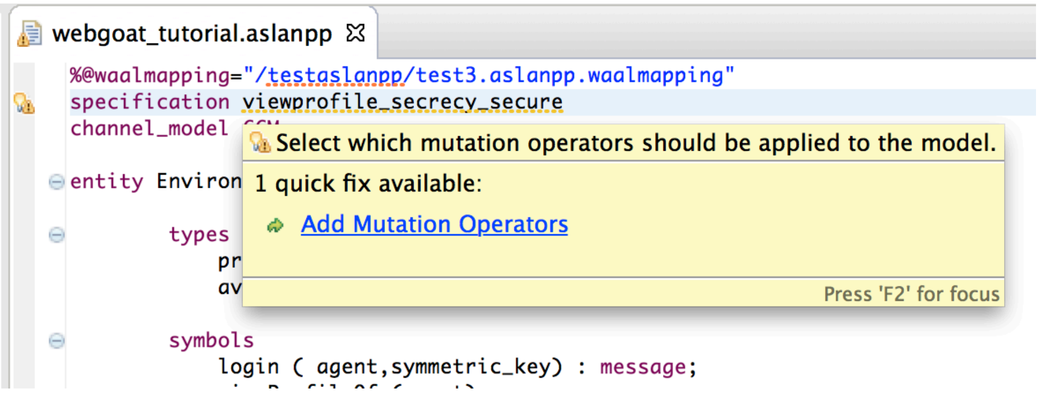
```
SIMULATED_AGENTS { "Server" }

AGENTS_CONFIGURATIONS {
    Agent: "Server" {
            httpBasicAuthentication_Username:
            httpBasicAuthentication_Password:
            BrowserIP:
            BrowserPort:
            Sessions:
            Driver: REMOTEWEBDRIVER
            Platform: LINUX
            Browser: firefox
            Browser Version: ANY
        }
}

MESSAGES_TO_ACTIONS {

    Message@Actions {
        abstract {
            // login(agent,symmetric_key)
            "login"
        }
        waal(generation) {
            // waal actions to generate the above message
        }
        waal(verification) {
            // waal actions to verify the above message
        }
    }
    Message@Actions {
```

Figure 3.6.: SPaCiTE Editor: WAAL Mapping Skeleton

the SPaCiTE editor based on the available messages in the SPaCiTE-conform model. If the particular message has to be generated, the Security Analyst specifies the corresponding abstract browser actions within the `waal(generation)` block. If the particular message has to be verified, the Security Analyst specifies the corresponding abstract browser actions within the `waal(verification)` block.

3.6. Selecting Mutation Operators

For every formal specification, the Security Analyst needs to specify which mutation operators SPaCiTE should apply. If the mutation operators are not specified, the SPaCiTE editor issues a warning, as shown in Figure 3.7. To specify which mutation operators are used, the Security Analyst clicks on the warning sign to let the SPaCiTE editor suggest quickfixes. To address the warning, the Security Analyst selects the quickfix `Add Mutation Operators` (see Figure 3.7). Alternatively, the Security Analyst selects the command



```
%@waalmapping="/testaslanpp/test3.aslanpp.waalmapping"
specification viewprofile_secrecy_secure
channel_model

entity Environ

        types
            pr
            av

        symbols
            login ( agent,symmetric_key) : message;
```

Select which mutation operators should be applied to the model.

1 quick fix available:

Add Mutation Operators

Press 'F2' for focus

Figure 3.7.: SPaCiTE Editor: Quickfix to Add Mutation Operators

`SPaCiTE Editor → Select mutation operators` in the menu. Upon this command, a dialog window with all available mutation operators are displayed. Each entry in this dialog can be selected or deselected (see Figure 3.8). Upon clicking on `OK` the selected mutation operators are added to the formal specification, as shown in Figure 3.9.

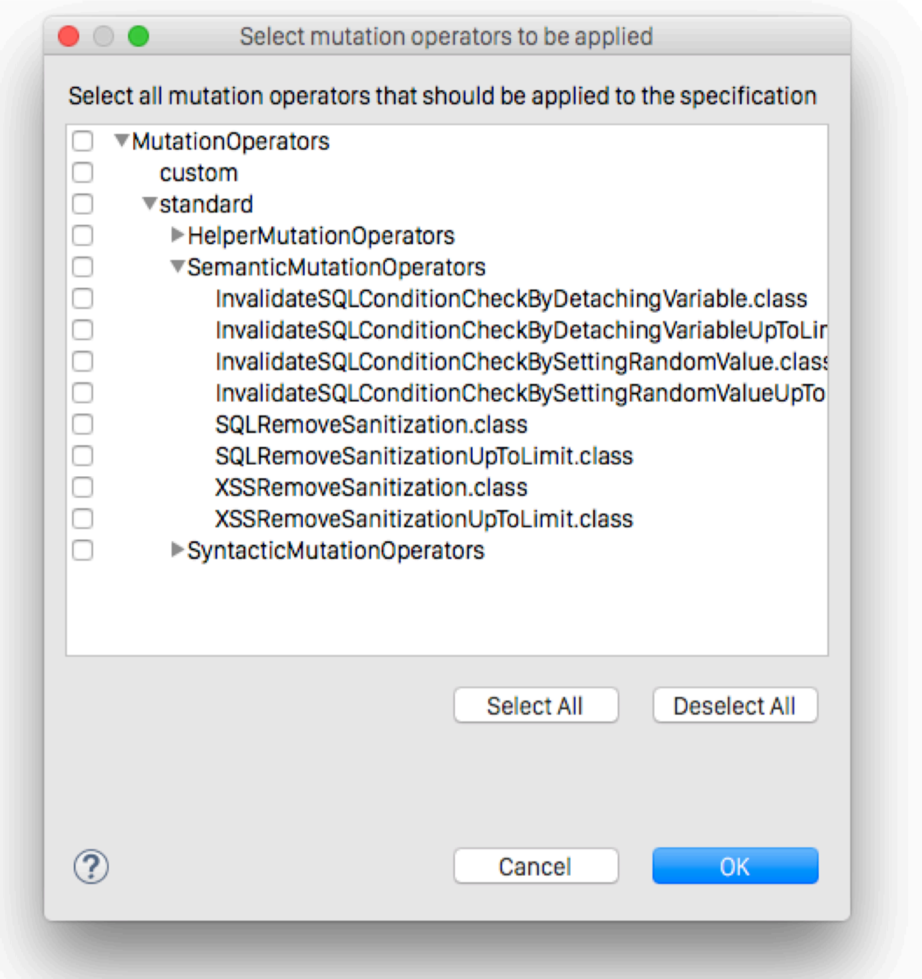
3.7. Model Checking All Mutated Specifications

After the mutation operators are selected, the selection is added as the first line to the formal specification. Upon saving the specification, SPaCiTE automatically applies the mutation operators and generates the mutated models. Therefore, there is no dedicated action to apply the mutation operators[4]. In general, applying the mutation operators finishes in seconds. In exceptional cases, where the formal specification and the applied mutation operator lead to a huge combinatorial number of mutated models, this process might take several minutes. SPaCiTE has created a directory where all the mutated specifications are stored. The exact location was specified during the configuration of SPaCiTE in Section 3.1. All specifications in this directory can be model checked at once. The Security Analyst opens the SPaCiTE-conform model from which all the mutated specifications were generated and clicks on **SPaCiTE Editor** → **Model Check All Mutated Models** (see Figure 3.10). In case some mutated specifications cannot be model checked, the SPaCiTE editor automatically opens the `Error Log` view and logs the corresponding errors (see Figure 3.11).

3.8. Selenium Grid Hub

As discussed in Section 2.7.4 we make use of the Selenium framework [26] for the underlying testing infrastructure. The central server is operated by the selenium hub, a component that is responsible for managing different selenium nodes and distributing operational test cases to them. To install and start the hub, the following commands (see Listing 3.1) are provided. The corresponding makefile is shown in Appendix E.1.

[4] Nevertheless, to enforce it the application of the mutation operator, one might modify the file and store it. This action will trigger the generation of mutated models.

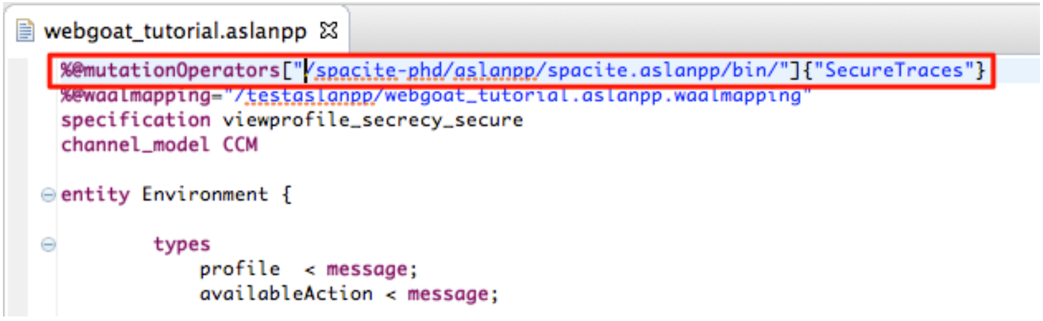


Select mutation operators to be applied

Select all mutation operators that should be applied to the specification

- □ ▼MutationOperators
  - □ custom
  - □ ▼standard
    - □ ▶HelperMutationOperators
    - □ ▼SemanticMutationOperators
      - □ InvalidateSQLConditionCheckByDetachingVariable.class
      - □ InvalidateSQLConditionCheckByDetachingVariableUpToLir
      - □ InvalidateSQLConditionCheckBySettingRandomValue.class
      - □ InvalidateSQLConditionCheckBySettingRandomValueUpTo
      - □ SQLRemoveSanitization.class
      - □ SQLRemoveSanitizationUpToLimit.class
      - □ XSSRemoveSanitization.class
      - □ XSSRemoveSanitizationUpToLimit.class
    - □ ▶SyntacticMutationOperators

Figure 3.8.: SPaCiTE Editor: List of Available Mutation Operators



webgoat_tutorial.aslanpp

```
%@mutationOperators["/spacite-phd/aslanpp/spacite.aslanpp/bin/"]{"SecureTraces"}
%@waalmapping="/testaslanpp/webgoat_tutorial.aslanpp.waalmapping"
specification viewprofile_secrecy_secure
channel_model CCM

entity Environment {

        types
            profile  < message;
            availableAction < message;
```

Figure 3.9.: SPaCiTE Editor: Selected Mutation Operators to be Applied

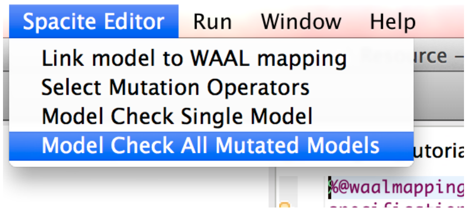
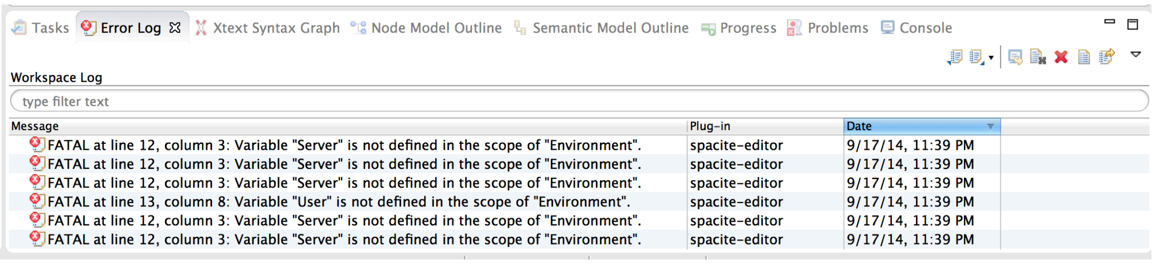

Spacite Editor
Run
Window
Help
Link model to WAAL mapping
Select Mutation Operators
Model Check Single Model
Model Check All Mutated Models
%@waalmapping

Figure 3.10.: SPaCiTE Editor: Model Check All Mutated Models



Message	Plug-in	Date
FATAL at line 12, column 3: Variable "Server" is not defined in the scope of "Environment".	spacite-editor	9/17/14, 11:39 PM
FATAL at line 12, column 3: Variable "Server" is not defined in the scope of "Environment".	spacite-editor	9/17/14, 11:39 PM
FATAL at line 12, column 3: Variable "Server" is not defined in the scope of "Environment".	spacite-editor	9/17/14, 11:39 PM
FATAL at line 13, column 8: Variable "User" is not defined in the scope of "Environment".	spacite-editor	9/17/14, 11:39 PM
FATAL at line 12, column 3: Variable "Server" is not defined in the scope of "Environment".	spacite-editor	9/17/14, 11:39 PM
FATAL at line 12, column 3: Variable "Server" is not defined in the scope of "Environment".	spacite-editor	9/17/14, 11:39 PM

Figure 3.11.: SPaCiTE Editor: Error Log View

Listing 3.1: Installing and Executing Selenium Grid Hub

```
cd selenium-grid-hub
make install
make run
```

The Selenium hub provides the infrastructure so that several nodes can be registered. Each node tells the hub which operating system it runs, the browser vendor and version it has installed, and how many parallel instances the node manages to execute.

The configuration of such a node is given as a JSON file. An example is shown in Listing 3.2. It expresses that it provides a firefox browser (Line 4) version 31 (Line 5), running on a Linux (Line 6). Furthermore the node is powerful enough to execute 5 parallel instances of the browser (Line 7). The protocol to manage the browser is WebDriver (Line 3). The corresponding Selenium hub that manages this node is available on host `localhost` (Line 14) at the port `4444` (Line 13).

Listing 3.2: JSON configuration of Selenium Grid Node

```
1  {   "capabilities": [
2           {
3             "seleniumProtocol": "WebDriver"
4             "browserName": "firefox",
5             "version": "31",
6             "platform": "LINUX",
7             "maxInstances": 5
8           },
9         ],
10     "configuration": {
11          "nodeTimeout":120,
12          "port":5555,
```

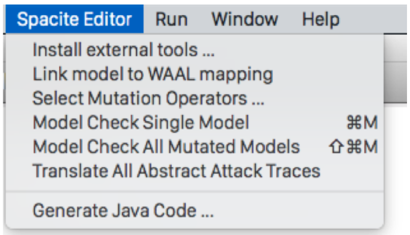

Spacite Editor
Run
Window
Help
Install external tools ...
Link model to WAAL mapping
Select Mutation Operators ...
Model Check Single Model ⌘M
Model Check All Mutated Models ⇧⌘M
Translate All Abstract Attack Traces
Generate Java Code ...

Figure 3.12.: SPaCiTE Editor: Generate Java Code

```
        "hubPort":4444,
        "hubHost":"localhost",
        "nodePolling":2000,
        "registerCycle":10000,
        "register":true,
        "cleanUpCycle":2000,
        "timeout":30000,
        "maxSession":1,
        }
}
```

Listing 3.3: Registering a New Selenium Grid Node CL

```
cd selenium-grid-node
make install
make register
```

These configuration options can directly be provided at the command line as well. An example is given in Appendix E.2. A new node can be registered by typing the commands given in Listing 3.3. At any time, the current state of the hub and the currently registered nodes can be inspected by going to the URL `http://[selenium-hub]:4444/grid/console`[5].

3.9. Turning AAT Operational and Execution of Attack Traces

The translation of AATs to operational test cases and their execution is straight forward. The Security Analyst opens the SPaCiTE-conform model from which the mutated models were generated and selects **SPaCiTE Editor** → **Translate all Abstract Attack Traces** and afterwards **Generate Java code...** (Figure 3.12). Issuing these commands, SPaCiTE applies the linked WAAL mapping described in Section 3.5 to all AATs and generates executable attack traces in form of Java classes. The Security Analyst executes the general TestNG test classes by right clicking on them and selecting **Run as** → **TestNG Test**.

[5] Substitute the expression `[selenium-hub]` with the corresponding IP address or DNS name of the Selenium Grid Hub.

4. Evaluation

4.1. Strategy and Metrics for the Evaluation

In the previous chapters, we have introduced the overall methodology to generate security-interesting test cases. In this chapter, we discuss and evaluate our approach. Our evaluation is motivated not only from a Security Analyst's but from a scientific point of view as well.

For clarity's sake, the hypothesis of this Ph.D thesis is:

> In a model-based context, non-trivial security vulnerabilities for web applications can be found with mutation operators and fault injections.
>
> Sub-hypothesis:
>
> Using Semantic Mutation Operators is more efficient than using Syntactic Mutation Operators.

The hypothesis claims that in a model-based context, non-trivial security vulnerabilities for web applications can be found with mutation operators and fault injections. To evaluate this hypothesis, we consider both theoretical and practical aspects. To demonstrate that SPaCiTE indeed finds non-trivial security vulnerabilities, it can be compared to different types of tools. Table 4.1 presents a possible classification and corresponding characteristics. Discussed aspects are:

(i) the input artifact the approach generates test cases from, (ii) the message parameters that are used for malicious injections, (iii) the source from which concrete malicious payloads are taken or generated, (iv) filter criteria that further select concrete malicious payloads, and (v) whether the approach is programming language dependent.

- The first column represents SPaCiTE that uses Semantic Mutation Operators. This approach generates test cases based on abstract behavioral descriptions written in ASLan++. Reported Abstract Attack Traces (AATs) contain the special keywords *inject* and *verify* (see Section 2.7). They indicate which message parameters have to be used for which kind of malicious injection. A pre-defined library contains concrete malicious payloads, from which SPaCiTE selects elements. SPaCiTE uses the applied Semantic Mutation Operators as a filter criterion to only get those elements from the library, that match with the injected vulnerability. Finally, SPaCiTE does not depend on the programming language the web application is implemented with.
- The second column represents tools that are based on Syntactic Mutation Operators. We consider the SPaCiTE approach again, but apply Syntactic Mutation Operators

	SPaCiTE (Semantic M.O.)	**Syntactic Mutation Operators**	**Black-box Scanner + Injection library**	**Black-box Fuzzer + Input Grammar for Injection**	**White-box Fuzzer**	**Guided White-box Approach**
Model	abstract behavioral description	abstract behavioral description	no model	multiple input grammars	source code	source code + line(s) you want to reach
Parameters used for malicious injection	input parameters affected by injected vulnerability	input parameters affected by mutation; all parameters if affected input parameter cannot be determined	every input parameter	only those input parameters are tested that follow the grammars.	input parameters those values reach a specific location in the source code	selected according to the line(s) to be reached
Source of malicious inputs	pre-defined library	pre-defined library	pre-defined library	dynamically generated	dynamically generated path constraint filters pre-defined library	dynamically generated path constraint filters pre-defined library
Generation / Selection of malicious inputs	selected according to the injected vulnerability	all elements in the library	all elements in the library	generated based on the grammar (type) , but not implementation dependent	implementation dependent malicious injections	need to be combined with additional techniques.
Programming language dependent	no	no	no	no	yes	yes

Table 4.1.: Tool Categories for Effectiveness Comparison

instead of semantic ones. Thus, this 'configuration' differs in two aspects to the first column. When Syntactic Mutation Operators are used, not every AAT contains the special keywords *inject* and *verify*. For traces that miss them, the Test Execution Engine (TEE) does not know when to inject which type of malicious payloads. Therefore, every parameter is used for the injection. The second difference is, that the type of underlying vulnerability is not known and therefore, every element in the malicious payload library is used.

- The third column represents black-box scanners that rely on a pre-defined library containing malicious payloads. Black-box scanners for web applications do not operate on a model, but typically use crawlers to learn the different requests and parameters the web application accepts. Since they operate on the API level, they have very limited knowledge about the internal behavior beyond the API. In particular, they do not know with which technology a message parameter value is processed. As a consequence, such tools do not know possible vulnerabilities that a message parameter value can exploit. Therefore, such tools iterate over a pre-defined library with malicious payloads and inject them using every discovered message parameter. Like SPaCiTE, such tools work independent of the programming language the web application is implemented with.

- Pre-defined malicious payloads have the disadvantage of being a random collection of payloads. To consider more structured malicious payloads, black-box scanners can be extended with input grammars for the malicious payload injection (fourth column). The purpose of such input grammars is to increase the chance of ending up with malicious payloads that are syntactically correct and therefore executable by the web application. Instead of taking malicious payloads from a pre-defined library, they fuzz the parameter values according to grammars (SQL, HTML, etc.). Like for black-box scanners, they do not depend on the programming language the web application is implemented with.

- White-box approaches generate test cases considering the source code of a web application. They can only be applied if the source code is available. While the Security Analyst needs to understand and select the vulnerability related source code lines when a guided white-box approach is considered (sixth column), this is not necessary for non-guided white-box fuzzers (fifth column). White-box testing tools exploit source code characteristics like path constraints or specific method invocations to construct implementation dependent test cases. To use such approaches in the context of security testing they need to be combined with pre-defined malicious payload libraries, or any other malicious payload generation approaches.

In general, SPaCiTE can be compared to tools of any of the above described categories to highlight different aspects. For our evaluation, we make the decision to compare SPaCiTE with black-box scanners and compare SPaCiTE with Syntactic and Semantic Mutation Operators because of the following reasons:

- We exclude white-box approaches because such approaches can only be applied if the source code is available. Such access might not always be given. Whether source code is available is often an external decision made by the product owner, and not by the Security Analyst. Both black-box approaches and SPaCiTE do not depend on the availability of source code and therefore, they can test web application even if source code is not available.

- Both black-box approaches and SPaCiTE can be applied to web applications independent of the programming languages used to implement the applications. Therefore, for web applications implemented with different programming languages, SPaCiTE can be compared to the same black-box tools.
- Both approaches use pre-defined libraries for malicious payloads and do not dynamically generate them either from the source code or input grammars. Both approaches more focus on finding an injection and verification point using pre-defined and general malicious payloads than constructing a specific, application dependent malicious payload for a given input parameter.
- Black-box scanners and SPaCiTE differ in terms of required models. While black-box scanners are point-and-click tools that learn the behavior of the web application, SPaCiTE requires a formal behavioral specification. Such a comparison elaborates whether formal specifications are suitable for being more effective in finding non-trivial vulnerabilities.
- SPaCiTE with Syntactic and Semantic Mutation Operators differ in terms of manually annotating the formal specification. Besides that, both approaches operate on the same formal specification, both use pre-defined libraries for malicious payloads and are not source-code dependent. Therefore, comparing Syntactic vs. Semantic Mutation Operators elaborates on the consequences of manual annotations.

In sum, we consider black-box scanners for a comparison with SPaCiTE, since a comparison with white-box approaches is more costly. Source code needs to be available, and tools that handle the corresponding programming languages need to be found. In this respect, black-box scanners and SPaCiTE are more convenient, since either no model is required at all, or an abstraction of the System Under Validation (SUV) is sufficient.

To evaluate the hypothesis of this Ph.D thesis, we first compare our approach, that uses vulnerability-based fault models at ASLan++ model level, with black-box vulnerability scanners. Since vulnerability scanners might not find vulnerabilities due to bugs in the scanners themselves, we also compare the test generation with Syntactic and Semantic Mutation Operators at the ASLan++ model-level.

The evaluation answers the following research questions:

- *If non-trivial security vulnerabilities are present, does SPaCiTE, that uses vulnerability-based fault models, find in practice a higher number of these vulnerabilities than black-box vulnerability scanners?*
- *Are vulnerability-based fault models at ASLan++ model level more efficient in generating 'security-interesting' test cases than syntax-based fault models at the same model level?*

As part of the evaluation, we consider a third research question:

- *How much does a Domain Specific Language (DSL) contribute to the efficiency of mapping abstract test cases to executable test cases?*

To answer the above questions we consider the following metrics.

- To answer the first research question, we consider Cross-site Scripting (XSS) and Structured Query Language (SQL) related non-trivial vulnerabilities and count the number of found vulnerabilities in case studies.
- To answer the second research question, the metrics are: (1) the number of mutated models, since model checking is very resource-intensive (2) the fraction of mutated models that generate an AAT, since only AATs are useful for test case generation, and (3) the number of executable test cases. Efficiency is better: (1) the smaller the set size of mutated models, (2) the higher the AAT generating ratio (mutated models that violate a security property), and (3) the smaller the number of test cases while still finding the potential vulnerability.
- Finally, to answer the third question, we consider the number of code lines the Security Analyst has to provide with our DSL compared to using a general purpose language. When we discuss this metrics later, we will see that these numbers have to be taken with care.

4.2. Vulnerable Web Applications

In this section, we select and describe vulnerable web applications that we consider for the evaluation. We use the OWASP Broken Web Applications Project version 1.1.1 virtual machine [20] (WebGoat and Wackopicko) and a bank application developed by students. The set of applications is split into two categories — training and realistic applications, intentionally and non-intentionally vulnerable applications. WebGoat was known and used already from the beginning of the development of SPaCiTE and initially inspired our work. Therefore, it is considered as a training and intentionally vulnerable application. Wackopicko and the bank application at the other side, were not considered during the development but used during the evaluation only. Furthermore, Wackopicko is intensionally vulnerable while the bank application is not. In the following, we give a short summary of the applications.

WebGoat. This web application [39] is an J2EE application developed for teaching purposes and is therefore made insecure on purpose. It consists of many different lessons, each dedicated to a specific vulnerability. For this effectiveness evaluation, we consider the application 'Goat Hills Financial - Human resources' inside WebGoat. The purpose of this web application is to manage user profiles. Different users can login to the system, view user profiles and edit them. According to security aspects, this application is in particular vulnerable against a stored XSS attack.

WackoPicko. This application is written in PHP. This application was developed as part of the paper published by Doupé et al. [92]. It represents a realistic application where new users can register with an account, upload and comment pictures. In addition, it provides a functionality to find other users with a similar name. The implementation contains many well-known vulnerabilities. Among other vulnerabilities, it suffer from stored XSS, multi-step stored XSS, and stored SQL injection vulnerabilities — vulnerabilities, that are hard to find by automatic tools.

Bank Application. The bank application was developed as part of the class 'Secure Coding' offered by the chair of Prof. Dr. A. Pretschner, taught at Technische Universität München during the winter semester 2014/2015. Master students had to implement a banking web application according to the following functional and non-functional requirements:[1]. The functionality is quite big and we only consider a subpart of the whole application for applying our approach. The covered functionality is:

- Register a new customer with a registration form.
- Log in and logout as client.
- Log in and logout as admin.
- Approval of client registration requests.

To test these applications, SPaCiTE requires a formal specification as input. In the following sections, we briefly describe them. When we model check these formal specifications, we use a virtual machine with 4 vCPUs @ 2.6 GHz, 8GB of RAM, and Ubuntu 14.04.1 as Operating System.

4.2.1. WebGoat

A formal specification of the WebGoat application is given online at [38]. It describes a session between a user (tom or jerry) and the WebGoat application. The application allows a user to login, view and edit profiles. The **login** request requires two parameters, a username and a password. The credentials are checked if they match with an entry in the database, represented by the fact `db`. After a successful login, WebGoat replies with an ID of a user profile, that the currently logged in user is authorized to view.[2] The **viewProfile** message accepts one parameter — the username of the agent, whose profile is requested. The matching profile is then retrieved from the database and the result is sent back to the client. Please note that the initial request parameter is contained in the response, as well as data from the database. Finally WebGoat allows to modify a profile by sending an **editProfile** message. This request requires two different parameters — the username of the agent whose profile should be edited, and the data that should be stored as a profile. The model describes that the user-provided input is first sanitized against HTML-based XSS attacks and SQL injections. The sanitized data is then stored in the database. Since we assume that the used sanitization method has completely removed dangerous characters from the user-provided input, also the corresponding values stored in the database are still sanitized against XSS. Finally the user-provided values are sent back as a response to the client. Important to note here is the fact, that for the **editProfile** request, the response contains the user-provided values, and not the values from the database. After editing a

[1] A full description can be found at `https://www22.in.tum.de/fileadmin/teaching/ws2014/seccoding/Phase1Desc-merged.pdf`

[2] Please note that neither the username nor the password is contained in that reply.

profile, the message retE2L allows to return to the welcome webpage that shows the list of profiles that the logged in user is authorized to view.

In terms of security goals, the WebGoat models specifies, that every value involved in an SQL query needs to be sanitized against SQL (see goal1 and goal2). Furthermore, every data element sent back to an honest user needs to be sanitized against XSS.

SPaCiTE model-checks the specification described in this section using Cl-Atse Version 2.5-21 with the parameters --nb 2 --free --not_hc --lvl 2 --short. After 37 hours, SPaCiTE terminated the model checker. Therefore, no verdict is present for this model and it is considered as a 'long-running' specification according to Section 2.1. In particular, there is no trace found by the model checker during the model-checking phase that violates the specified security properties. Therefore, also no AATs are generated.

4.2.2. Wackopicko

Wackopicko is an application known from the paper: *Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners* [92] and represents a real-world web application. In particular, it implements sophisticated features that are often found in modern web application but are difficult to test. Doupé et al. [92] show that many automatic web crawlers fail to find well-known vulnerabilities. Therefore, we apply SPaCiTE to that application and show that it finds vulnerabilities that are missed by other approaches.

A summary of the functionality that we consider for this case study is given in Figure 4.1. We consider the following four functionalities:

Register. A new user has the possibility to open an account with Wackopicko following the Register link. He is asked to provide the following information: username, first name, last name, and a password.

Welcome page. After successfully registering with the web application, the user is forwarded to the welcome page that shows his username as part of the page.

Similar users. The web application provides the functionality to find users with similar first names as the logged-in user's own first name. Implementation-wise, the user's first name is used to construct a corresponding SQL query.

Uploading pictures. To upload a picture to the Wackopicko platform, the user is asked to provide a tag, a filename, a title and a price. Since a picture can be augmented by a comment, the user is forwarded to a page that reflects the title, his username, and a form to enter the desired comment. As a special functionality, the provided comment is not directly published but the user is again forwarded to a preview page that reflects the provided comment together with his username and the title of the picture. Only after confirming the preview, the entered information is displayed and stored.

Recent. Finally to display the recently uploaded pictures, Wackopicko provides a link to display the title, filename, comment, and username of the commenter.

A possible formal specification of the above described Wackopicko functionality is given online at [35]. Since Wackopicko represents a real-world application in terms of functionality, the model is rather large. We model checking the specification with SPaCiTE (Cl-Atse

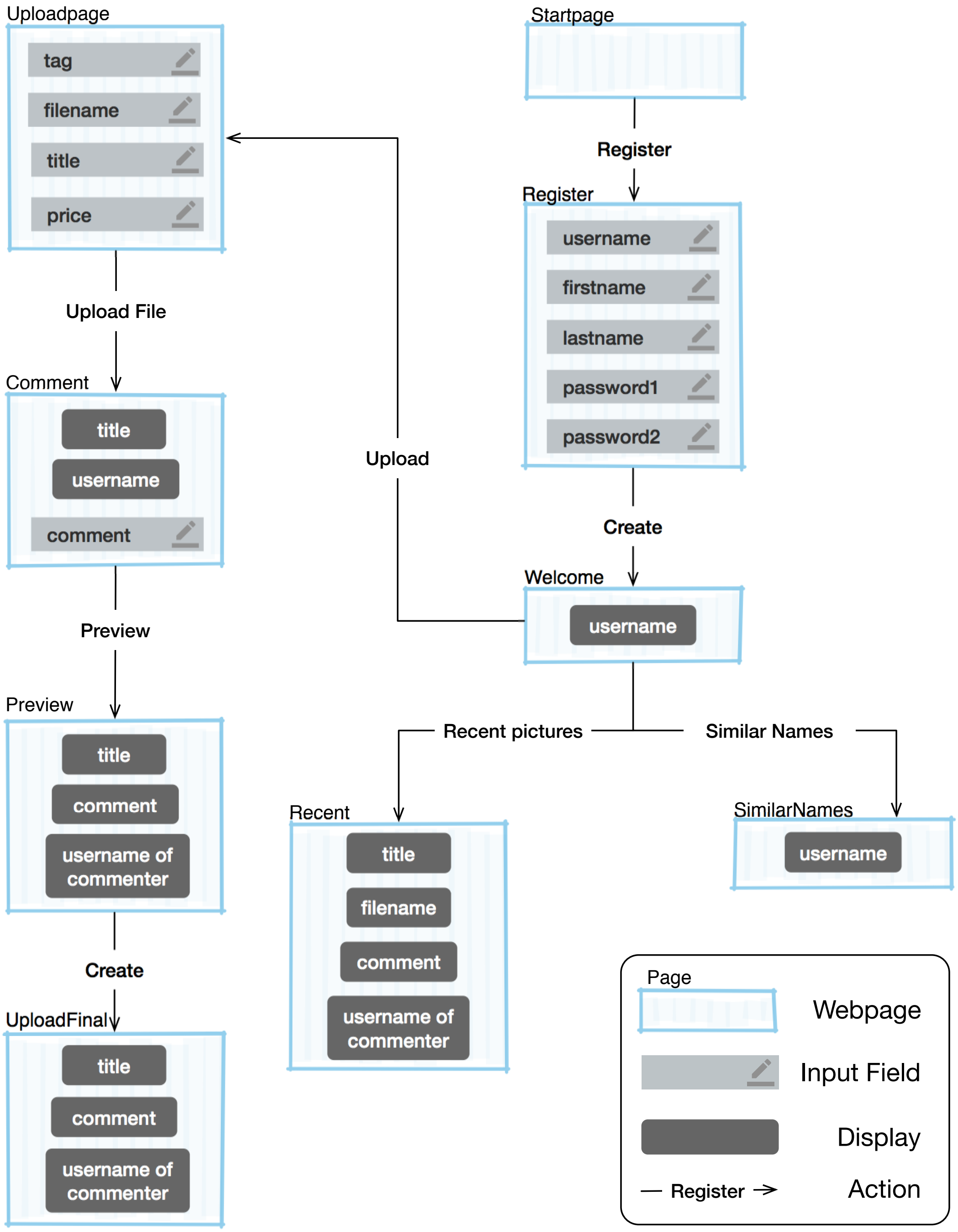


Figure 4.1.: Functionality of Wackopicko

Version 2.5-21 with the parameters `--nb 2 --free --not_hc --lvl 2 --short`). Since the model checker did not find any issues withing 66 hours, the Wackopicko specification is a long-running specification (see Section 2.1).

4.2.3. Bank Application

A formal specification of the Bank application is given online [7]. The model describes five different functionalities of the banking application — *registering a new client*, *login of the client*, *login of an admin*, *activating a client*, and *logout*.

LoginAdmin. The bank application allows an administrator to login by sending the abstract message `loginAdmin`. This message requires two arguments — the username and the password. Upon successful login, the web application selects a newly created user account that is not activated yet, and sends back a webpage to the administrator which contains the `username`, `first name`, `last name`, and `email address`.

Logout. The purpose of the `logout` message is to de-authenticate a user. The message does not require any arguments.

ActivateClient. After the administrator has received a non-activated user account, he activates it by sending the `activateClient` message. This message requires one argument only, namely the username of the account to be activated. Using this argument, the corresponding record from the database is retrieved. To activate the user account, the retrieved username from the database is used to update the database record. Finally the action is acknowledged by sending a static value (`ack_activateClient`) back to the administrator.

LoginClient. Similar to an administrator, a client can login to the bank application by providing his `username` and his `password`. A successful login by a client requires that his account is activated. The bank application acknowledges a successful login with the message `ack_login` which contains the client's `first name`, `last name`, and his `email address`.

Register. Before a client can login to the bank application, he has to register an account using the `register` message. This message requires five arguments — the `username`, `first name`, `last name`, the `email address`, and the `password`. After the bank application has stored this information in a database, it acknowledges the registration with the message `ack_register`.

The model checker we use in SPaCiTE (Cl-Atse Version 2.5-21 with the parameters `--nb 2 --free --not_hc --lvl 2 --short`) needs 2800 seconds to check 228 states and 3096 transitions. The model checker terminates with the verdict `NO ATTACK FOUND`.

```
SUMMARY:
  NO_ATTACK_FOUND

DETAILS:
  TYPED_MODEL
```

```
BACKEND:
  CL-ATSE 2.5-21_(2012-dÃ©cembre-13)

STATISTICS:
  TIME 2801989 ms
  TESTED 3096 transitions
  REACHED 228 states
  READING 12.96 seconds
  ANALYSE 2789.03 seconds
```

4.3. Black-Box Vulnerability Scanners

WebGoat

[The following section was partially published in the SPaCIoS Deliverable 5.5 [200]]

In order to security test WebGoat and to compare it with SPaCiTE, we use the following scanner.

OWASP Zed Attack Proxy (ZAP) [49]. This vulnerability scanner is a penetration testing tool to find vulnerabilities in web applications. At the time of writing, version 2.3.1 was the most recent version. Among other vulnerabilities, it checks for reflected and stored XSS, as well as SQL injections.

To demonstrate the effectiveness of the scanner, we applied ZAP in two different ways. First of all, we manually explored the complete WebGoat application (including the lessons with the Goat Hills Financial app but also other lessons) and run ZAP afterward. In this setting (i.e., manual+ZAP), ZAP finds simple vulnerabilities: 20 reflexive XSSs (CWE-79), 1 path traversal (CWE-22), and 3 SQL injections (CWE-89). Several reported XSS by ZAP are not real. Indeed, ZAP tests for XSS vulnerabilities by entering HTML code into potential vulnerable spots and check whether or not the HTML that was injected is present on the page. However, if the injected HTML is present in an input box, the tool will count it as an XSS vulnerability even though code inside such a box will never be executed. So, even though the injected HTML is present in the page, it can not be exploited. Note that none of the reported vulnerabilities are related to the Goal Hills Financial app. The reason is that this lesson requires authentication. Even though we authenticate ourselves during the manual exploration, ZAP does not handle the WebGoat session properly and therefore, it is unable to find the XSS vulnerabilities in the Goat Hill Financial lesson. Exploring the application thanks to the SPaCiTE-conform model, and therefore guiding ZAP, and running ZAP after each test case execution (i.e., BFS+ZAP), ZAP is able to find 8 reflexive XSSs in the Goat Hill Financial. However, it does not find any of the authentication flaws nor the stored XSS. The results are shown in Table 4.2.

			Vulnerabilities (TP/FP)				
Method	# tests	Overall	rXSS	sXSS	PathT	SQLi	Logical
manual + ZAP	16886	24 / 9	20 / 9	0 / 0	1 / 0	3 / 0	0 / 0
BFS +ZAP	1113	8 / 0	8 / 0	0 / 0	0 / 0	0 / 0	0 / 0

Table 4.2.: Results For ZAP Applied to WebGoat (BFS=Breadth First Search)

Wackopicko

Wackopicko was developed to show that vulnerability scanners have great difficulties to find interesting and important vulnerabilities in this web application. The web application suffers from the following types of vulnerabilities. We group the vulnerabilities according to their detectability by the vulnerability scanners: Vulnerabilities 1. - 8. are not discovered by any scanner. The remaining vulnerabilities were discovered by at least one scanner.

(1.) ***Stored SQL Injection.***, (2.) ***Multi-Step Stored XSS.***, (3.) *SessionID vulnerability.*, (4.) *Weak username/password.*, (5.) *Parameter manipulation.*, (6.) *Directory Traversal.*, (7.) *Forceful Browsing.*, (8.) *Logic Flaw.*

(9.) *Reflected XSS Behind a Flash Form.* After a failed login, Wackopicko shows the error page using flash. That means that the TEE needs to handle Flash correctly to build the verdict when the login page is involved.

(10.) ***Stored XSS.*** Usually the stored XSS vulnerability is challenging for automatic vulnerability scanners. For the specific vulnerability in the guestbook of Wackopicko, 10 out of 11 scanners found the vulnerability, although 4 scanners needed manual configurations. We believe that the reason that 10 scanners found the vulnerability is two fold — (1.) The vulnerability is not multi-step, that means the guestbook is accessible following one link only, and (2.) no authentication/login step is required both for the exploitation of the vulnerability and the suffering from the attack by the victim.

(11.) *Reflected XSS Behind JavaScript.* On the index page of the Wackopicko application, a user enters the name of a file into the textfield with the name 'name'. The web application adds this textfield by Javascript code. That means that the TEE needs to interpret Javascript code to correctly use the web application.

(12.) Unauthorized File Exposure, (13.) *Reflected XSS*, (14.) *Reflected SQL Injection*, (15.) *Command-line injection*, (16.) *File Inclusion*.

Doupé et al. [92] tested the web application using eleven different vulnerability scanners (*Acunetix, AppScan, Burp, Grendel-Scan, Hailstorm, Milescan, N-Stalker, NTOSpider, Paros, W3af, and Webinspect*). The authors report that 8 out of 16 vulnerabilities (1. - 8. of the above enumeration) were not detected by any of the used scanners. The rest of the vulnerabilities were discovered by at least one scanner. Bold vulnerabilities are vulnerabilities that SPaCiTE focuses on in the current version. In particular, we will see later that SPaCiTE's strength are stored SQL injection and the multi-step stored XSS vulnerability, both vulnerabilities, that were not discovered by any scanners. SPaCiTE will generate AATs for these kind of vulnerabilities and find them by executing the corresponding test cases. Furthermore the vulnerability '*Reflected XSS Behind a Flash Form.*' was not detected by 5 out of 11 scanners, and the remaining 6 scanners require manual configuration to discover the vulnerability. Even more complicated seems the *Reflected XSS Behind JavaScript* vulnerability. Only 3 out of 11 scanners discover this vulnerability. These kind of vulnerabilities do not offer an additional burden for SPaCiTE since the TEE executes attack cases with the help of the browser and therefore, Flash and Javascript are automatically handled. The 'reflected' vulnerabilities (i.e., reflected XSS, reflected SQL) are found by SPaCiTE as well due to the systematic generation of attack traces but since they are also discovered by automatic vulnerability scanners, they are not in the focus of SPaCiTE.

Bank Application

Students were asked to implement the application and the application used for this evaluation was manually and automatically tested. The students used ZAP, w3af, Nikto, Burp, Grendel, and InjectMe to test the application for known vulnerabilities. For the manual task, they followed the OWASP Guide. All students that assessed this application reported two vulnerabilities, — a transaction data validation flaw and a click jacking vulnerability. Both of these vulnerabilities are not covered by our approach, since no mutation operators for these kind of vulnerabilities are developed so far. According to XSS and SQL vulnerabilities, neither the manual nor the automatic approach applied by the students disclosed any vulnerability.

4.4. Effectiveness Evaluation

To show that the proposed Semantic Mutation Operators are powerful enough to generate test cases for non-trivial vulnerabilities, we apply Semantic Mutation Operators to the three discussed formal specifications of WebGoat, Wackopicko, and the bank application. For this evaluation, SPaCiTE was configured to use the Cl-Atse parameters `--nb 2 --not_hc --free --lvl 2 --short`, four concurrent threads, and a model checking timeout of 2400 seconds. This configuration stays the same for all three models.

In the first part, we will discuss the kind of AATs that are generated with different Semantic Mutation Operators. In particular we are interested whether Semantic Mutation Operators lead to test cases that discover non-trivial vulnerabilities. Then, we will show that selected AATs can be instantiated to successfully confirm a potential non-trivial vulnerability. We are aware that no comprehensive and excessive test case execution is possible due to several reasons. It turns out that automation starting from abstract specifications and ending at executable test cases is complex and faces many technical challenges. Nevertheless executing those AATs that are dedicated to non-trivial vulnerabilities is sufficient to show the effectiveness.

To not be repetitive, we will focus on different aspects for each available formal specification. We use the WebGoat model to discuss in detail the application of different mutation operators and their corresponding generated attacks. Using the Wackopicko model, we focus more on the overall set of generated mutation operators and compare the generated AATs in terms of first-order vs. higher-order, and Syntactic vs. Semantic Mutation Operators. Finally, we use the bank application formal specification to focus on the false positive evaluation.

For a compact representation of the generated AATs, we use classification tables like Table 4.3. They have to be read in the following way:

- On the x-axis we show different messages that are exchanged between the browser and the web application. If such a message consists of several parameters, we list them below the message name, separated with a short horizontal line. E.g., in Table 4.3, the `Edit` message consists of the parameters `ID` and `data`. Furthermore, we use the label *check after X* to specify, after which message the victim gets attacked. E.g., in row 1 of Table 4.3, the victim gets attacked after sending the `view` message.

- On the y-axis we list different AATs. We use labels like XSS and SQL to indicate what kind of malicious code is injected. A check mark (✓) is used to indicate after which message the victim is attacked. If multiple agents are involved in the test case, they are added in parenthesis. E.g., AAT 7 in Table 4.3 expresses that the agent `tom` injects XSS malicious code using the `data` parameter of the `edit` message and that the agent `jerry` gets attacked after sending the `view` message.

4.4.1. Semantic Mutation Operators on WebGoat

4.4.1.1. XSSRemoveSanitization

XSSRemoveSanitizationUpToLimit[1][1]. We first apply the `XSSRemoveSanitizationUpToLimit[1][1]` mutation operator that represents a first-order mutation operator. It only invalidates one single sanitization block per mutated model. In this configuration, a total of eight mutated models are generated. Three mutated models generate an AAT, all representing reflected XSS attacks (AATs 1 to 3 in Table 4.3). They exploit the single parameter of the `view` message and both parameters of the `edit` message. Intuitively, the fact that only reflected XSS attacks are reported makes sense since for a reflected attack, the input is received by the server, must not be sanitized, and is sent back to the client all happening in the same message handler.

XSSRemoveSanitizationUpToLimit[1][2]. To detect more difficult vulnerabilities like stored XSS, higher-order Semantic Mutation Operators are required since XSS sanitization must not happen when the data is stored and also when the same data is read again and sent back to the client. Since in this situation multiple sanitization blocks are involved, we apply the `XSSRemoveSanitizationUpToLimit[1][2]` higher-order Semantic Mutation Operator to the model. It mutates up to two mutation candidates to generate a mutated model. We end up with a total of 36 mutated models. Model checking all of them, 24 mutated models generate AATs. These attacks include all AATs reported by `XSSRemoveSanitizationUpToLimit[1][1]`, and include in addition split reflected XSS attacks and multi-step stored XSS attacks using one parameter. For (multi-step) stored XSS attacks performed with one parameter, two mutated sanitization blocks are required, namely a sanitization block when the data is stored, and a sanitization block when the data is retrieved later.

AAT	**edit**		**login**		**view**	**check after**			
	ID	data	uname	pwd		login	edit	view	std.page
1					XSS			✓	
2		XSS					✓		
3	XSS						✓		
4	XSS	XSS					✓		
5	XSS(tom)					✓(jerry)			
6	XSS								✓
7		XSS(tom)						✓(jerry)	
8(*)	XSS(tom)	XSS(tom)				✓(jerry)			
9	XSS(tom)	XSS(tom)						✓(jerry)	
10(*)	XSS	XSS							✓

Table 4.3.: XSSRemoveSanitization Application on WebGoat (✓=Agent Gets Attacked)

AAT 4 represents a split reflected XSS attack where the profile ID and the profile data parameter of the edit requests are not sanitized against XSS. **AAT 5** represents a stored XSS attack where a dishonest user injects a malicious XSS payload using the profile ID of the `edit` message. To be attacked, a victim logs in and is forwarded to the welcome/standard page. Since that page shows available profiles, the malicious payload might be executed. **AAT 6** represents a multi-step stored XSS attack where a user injects a malicious XSS payload using the profile ID of the `edit` message. In contrast to AAT 5, the malicious input gets executed upon navigating to the standard page. Finally, **AAT 7** represents a multi-step stored XSS attack where a dishonest user injects a malicious XSS payload using the data parameter of the `edit` message. To be attacked, a victim logs in and requests the profile with the malicious content. This AAT exploits the vulnerability that leads to the stored XSS attack. Therefore, the proposed Semantic Mutation Operators are powerful enough to find the vulnerability at the abstract model level. In Section 4.4.1.6 we will further see that SPaCiTE is also powerful enough to find the vulnerability in the implementation of the SUV.

XSSRemoveSanitization. Finally, we apply the `XSSRemoveSanitization` Semantic Mutation Operator which considers every possible combination of mutation candidates. Applying it generates a total of 255 mutated models. Model checking all of them results in 241 models that generate an AAT. This set contains three new AATs (AAT 8 to 10 in

Table 4.3) that are not generated so far with the above mutation operators. All of them represent split multi-step XSS attacks. The injected vulnerability can be exploited by a user who edits his own profile. The `edit` request contains malicious XSS code in the profile ID and profile data field. To get attacked, a victim user logs into the web application and receives a malicious profile ID for **AAT 8**, views the malicious profile for **AAT 9**, and requests the standard page for **AAT 10**.

In Table 4.3 we marked AAT 8 and 10 with a star because we will discuss the rationales why they are generated in Section 5.7. Carefully analyzing them, it sounds counter-intuitive why using the data parameter of the `edit` message should be used for an injection in the situation where the victim gets attacked after the `login` message.

4.4.1.2. XSSRemoveSanitization on Adapted WebGoat

In Section 4.2 we described the WebGoat application. In particular, we discussed that the WebGoat application accepts the messages *login*, *view*, *edit*, and *retE2L*. Analyzing the reported AATs in Table 4.3, one observes that not all possible combinations on injection and verification location for XSS are reported so far. For instance, test cases for the following situations are missing:

X1. No AAT exploits the parameters of the `login` request for an XSS attack.

X2. No AAT exploits the retE2L request for an XSS attack.

X3. No AAT represents a multi-step stored XSS attack using the ID parameter of the `edit` message and being attacked while viewing the profile.

In this paragraph we will argue about the reasons for the missing AATs. The above three mentioned AATs are missing due to modeling characteristics. Neither of the two `login` request parameters are contained in any response the web application sends to the client. Therefore, the first attack behavior cannot be induced at the model level. Test cases for X2 are not generated because the request for the standard page does not contain any parameters at all and therefore, there is also no possibility to inject XSS malicious code using that request at the model level. Finally, the `view` message expects a profile ID as one of the parameters that is used to select the profile to be shown. In the response, the user-provided ID is reflected instead of the profile ID stored in the database. Therefore, no test case for X3 is generated.

For illustration purposes, we modify the formal specification to demonstrate the power of the different `XSSRemoveSanitization` Semantic Mutation Operators. We want to show that X1 and X3 are missed not because of weaknesses of the mutation operators, but because of the formal specification. Listing 4.1 shows the crucial server-side changes in the formal specification in diff format. Client-side changes and declarations are ignored. Listing 4.1 shows the following changes: For the `login` request, we will add a parameter to reflect the username. For the `view` message, we return the stored profile ID rather than the user-provided value part of the view message. Finally, please note that the `standard webpage` was not adapted and still does not require any parameters.

Applying the mutation operator to the mutated formal specification shows that two missing AATs for X1 and X3 are generated, that exploit the username parameter of the `login` message and the ID parameter of the `edit` message (see Table 4.4).

AAT	edit		login		check after	
	ID	data	uname	pwd	view	login
X1			XSS			✓
X3	XSS(tom)				✓(jerry)	

Table 4.4.: XSSRemoveSanitization Application on Adapted WebGoat (Syntax: XSS(y) = Agent y Performs an XSS Attack); ✓[y] = Agent y Is Attacked

Listing 4.1: Adaptation to the `login` Message of the WebGoat Model

```
Symbols Declarations:
-  listStaff( username_t ) : message ;
+  listStaff( username_t, username_t ) : message ;

Login Request Handler
+  LoginUsername_Received:=instance_username_t(LoginUsername,RandomValue1Login);
+  %@Semantics[ HTML, Sanitize, Input ]
+  xss_sanitize_username_t( LoginUsername_Received ) ;

-  Actor *->* Us : listStaff( ReceivedID_login ) ;
+  Actor *->* Us : listStaff( LoginUsername_Received, ReceivedID_login ) ;


View Request Handler: sending back database value
-  Actor *->* Us : profile( ReceivedIDParameter_view, ReadData_view ) ;
+  Actor *->* Us : profile( DBIDParameter_view, ReadData_view ) ;
```

4.4.1.3. SQLRemoveSanitization

So far we were focused on XSS vulnerabilities and their corresponding Semantic Mutation Operators. In this section we consider vulnerabilities related to SQL injections.

SQLRemoveSanitizationUpToLimit[1][1]. We first apply the `SQLRemoveSanitizationUpToLimit[1][1]` mutation operator that represents a first-order mutation operator. In this configuration, a total of three mutated models are generated where all of them generate an AAT (AAT 1 to 3 in Table 4.5). All three AATs generate reflected attacks. While an attacker uses the profile ID and the data parameter respectively of the `edit` message to inject malicious SQL code in **AAT 1** and **AAT 2** respectively, he makes use of the single view parameter to inject malicious SQL code in **AAT 3**.

SQLRemoveSanitizationUpToLimit[1][2]. The results of applying the previous SQL related Semantic Mutation Operators show that so far, no test cases for split or stored SQL attacks were generated. For a split or a stored SQL injection we need higher-order mutation operators. Therefore, we apply `SQLRemoveSanitizationUpToLimit[1][2]`

AAT	edit		login		view	check after		
	ID	data	uname	pwd		view	edit	login
1	SQL						✓	
2		SQL					✓	
3					SQL	✓		
4	SQL	SQL					✓	
5			SQL					✓
6				SQL				✓
7			SQL	SQL				✓

Table 4.5.: SQLRemoveSanitization and InvalidateSQLConditionCheck* on WebGoat

which generates four mutated models in total. All of them generate AATs. Compared to the application of `SQLRemoveSanitizationUpToLimit[1][1]`, only one new AAT is generated (**AAT 4** in Table 4.5). It describes a split reflected SQL injection using the profile ID and the data parameter of the `edit` message.

SQLRemoveSanitization. Finally, we apply the `SQLRemoveSanitization` Semantic Mutation Operators to the formal specification. It turns out that no new AATs are generated. Intuitively this makes sense since there is no message that consumes more than two parameters that could be exploited for a split SQL injection. Considering the formal specification [38], there is also no stored SQL attack possible, since no SQL value is read from the database that is later used as part of another SQL query. Therefore, AATs related to (multi-step) stored SQL injections are not generated.

4.4.1.4. InvalidateSQLConditions

InvalidateSQLConditionCheckBy{ DetachingVariable, SettingRandomValue }[1][1]. The modeling of the `login` message handler in Listing 4.2 shows that the receiving of the message and the SQL operation are combined in one single condition. Therefore, the mutation operators `SQLRemoveSanitization` and `SQLRemoveSanitizationUpToLimit` do not generate mutated models, since they do not operate on ASLan++ conditions. To inject SQL vulnerabilities in this situation, we apply the mutation operators `InvalidateSQLConditionCheckByDetachingVariable` and `InvalidateSQLConditionCheckBySettingRandomValue`. Applying `InvalidateSQLConditionCheckByDetachingVariable[1][1]` to the WebGoat formal specification generates in total four mutated specifications. Upon model checking, all of them generate AATs. The AATs correspond to AATs 1, 3, 5, and 6 in Table 4.5. The first two AATs are also reported by the

SQLRemoveSanitization mutation operator. The reason is that the ID parameter of the view message handler is involved in two different SQL queries. The first query is used to check if the user is authorized to view the profile (Line 22 in Listing 4.2). The InvalidateSQLConditionCheckByDetachingVariable mutation operator is applied to this condition check and generates AAT 3. Since the same ID parameter is used for updating the profile (Lines 31 to 38) as well, and therefore sanitized in Line 29, the same AAT is also generated by the SQLRemoveSanitization mutation operator. Exactly the same argumentation is applicable to the ID parameter of the view message handler and is therefore not discussed separately. The latter two traces (AAT 5 and 6) have not been generated before. They are reflected SQL injections using the username and the password parameter of the login message.

Although the InvalidateSQLConditionCheckBySettingRandomValue [1][1] mutation operator generates four mutated models as well, it does not generate any AATs. The reason for this is the annotated condition that involves an SQL query. It occurs in positive form[3] and has to be evaluated to true to violate the corresponding security property.

Listing 4.2: Extract of WebGoat Model (inst_u_t=instance_username_t)

```
1  on( ?Us *->* Actor : login(
2             inst_u_t(?LoginUsername,?RandomValue1Login),
3             instance_password_t(?LoginPassword,?RandomValue2Login))
4           & inSUT( ?Us ) & !authenticatedAsClient( ?Us )
5           &
6           %@Semantics[ SQL, PermissionCheck ] {
7             db( instance_username_t(?LoginUsername,?),
8                 instance_password_t(?LoginPassword,?),
9                 ?
10            )
11          %@}
12      ): { ... }
13
14 on( Us *->* Actor : editProf(
15            inst_u_t(?ReceivedID_edit,?RandomValue1_edit_Server),
16            instance_text(?ReceivedData_edit,?RandomValue2_edit_Server)
17   ) &
18   inSUT(Us) &
19   inSUT_text(?ReceivedData_edit) &
20   authenticate( Us ) &
21   %@Semantics[ SQL, PermissionCheck ] {
22     Us->canEdit( ?ReceivedID_edit )
23   %@}
24   & db(inst_u_t(?ReceivedID_edit,?),?,?)
25 ):{
26   ReceivedID_edit2:=inst_u_t(ReceivedID_edit,RandomValue1_edit_Server);
27
28   %@Semantics[SQL, Sanitize, Input]
29   sql_sanitize_username_t( ReceivedID_edit2 );
30
31   %@Semantics[ SQL ] {
32    select{ on(
```

[3] In particular, it does not have the form !condition.

```
    db(inst_u_t(ReceivedID_edit,?RandomValue4_edit_Server),?Password_edit,?)):{}
   }
   retract db(...) ;
   db(StoreID,Password_edit,StoreData) ;
   writtenToDB( ReceivedID_edit2, StoreID,ReceivedData_edit2,StoreData);
  %@}
...
```

InvalidateSQLConditionCheckBy{ DetachingVariable, SettingRandomValue } [1][2]. Applying these two mutation operators, there is one additional AAT that is generated (AAT 7 in Table 4.5). It represents a split reflected SQL injection using both the username and the password parameter of the `login` message. Since this kind of attack requires at least two injections — invalidating the sanitization of both `login` message parameters — it is only generated with a higher-order mutation operator. In addition, the `InvalidateSQLConditionCheckBySettingRandomValue[1][2]` generates no AAT with the same rationales as mentioned for `InvalidateSQLConditionCheckBySettingRandomValue[1][1]`.

InvalidateSQLConditionCheckBy{ DetachingVariable, SettingRandomValue }. We finally check whether we miss AATs that have not been generated so far. We apply both the `InvalidateSQLConditionCheckByDetachingVariable` and `InvalidateSQLConditionCheckBySettingRandomValue` mutation operator. For the WebGoat model, no new AATs are generated.

4.4.1.5. Conclusion

In sum, we have seen that the presence of a message parameter is no guarantee that it is used for an exploit. A message parameter is only part of a test case, if the mutation operator injected a vulnerability that affects that parameter. This is a characteristic of Semantic Mutation Operators since they allow to ask the Security Analyst enough information so that the injection parameter can be determined. This is different for Syntactic Mutation Operators since no additional information can be provided by the Security Analyst. We will see later that heuristics are needed to operationalize some AATs generated by Syntactic Mutation Operators. Such heuristics have the consequence that more parameters are tested compared to AATs generated by Semantic Mutation Operators. When automatic black-box scanners are used, even more blindly is tested since they often apply a brute force approach to inject vulnerabilities wherever they can.

Furthermore, no test cases for XSS attacks are generated for messages that do not have parameters. This is important for the Security Analyst that generates the initial model. E.g., one has to be careful while modeling a state transition performed by a click in the browser. Since one might think that no input data is required for that transition, a click event can propagate data in the background. If such a 'hidden' data flow can be security relevant, it needs to be modeled. Otherwise, the model checker will not be able to properly verify the security properties.

Finally, subtle details like e.g., returning the user-provided message parameter rather than the parameter stored in the database determine, whether a specific test case is generated or not. This is in particular crucial for *stored* attacks. Due to the fact that the Semantic Mutation Operators represent vulnerabilities and not attacks, test cases for stored attacks (e.g., stored XSS or stored SQL attacks) are only generated if corresponding traces are possible in the model due to the vulnerability injection. Therefore, an exact and accurate modeling of the data flow is a crucial quality aspect, especially if several copies of the same data element are introduced at the model level, as motivated in the guidelines for modeling web applications (see Section 2.3.3.1).

When it comes to applying the mutation operators to the formal specification, one has to be aware that the same concept can be modeled differently. In particular, if expected AATs are not generated, one has to be aware that several mutation operators might exist that represent the same source code level vulnerability. As an example, we have demonstrated in Section 4.4.1.3 that depending on whether ASLan++ conditions or facts are used for modeling SQL queries, different Semantic Mutation Operators need to be applied to generate corresponding AATs. Finally, the same AAT might be generated by different mutation operators since the same message parameters can be involved in vulnerabilities at different locations in the specification. This has a consequence e.g., for fault localization. If a test case finds a vulnerability and this test case can be generated by a vulnerability injection at different locations, it is not easily known which vulnerability is actually present in the SUV.

4.4.1.6. Instantiation of AATs

Multi-step Stored XSS Attack

One of the more interesting and more difficult exploitation of an XSS vulnerability is the multi-step, stored XSS attack. We have seen in Section 4.3 that ZAP [49] does not find this vulnerability. In contrast, model checking the mutated WebGoat models reveals this AAT (AAT 7 in Table 4.3 generated by the `XSSRemoveSanitization` Semantic Mutation Operator). Therefore, SPaCiTE finds this vulnerability and generates a corresponding abstract AAT at the model level. Since an AAT at model level does not tell anything about the SUV, we operationalize and execute the AAT in this section. We show that SPaCiTE also successfully reveals the vulnerability at the implementation level.

The complete AAT 7 is shown in Listing 4.3. The logical steps described by the AAT are:

1. Login as user tom by providing the corresponding credentials.
2. The web application will return the welcome page that lists that the user `tom` is authorized to view its own profile.
3. `tom` request his profile.
4. The web application returns `tom`'s profile.
5. `tom` then edits and stores his own profile using malicious data, indicated by the keyword `INJECT(HTML)` in Line 6 in Listing 4.3.
6. The web application displays the modified profile.
7. `tom` returns to the welcome page.
8. After the injection, `jerry` logs in with her credentials.
9. Since he is authorized to view `tom`'s profile, the web application responses with the welcome page that contains a link to `tom`'s profile.

10. `jerry` queries `tom`'s profile and gets attacked, indicated by the keyword `VERIFY` in Line 13.

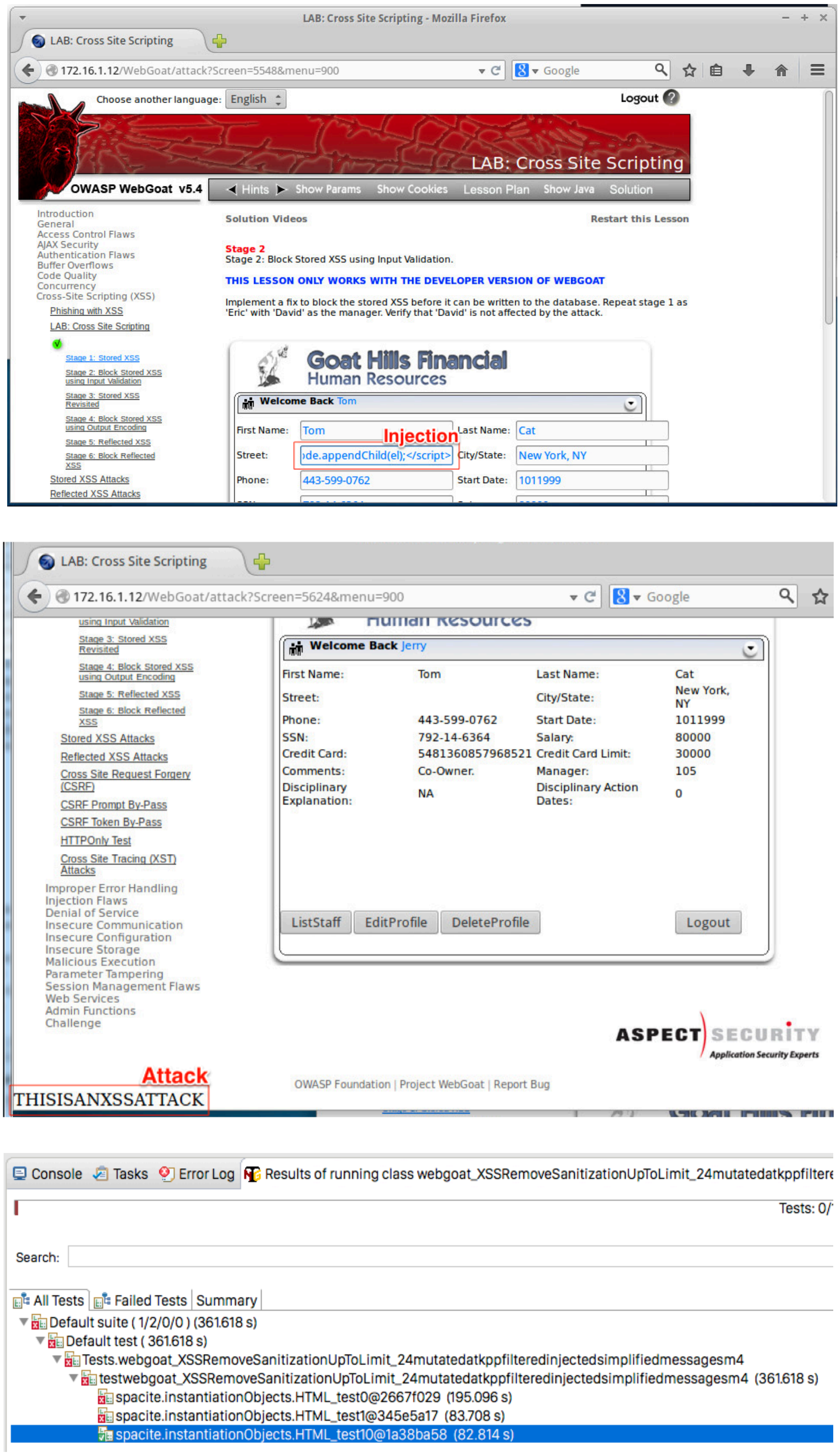
Listing 4.3: Multi-step Stored XSS Attack in WebGoat

```
1  MESSAGES:
2  <tom>      *->*  webServer  : login( username( tom ),pwd( tom ) )
3   webServer *->* <tom>       : listStaff( username( tom ) )
4  <tom>      *->*  webServer  : viewProf( username( tom ) )
5   webServer *->* <tom>       : profile( username( tom ),tom_profile )
6  <tom>      *->*  webServer  : editProf( username( tom ),INJECT( HTML ) )
7   webServer *->* <tom>       : profile( username( tom ),tom_profile )
8  <tom>      *->*  webServer  : retE2L
9   webServer *->* <tom>       : listStaff( username( tom ) )
10 <jerry>    *->*  webServer  : login( username( jerry ),pwd( jerry ) )
11  webServer *->* <jerry>     : listStaff( username( tom ) )
12 <jerry>    *->*  webServer  : viewProf( username( tom ) )
13  webServer *->* <jerry>     : VERIFY( jerry )
```

To instantiate and make this AAT operational, the Security Analyst is asked to provide a Web Application Abstract Language (WAAL) mapping from abstract messages to abstract browser actions. This mapping is given in the Appendix D.1. It starts with the declaration of the simulated agents and the configuration of the agents `tom` and `jerry`. In the `Mapping` section, abstract values like `username` and the `password` are specified. Finally, in the `MESSAGES_TO_ACTIONS` section, the mappings from abstract messages to abstract browser actions are expressed.

When initially the WAAL mapping file is linked to the formal specification using the menu **SPaCiTE Editor → Link model to WAAL mapping**, SPaCiTE generates a basic skeleton adapted to the specification. The Security Analyst needs to complete it by adding additional 74 lines (1 line for the simulated agents, 21 lines for the agent configuration, 7 lines for the mapping of abstract to concrete values, 3 lines for the profile message, 9 lines for the editProf message, 6 lines for the viewProf message, 21 lines for the login message, 3 lines for the retE2L message, and 3 lines for the listStaff message). Out of this WAAL mapping file and the reported AAT, SPaCiTE fully automatically generates a TestNG test case that consists of 188 lines of code. By making use of the WAAL DSL, the Security Analyst has to write approximately 40% of the lines compared to specifying the AAT directly as a TestNG test case. This number has to be taken with care since the actual lines of codes depend on multiple parameters — the version of Selenium, the specific source code language (Java, Python, etc.), the robustness of the implementation (error handling), the set size of unique methods involved, and so on. In addition the complexity of a code line might differ significantly. Therefore, not the exact number is important but the rough level of magnitude. In addition to writing less code, the Security Analyst can use a declarative language instead of an imperative one. That means that technical details like the exact Selenium API calls, timing errors, and raised exceptions can be encapsulated by the WAAL DSL.

Executing the corresponding TestNG test case for the AAT given in Listing 4.3 discovers the stored XSS vulnerability. The first part of Figure 4.2 shows the injection of malicious XSS input, the second part of Figure 4.2 the attack. The third part of Figure 4.2 shows that


LAB: Cross Site Scripting - Mozilla Firefox
LAB: Cross Site Scripting
172.16.1.12/WebGoat/attack?Screen=5548&menu=900
Google
Choose another language: English
Logout
LAB: Cross Site Scripting
OWASP WebGoat v5.4
Hints
Show Params
Show Cookies
Lesson Plan
Show Java
Solution
Introduction
General
Access Control Flaws
AJAX Security
Authentication Flaws
Buffer Overflows
Code Quality
Concurrency
Cross-Site Scripting (XSS)
Phishing with XSS
LAB: Cross Site Scripting
Stage 1: Stored XSS
Stage 2: Block Stored XSS using Input Validation
Stage 3: Stored XSS Revisited
Stage 4: Block Stored XSS using Output Encoding
Stage 5: Reflected XSS
Stage 6: Block Reflected XSS
Stored XSS Attacks
Reflected XSS Attacks
Solution Videos
Restart this Lesson
Goat Hills Financial
Human Resources
Injection

Stage 2
Stage 2: Block Stored XSS using Input Validation.

THIS LESSON ONLY WORKS WITH THE DEVELOPER VERSION OF WEBGOAT

Implement a fix to block the stored XSS before it can be written to the database. Repeat stage 1 as 'Eric' with 'David' as the manager. Verify that 'David' is not affected by the attack.

Welcome Back Tom

First Name:	Tom	Last Name:	Cat
Street:	›de.appendChild(el);</script>	City/State:	New York, NY
Phone:	443-599-0762	Start Date:	1011999

Welcome Back Jerry

First Name:	Tom	Last Name:	Cat
Street:		City/State:	New York, NY
Phone:	443-599-0762	Start Date:	1011999
SSN:	792-14-6364	Salary:	80000
Credit Card:	5481360857968521	Credit Card Limit:	30000
Comments:	Co-Owner.	Manager:	105
Disciplinary Explanation:	NA	Disciplinary Action Dates:	0

THISISANXSSATTACK

Results of running class webgoat_XSSRemoveSanitizationUpToLimit_24mutatedatkppfiltere

Tests: 0/

```
Default suite ( 1/2/0/0 ) (361.618 s)
  Default test ( 361.618 s)
    Tests.webgoat_XSSRemoveSanitizationUpToLimit_24mutatedatkppfilteredinjectedsimplifiedmessagesm4
      testwebgoat_XSSRemoveSanitizationUpToLimit_24mutatedatkppfilteredinjectedsimplifiedmessagesm4  (361.618 s)
        spacite.instantiationObjects.HTML_test0@2667f029  (195.096 s)
        spacite.instantiationObjects.HTML_test1@345e5a17  (83.708 s)
        spacite.instantiationObjects.HTML_test10@1a38ba58  (82.814 s)
```

Figure 4.2.: WebGoat: Stored XSS Vulnerability Disclosure

the same AAT is instantiated multiple times (see the instantiation library in Section 2.7.4.2). Since every element in the instantiation library tries to exploit the vulnerability in a different way, not all instantiation succeed. As long as there is at least one element that succeeds, the vulnerability is exploitable. For the WebGoat example, SPaCiTE successfully exploited the vulnerability since e.g., the instantiation element `HTML_text0` succeeded.

4.4.2. Semantic Mutation Operators on Wackopicko

4.4.2.1. Wackopicko.aslanpp

In the previous section we discussed in details the application of the different mutation operators for the WebGoat formal specification. In this section we focus on the Wackopicko web application. To not be repetitive, we summarize the application of the Semantic Mutation Operators. Applying the standard Semantic Mutation Operators leads to a total of 310 mutated models. Model checking all of them, 136 mutated models also lead to an AAT, where 60 traces are unique.

Note 1.
The formal specification makes use of the fact `instance_[type]([Value],[Nonce])` to differentiate instances of the same value. To simplify the discussion of the reported AATs, we get rid of the constant `instance_[type]` and the `[Nonce]` and only present the corresponding value. E.g., if an AAT contains the fact `instance_text(username1, n134( RandomValue ))` we simply write `username1`.

4.4.2.2. SQLRemoveSanitization

The Wackopicko formal specification contains in total 15 annotated sanitization blocks against SQL injections. Applying the `SQLRemoveSanitization` Semantic Mutation Operator considers every possible combination but respecting the decision diagram (Figure 2.11 on page 56). Therefore, 16’702 different mutated models are created that all need to be model checked. Model checking this amount of models is not feasible in practice. To provide a feeling about how time consuming the model-checking process can be, we model-checked, as part of Section 4.5.1, 3851 mutated models generated from the Wackopicko formal specification. That process lasts almost 312 hours. Since it is not possible to generalize and interpolate the result to 16’702 models (it is hard to predict how long the model checker needs for a specific mutated model), the number provides an intuition of the level of magnitude only about the resource intensiveness. To limit the number of generated mutated models, we apply `SQLRemoveSanitizationUpToLimit[1][2]`.

4.4.2.3. SQLRemoveSanitizationUpToLimit

Applying the `SQLRemoveSanitizationUpToLimit[1][2]` Semantic Mutation Operator generates in total 72 mutated models, where 58 generate an AAT. Filtering out syntactically equivalent AATs, we end up with a total of 32 unique AATs. Table 4.6 shows a manual classification of them. A cross (x) marks the parameters that are used for injecting malicious payloads, whereas a tick (✓) tells where the victim is attacked.

AAT	register				upload					preview	attack after				
	un	fn	ln	pw	un	tag	file	title	price		reg.	rec.	sim.	pre.	cr.
1	x										✓				
2		x									✓				
3			x								✓				
4				x							✓				
5	x	x									✓				
6	x		x								✓				
7	x			x							✓				
8		x	x								✓				
9		x		x							✓				
10			x	x							✓				
11		x											✓		
12					x										✓
13						x									✓
14							x								✓
15								x							✓
16									x						✓
17					x	x									✓
18					x		x								✓
19					x			x							✓
20					x				x						✓
21						x	x								✓
22						x		x							✓
23						x			x						✓
24							x	x							✓
25							x		x						✓
26								x	x						✓
27										x					✓
28					x					x					✓
29						x				x					✓
30							x			x					✓
31								x		x					✓
32									x	x					✓

Table 4.6.: SQLRemoveSanitization[1][2] Application on Wackopicko (un=username; fn=firstname; ln=lastname; pw=password; reg.=register; rec.=recent; sim.=similarName; pre.=preview; cr.=create; x=injection; ✓=verification)

Table 4.6 demonstrates that all inputs that are part of an SQL query are covered in a systematic way. The reported AATs for the `SQLRemoveSanitization[1][2]` cover both reflected SQL injections (see AAT 1 to 10), as well as multi-step SQL injections (see AAT 11 to 32). It is known that Wackopicko suffers from Reflected- and a Stored-SQL-Vulnerability [37]. The related paper [92] reports that the stored SQL vulnerability was not discovered by any vulnerability scanner. In contrast, SPaCiTE *does* generate a corresponding AAT (see gray row in Table 4.6). The reflected SQL injection is not reported by SPaCiTE, not because it is too difficult for finding it, but because the `login` message, where the reflected SQL vulnerability is located, is not modeled.

Considering Table 4.6, one might argue that SPaCiTE blindly generates an AAT for every possible combination, because Table 4.6 looks very regularly and systematic. To argue against this doubt, we remove the SQL annotations from the `register` message and let SPaCiTE again generate AATs. Using such a model, 21 mutated models are generated, where all of them generate an AAT. Even more, all 21 AATs are syntactically unique. Comparing the results with Table 4.6, one observes, that the first eleven AATs in Table 4.6 are not generated anymore. Another argument is that e.g., no split SQL injection is generated where both the `register` and the `upload` message is involved. This highlights, that SPaCiTE indeed operates on the security annotations and is vulnerability-based, compared to pure syntactical coverage criteria.

4.4.2.4. InvalidateSQLConditionCheckBy{DetachingVariable,SettingRandomValue}

Both of these Semantic Mutation Operators do not generate mutated models, and consequently, also no AATs are generated. Considering the formal specification given online [35], no condition was annotated with the semantics keyword `SQL`. Therefore, the above two Semantic Mutation Operators cannot be applied.

4.4.2.5. XSSRemoveSanitization

The Wackopicko formal specification contains in total 23 annotated sanitization blocks against XSS injections. Applying the `XSSRemoveSanitization` Semantic Mutation Operator considers every possible combination. Therefore, 8'388'607 different mutated models are created that all need to be model checked. This goes far beyond what is possible in practice. To limit the number of generated mutated models, we apply the `XSSRemoveSanitizationUpToLimit[2][2]` Semantic Mutation Operator, that mutates two annotated blocks per mutated model.

4.4.2.6. XSSRemoveSanitizationUpToLimit[2][2]

We directly focus on more difficult vulnerabilities like (multi-step) stored XSS attacks that require more than one mutation. Therefore, we apply the higher-order `XSSRemoveSanitizationUpToLimit[2][2]` Semantic Mutation Operator to the Wackopicko specification. In total 253 mutated models are generated, where 88 generate an AAT. Pairwise comparing them, 28 unique AATs are identified that are classified in Table 4.7. A cross (x) marks the parameters that are used for injecting malicious payloads, whereas a tick (✓) tells where the victim is attacked. Table 4.7 shows that, for good reasons, not all possible combinations of injections and verification location is considered while generating AATs. E.g., no AAT is generated for injecting malicious XSS code using the `first name` parameter of the `register` message, because according to the formal specification, the first name is not displayed anymore.

An interesting AAT is highlighted in Table 4.7. It represents the attack for the multi-step XSS attack described in the paper by Doupé et al. [92], which was not found by any vulnerability scanner. The vulnerability can be exploited by injecting malicious XSS code using the comment parameter of the `preview` message. Upon going to the most recently uploaded picture, the victim gets attacked. This AAT shows that the proposed Semantic Mutation Operators are powerful enough to discover vulnerabilities for Wackopicko that other tools often miss. In Section 4.4.2.7 we further show that the same vulnerabilities are also found on the implementation of the SUV.

4.4.2.7. Instantiation of AATs

Multi-step Stored XSS Attack

To demonstrate that SPaCiTE not only reports this issue at the model level, we let SPaCiTE operationalize and execute the highlighted AAT, given in Listing 4.4. The logical steps described by the AAT are as follows:

1. A new user registers an account on Wackopicko.
2. The webServer shows the welcome page.
3. The user checks if someone has a similar name.
4. Then the user uploads a new picture to the system.
5. After he gets asked for a comment, he provides it using the `preview` message. In this step, the user injects malicious XSS code, indicated by the keyword `INJECT(HTML)` in Line 8 in Listing 4.4.
6. Since the attack is a stored injection, the webServer shows a preview page, where the user has to click on the create button.
7. To be attacked by the injected malicious payload, the user needs to visit the page with the recently uploaded pictures and select the most recent one.
8. After the server has replied with the most recent uploaded picture, the user gets attacked, indicated by the keyword `VERIFY` in Line 12.

Listing 4.4: Multi-step Stored XSS Attack in Wackopicko
(cl=client; wS=webServer; upl=upload)

```
1 <cl> ⋆->⋆ wS : register(cl,username_cl,firstname_cl,lastname_cl,password_cl)
2 wS ⋆->⋆ <cl> : welcome(username_cl)
3 <cl> ⋆->⋆ wS : similarName
4 wS ⋆->⋆ <cl> : similarName_response(sanitize_SQL_username_t(username_cl))
5 <cl> ⋆->⋆ wS : upload(Username_upload(156),Tag_upload(156),Filename_upload(156),
```

AAT	register				upload					preview	attack after				
	un	fn	ln	pw	un	tag	file	title	price		reg.	rec.	sim.	pre.	cr.
1	x										✓				
2	x												✓		
3										x		✓			
4										x				✓	
5				x						x				✓	
6			x							x				✓	
7		x								x				✓	
8	x									x				✓	
9	x	x									✓				
10	x		x								✓				
11	x			x							✓				
12	x							x						✓	
13					x									✓	
14					x							✓			
15	x				x									✓	
16		x			x									✓	
17			x		x									✓	
18				x	x									✓	
19								x				✓			
20								x						✓	
21		x						x						✓	
22			x					x						✓	
23				x				x						✓	
24					x			x						✓	
25					x					x				✓	
26								x		x				✓	
27							x					✓			
28						x						✓			

Table 4.7.: XSSRemoveSanitization[2][2] Application on Wackopicko (un=username; fn=firstname; ln=lastname; pw=password; reg.=register; rec.=recent; sim.=similarName; pre.=preview; cr.=create; x=injection; ✓=verification)

```
                      Title_upload(156),Price_upload(156))
wS *->* <cl> : askForComment(Username_upload(156),Title_upload(156))
<cl> *->* wS : preview(INJECT(HTML))
wS *->* <cl> : show_preview(Username_upl(156),Title_upl(156),Comment_upl(156))
<cl> *->* wS : create
<cl> *->* wS : recent
wS *->* <cl> : VERIFY(cl)
```

To instantiate and making this AAT operational, the Security Analyst is asked to provide a WAAL mapping from abstract messages to abstract browser actions. This mapping is given in the Listing D.2. It starts with the declaration of the simulated agents and the configuration of the agent ‘client’. In the `Mapping` section, abstract values like `username_client`, `password_client` are specified. Finally, in the `MESSAGES_TO_ACTIONS` section, the mappings from abstract messages to abstract browser actions are expressed.

When initially the WAAL mapping file is linked to the formal specification, SPaCiTE generates a basic skeleton adapted to the specification. E.g., SPaCiTE parses the specification and creates a skeleton for each defined agent and messages. The Security Analyst needs to complete the skeleton for the Wackopicko specification by adding additional 83 lines (1 line for the simulated agents, 7 lines for the agent configuration, 9 lines for the mapping of abstract to concrete values, 6 lines each for the preview and recent message, 24 lines for the register message, 21 lines for the upload message, and 3 lines each for the create and verify message). Out of this WAAL mapping file and the reported AAT, SPaCiTE fully automatically generates a TestNG test case that consists of 183 lines of code. By making use of the WAAL DSL, the Security Analyst has to write approximately half of the lines compared to specifying the AAT directly as a TestNG test case. As already mentioned for the WebGoat model the purpose of this number is to provide an intuition.

Executing the corresponding TestNG test case for the AAT given in Listing 4.4 discovers the stored XSS vulnerability. The injected malicious XSS payload gets executed upon displaying the most recent uploaded picture (see Figure 4.3).

Stored SQL Attack

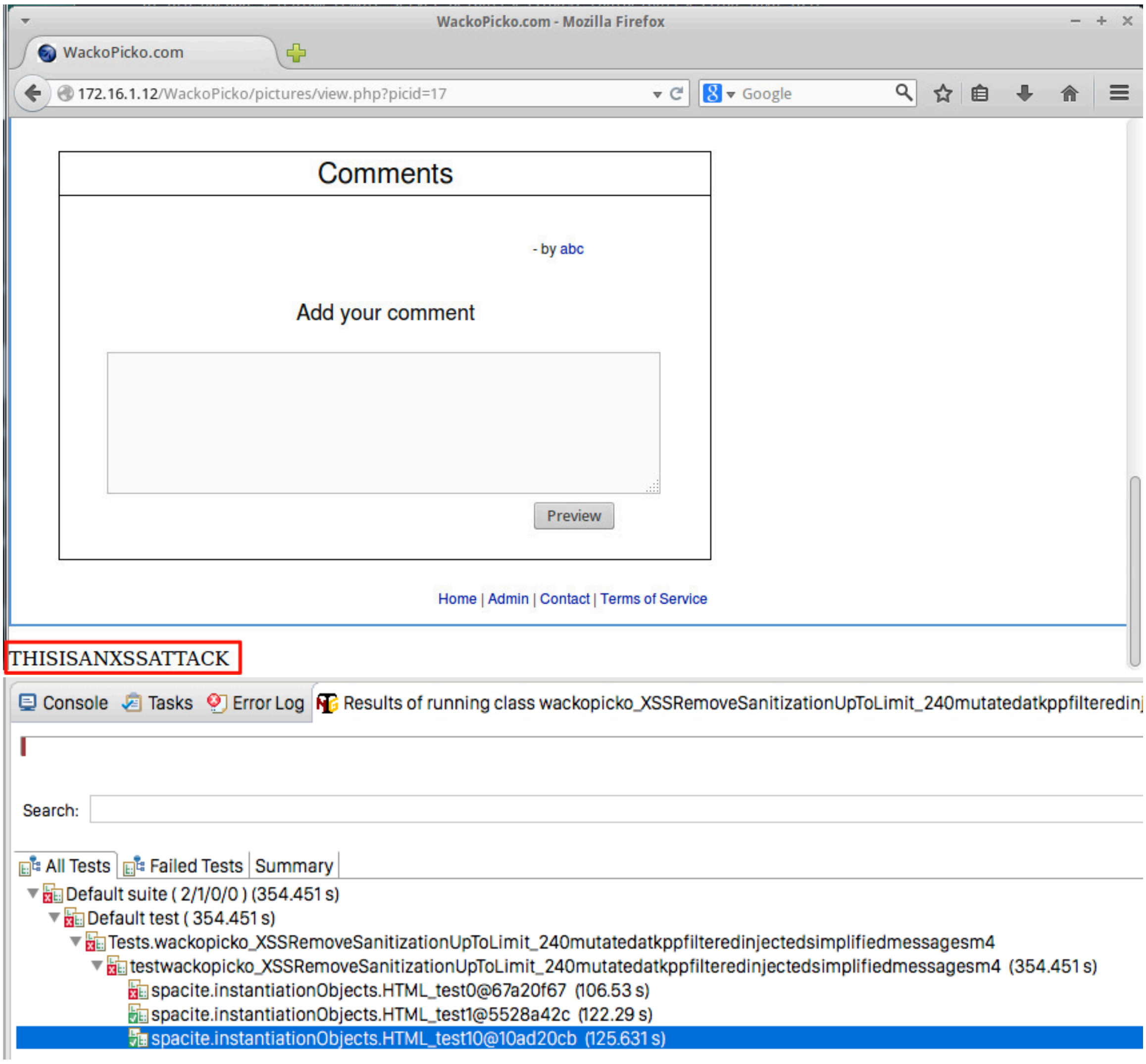
Applying the `SQLRemoveSanitization[1][2]` Semantic Mutation Operator, the multi-step stored SQL attack reported by Doupé et al. [92] is found by SPaCiTE. The corresponding AAT is shown in Listing 4.5. The logical steps described by the AAT are:

1. A new user registers an account on Wackopicko. During the registration, the attacker injects malicious SQL code into the first name field. This step is indicated by the expression `INJECT(SQL)` in Line 2.

2. The webServer shows the welcome page.

3. The user checks if someone has a similar name.

4. Upon showing users with similar names, the injected SQL code gets executed, indicated by the keyword `VERIFY` in Line 7.


WackoPicko.com - Mozilla Firefox
WackoPicko.com
172.16.1.12/WackoPicko/pictures/view.php?picid=17
Google
Comments
- by abc
Add your comment
Preview
Home | Admin | Contact | Terms of Service
Console
Tasks
Error Log
Search:
All Tests
Failed Tests
Summary

THISISANXSSATTACK

Results of running class wackopicko_XSSRemoveSanitizationUpToLimit_240mutatedatkppfilteredinj

```
Default suite ( 2/1/0/0 ) (354.451 s)
  Default test ( 354.451 s)
    Tests.wackopicko_XSSRemoveSanitizationUpToLimit_240mutatedatkppfilteredinjectedsimplifiedmessagesm4
      testwackopicko_XSSRemoveSanitizationUpToLimit_240mutatedatkppfilteredinjectedsimplifiedmessagesm4  (354.451 s)
        spacite.instantiationObjects.HTML_test0@67a20f67  (106.53 s)
        spacite.instantiationObjects.HTML_test1@5528a42c  (122.29 s)
        spacite.instantiationObjects.HTML_test10@10ad20cb  (125.631 s)
```

Figure 4.3.: Wackopicko: Stored XSS Vulnerability Disclosure

Listing 4.5: Multi-step Stored SQL Attack in Wackopicko (cl=client; wS=webServer)

```
<client> *->* webServer : register(client,username_client,
                                   INJECT(SQL),
                                   lastname_client,
                                   password_client)
webServer *->* <client> : welcome(username_client)
<client> *->* webServer : similarName
webServer *->* <client> : VERIFY(client)
```

To instantiate the reported multi-step stored SQL AAT, the minimal needed WAAL mapping file consists of 47 lines. E.g., the mapping for the `upload` message is not needed in this example. Applying the provided WAAL mapping (see Appendix D.2) to the reported multi-step stored SQL AAT, SPaCiTE generates a TestNG test case that consists of 129 lines of code. In this example, the manually written WAAL mapping file is about 37% of the size of the TestNG file.

The WAAL mapping is specified in a reusable way because actual parameter values for a message are referenced instead of hard-coded (see Definition 4 on page 83). That allows to reuse the same mapping for different AATs. Since we have already provided a WAAL mapping for the stored XSS AAT we can just reuse the same mapping instead of creating a new one.

Executing the corresponding TestNG test case for the AAT given in Listing 4.5 discovers the stored SQL vulnerability. The injected malicious SQL payload gets executed upon displaying users with similar names. The purpose of the used malicious SQL payload is to make the executed SQL query syntactically wrong, so that the intended functionality cannot be executed anymore. The output of Wackopicko is shown in Figure 4.4.

4.4.2.8. Conclusion

In this section we have seen that applying Semantic Mutation Operators without a limit of possible combinations very quickly leads to an amount of mutated models that is not model-checkable anymore in practice. Higher-order Semantic Mutation Operators generate abstract AATs that represent non-trivial vulnerabilities that are not found by any black-box vulnerability scanner studied by Doupé et al. [92]. Finally, the TEE and the WAAL language are powerful enough to successfully execute the non-trivial XSS and SQL injections on the SUV.

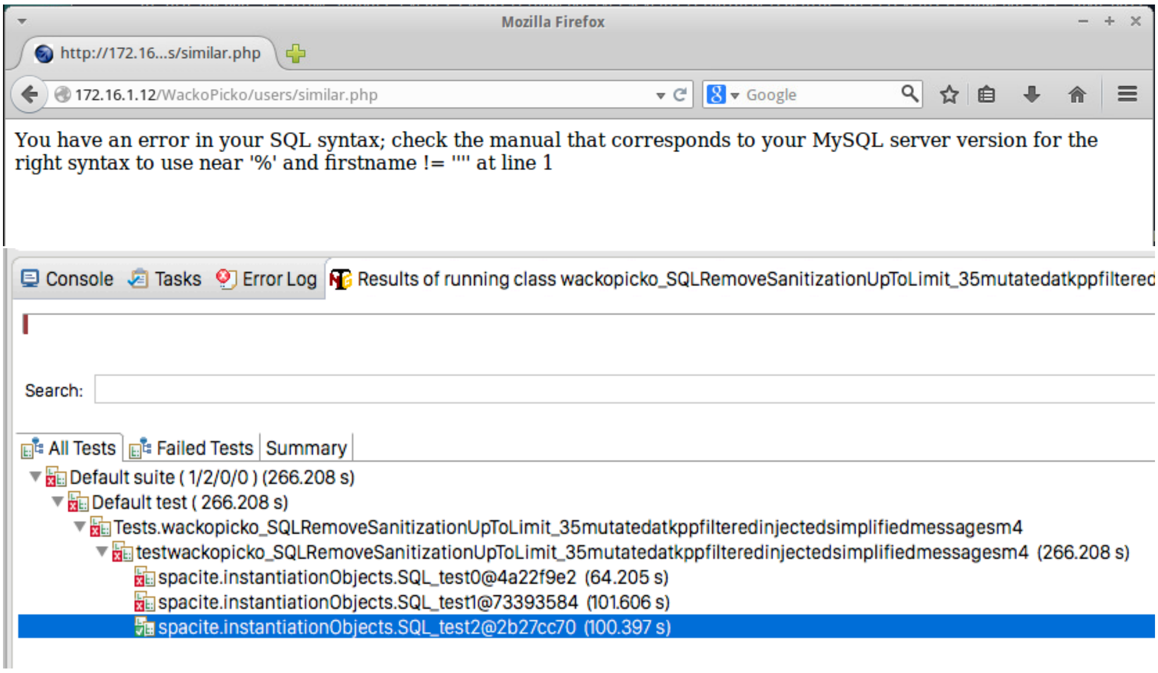
4.4.3. Semantic Mutation Operators on Bank Application

4.4.3.1. XSSRemoveSanitizationUpToLimit

Finally, we apply all standard Semantic Mutation Operators, that consider up to two mutation candidates per mutated model, to the Bank application formal specification, given online [7]. Applying them leads to a total of 248 mutated models. Model-checking of all mutated models is quite time consuming and last approximately 29 hours. Table 4.8 shows all reported AATs. In total, 62 mutated models lead to an AAT. Syntactically comparing them, 33 AATs are unique.

AAT	**loginClient**		**loginAdmin**		**actC**	**register**					**check after**			
	un	pwd	un	pwd	id	un	fn	ln	em	pwd	logA	logC	reg.	actC
1						XSS					✓			
2							XSS				✓			
3								XSS			✓			
4									XSS		✓			
5			SQL								✓			
6				SQL							✓			
7			SQL	SQL							✓			
8	SQL											✓		
9		SQL										✓		
10	SQL	SQL										✓		
11						SQL							✓	
12							SQL						✓	
13								SQL					✓	
14									SQL				✓	
15										SQL			✓	
16						SQL	SQL						✓	
17						SQL		SQL					✓	
18						SQL			SQL				✓	
19						SQL				SQL			✓	
20							SQL	SQL					✓	
21							SQL		SQL				✓	
22							SQL			SQL			✓	
23								SQL	SQL				✓	
24								SQL		SQL			✓	
25									SQL	SQL			✓	
26						SQL								✓
27					SQL									✓

Table 4.8.: Semantic Mutation Operator on Bank Application (logC=loginClient; logA=loginAdmin; actC=activateClient; un=username; pwd=password; fn=first name; ln=last name; em=email;)



Mozilla Firefox
http://172.16...s/similar.php
172.16.1.12/WackoPicko/users/similar.php
Google
Console
Tasks
Error Log
Results of running class wackopicko_SQLRemoveSanitizationUpToLimit_35mutatedatkppfiltered
Search:
All Tests
Failed Tests
Summary

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '%' and firstname != "" at line 1

```
Default suite ( 1/2/0/0 ) (266.208 s)
  Default test ( 266.208 s)
    Tests.wackopicko_SQLRemoveSanitizationUpToLimit_35mutatedatkppfilteredinjectedsimplifiedmessagesm4
      testwackopicko_SQLRemoveSanitizationUpToLimit_35mutatedatkppfilteredinjectedsimplifiedmessagesm4  (266.208 s)
        spacite.instantiationObjects.SQL_test0@4a22f9e2  (64.205 s)
        spacite.instantiationObjects.SQL_test1@73393584  (101.606 s)
        spacite.instantiationObjects.SQL_test2@2b27cc70  (100.397 s)
```

Figure 4.4.: Wackopicko: Stored SQL Vulnerability Disclosure

We will not discuss every AAT in Table 4.8 but only highlight a few observations.

- AATs 1 to 4 are generated by the `XSSRemoveSanitization` Semantic Mutation Operator. Although the register request has five parameters, no test case, for good reasons, for an XSS attack using the password parameter is generated. The password is never displayed in the browser and therefore, also no AAT is generated for that parameter.

- The parameters of the `register` message are not combined to generate split XSS attacks. The rationale is that such attacks require four mutation candidates to be mutated at the same time. Both parameters must not be sanitized during the `register` message and in addition, they must not be sanitized during the `loginAdmin` message. Since the mutation operators only consider up to two mutation candidates at the same time, these AATs are missed.

- `InvalidateSQLConditionCheckByDetachingVariable` generates in total 15 mutated models from which 12 lead to a security property violation. Analyzing all reported AATs, these 12 AATs can be reduced to 6 unique AATs (AAT 5 - 10 in Table 4.8). They all exploit one or two parameters of the `loginAdmin` and `loginClient` message. Exemplary, AAT 5 (for AAT 5-7) and AAT 8 (for AAT 8 - 10) are shown in Listing 4.8 and Listing 4.9 respectively. Please note that they significantly differ in terms of the length. While an administrator can login at any time, he immediately logs in when a pending activation action is available. In contrast, a client can only login after the activation by an administrator. Therefore, AAT 8 - 10 contain these steps before the injection is performed. It is important to stress that the decision to let an admin only login when an activation request is available, is mainly model-checker motivated since it helps to reduce the size of the state space of the formal specification. At a model level, a `loginClient` message is only accepted by the web application if the user account is activated. Therefore, such an account has to be registered and activated first.

- The `InvalidateSQLConditionCheckByDetachingVariable` mutation operator does not generate mutated models exploiting parameters of the `activateClient` and `register` message that lead to an SQL attack. For these two message handlers, the `SQLRemoveSanitization` mutation operator is required. The rationale is that SQL related operations are not modeled as part of the message receiving condition (see the complete specification [7] or Listing 4.6).

- `SQLRemoveSanitization` is generating 5120 mutated models. Due to the impracticality of model checking such an amount, we apply the Semantic Mutation Operator `SQLRemoveSanitizationUpToLimit[1][2]`. It generates AATs 11 - 27. Among these AATs we want to highlight AAT 26 and 27.

  AAT 26. This AAT represents stored SQL injections using the `register` message to inject the malicious payload into the system, and the `activateClient` message to let the system execute the injected malicious code. A stored SQL injection is possible for the `activateClient` message since `activateClient` first reads the data from the database using a first SQL query, and then uses the stored username to activate that username during a second SQL query. Therefore, the stored SQL injection exploits the seconds SQL query of the `activateClient` message. Listing 4.10 shows AAT 26.

  AAT 27. This AAT represents a reflected SQL injections using the `activateClient` message. We have seen in Section 4.2.3 that the `activateClient` message handler consists of two SQL queries. While AAT 26 exploits the second SQL query (and is therefore a stored SQL injection), this AAT exploits the first SQL query (and is therefore a reflected injection). This AAT is shown in Listing 4.11.

Listing 4.6: Bank Application Model Extract

```
on (?User *->* Actor : activateClient(
    inst_u_t(?ActivateClientUsername,?RandomValue1Activate) ) &
    authenticatedAsAdmin(?User) &
    waiting_for_activation( instance_username_t(?ActivateClientUsername,?) )
): { ... }
```

Listing 4.7: Stored Multi-step Attack on the Bank Application (wB=webBrowser; wS=webServer)

```
<wB> *->*  wS   : register(INJECT(HTML),firstname1,lastname1,email1,password1)
 wS  *->* <wB> : ack_register
<wB> *->*  wS   : loginAdmin(username_admin,password_admin)
 wS  *->* <wB> : VERIFY(wB)
```

Listing 4.8: AAT 5 of Table 4.8
(wS=webServer; wB=webBrowser)

```
<wB> ⋆->⋆  wS  : register(username1,firstname1,lastname1,email1,password1)
 wS  ⋆->⋆ <wB> : ack_register
<wB> ⋆->⋆  wS  : loginAdmin(INJECT( SQL ),password_admin)
 wS  ⋆->⋆ <wB> : VERIFY(wB)
```

Listing 4.9: AAT 8 of Table 4.8
(wS=webServer; wB=webBrowser; pwd_admin=password_admin)

```
<wB> ⋆->⋆  wS  : register(username1,firstname1,lastname1,email1,pwd_admin)
 wS  ⋆->⋆ <wB> : ack_register
<wB> ⋆->⋆  wS  : loginAdmin(username_admin,pwd_admin)
 wS  ⋆->⋆ <wB> : listPendingAccounts(username1,firstname1,lastname1,email1)
<wB> ⋆->⋆  wS  : activateClient(username1)
 wS  ⋆->⋆ <wB> : ack_activateClient
<wB> ⋆->⋆  wS  : logout
<wB> ⋆->⋆  wS  : loginClient(username1,INJECT( SQL ))
 wS  ⋆->⋆ <wB> : VERIFY(wB)
```

Listing 4.10: AAT 26 of Table 4.8

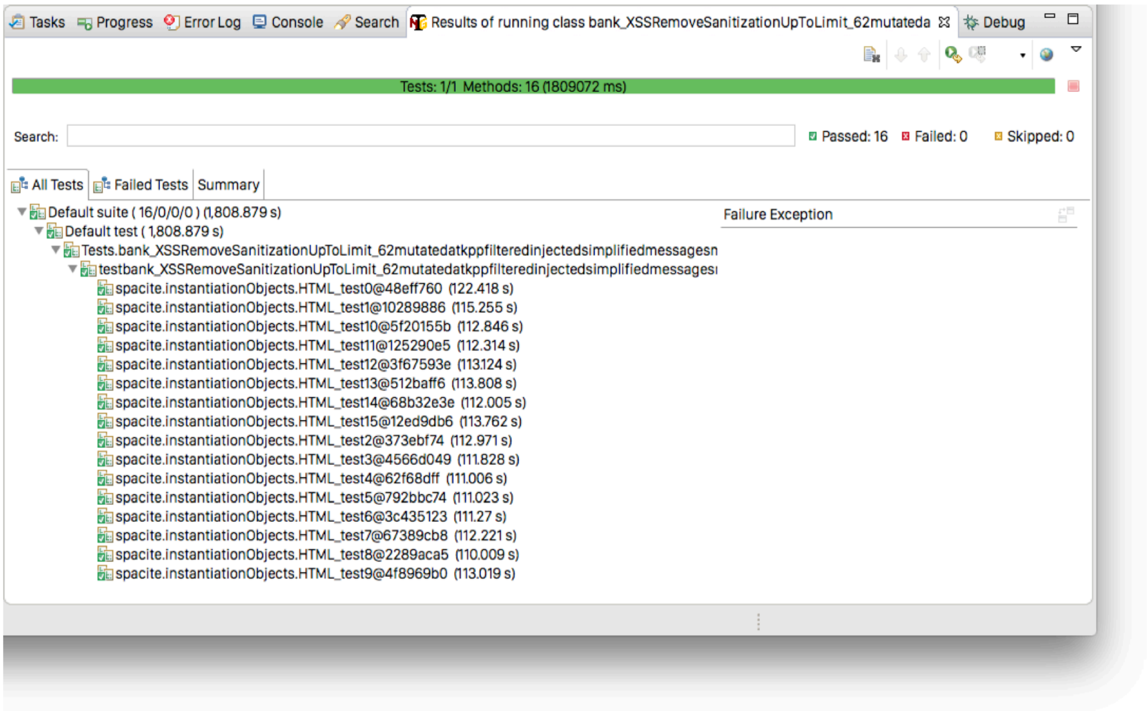
```
<wB> ⋆->⋆  wS  : register(INJECT(SQL),firstname1,lastname1,email1,password1)
 wS  ⋆->⋆ <wB> : ack_register
<wB> ⋆->⋆  wS  : loginAdmin(username_admin,password_admin)
 wS  ⋆->⋆ <wB> : listPendingAccounts(username1,firstname1,lastname1,email1)
<wB> ⋆->⋆  wS  : activateClient(username1)
 wS  ⋆->⋆ <wB> : VERIFY(wB)
```

Listing 4.11: AAT 27 of Table 4.8

```
<wB> ⋆->⋆  wS  : register(username1,firstname1,lastname1,email1,password1)
 wS  ⋆->⋆ <wB> : ack_register
<wB> ⋆->⋆  wS  : loginAdmin(username_admin,password_admin)
 wS  ⋆->⋆ <wB> : listPendingAccounts(username1,firstname1,lastname1,email1)
<wB> ⋆->⋆  wS  : activateClient(INJECT(SQL))
 wS  ⋆->⋆ <wB> : VERIFY(wB)
```

4.4.3.2. Instantiation of AATs

As discussed in Section 4.3, the students of the Secure Coding class used ZAP, w3af, Nikto, Burp, Grendel, and InjectMe to test the application for known vulnerabilities. In terms of XSS and SQL vulnerabilities, they didn't find any issues. In this section we operationalize the reported AAT from Section 4.4.3. The required WAAL mapping is given in Appendix D.3. Executing all test cases, SPaCiTE did not discover any XSS and SQL security issues. Figure 4.5 shows an exemplary reported verdict.



```
Tests: 1/1  Methods: 16 (1809072 ms)

Passed: 16   Failed: 0   Skipped: 0

▼ Default suite ( 16/0/0/0 ) (1,808.879 s)
  ▼ Default test ( 1,808.879 s)
    ▼ Tests.bank_XSSRemoveSanitizationUpToLimit_62mutatedatkppfilteredinjectedsimplifiedmessagesn
      ▼ testbank_XSSRemoveSanitizationUpToLimit_62mutatedatkppfilteredinjectedsimplifiedmessagesı
          spacite.instantiationObjects.HTML_test0@48eff760  (122.418 s)
          spacite.instantiationObjects.HTML_test1@10289886  (115.255 s)
          spacite.instantiationObjects.HTML_test10@5f20155b  (112.846 s)
          spacite.instantiationObjects.HTML_test11@125290e5  (112.314 s)
          spacite.instantiationObjects.HTML_test12@3f67593e  (113.124 s)
          spacite.instantiationObjects.HTML_test13@512baff6  (113.808 s)
          spacite.instantiationObjects.HTML_test14@68b32e3e  (112.005 s)
          spacite.instantiationObjects.HTML_test15@12ed9db6  (113.762 s)
          spacite.instantiationObjects.HTML_test2@373ebf74  (112.971 s)
          spacite.instantiationObjects.HTML_test3@4566d049  (111.828 s)
          spacite.instantiationObjects.HTML_test4@62f68dff  (111.006 s)
          spacite.instantiationObjects.HTML_test5@792bbc74  (111.023 s)
          spacite.instantiationObjects.HTML_test6@3c435123  (111.27 s)
          spacite.instantiationObjects.HTML_test7@67389cb8  (112.221 s)
          spacite.instantiationObjects.HTML_test8@2289aca5  (110.009 s)
          spacite.instantiationObjects.HTML_test9@4f8969b0  (113.019 s)
```

Figure 4.5.: Bank: Exemplary Verdict of Testing the Bank Application

4.4.3.3. Summary

Verification techniques contribute with the important task of following the application logic when generating test cases. Providing the steps between an injection and the attack, or providing a correct sequence to bring the system into the state where an injection can be performed is the task of the used verification techniques. By considering traces of the behavior specification, such traces are application-dependent and contribute to application specific injection and attack locations. As an example, we have seen in this section that due to the model checker, the test case AAT 8 first let an administrator activate the client account before the injection is performed.

4.4.4. Effectiveness Conclusion

In Section 1.4 we discussed the hypothesis that in a model-based context, non-trivial security vulnerabilities for web applications can be found with mutation operators and fault injections. In Section 4.1 we raised the research question: *If non-trivial security vulnerabilities are present, does SPaCiTE, that uses vulnerability-based fault models, find in practice a higher number of these vulnerabilities than black-box vulnerability scanners?* To answer this question, we have seen in Section 4.4 that SPaCiTE successfully discovered the stored XSS vulnerability in the WebGoat application, as well as the multi-step stored XSS and stored SQL vulnerability in the Wackopicko web application. The comparison with black-box vulnerability scanners showed that all these three vulnerabilities have not been discovered by such scanners.

4.5. Efficiency Evaluation

We have seen in the previous section that SPaCiTE generates test cases for security-interesting non-trivial XSS and SQL vulnerabilities that black-box vulnerability scanner often miss. Executing test cases on a SUV faces a lot of technical and practical problems that can prevent a tool to successfully execute such test cases. We have argued in Section 4.1 that black-box vulnerability scanners use syntax-based fault-models for generating test cases. To compare vulnerability-based and syntax-based fault models also on a more conceptual level, we consider the three ASLan++ formal specifications introduced in Section 4.2 and compare the generated AATs by Syntactic and Semantic Mutation Operators to highlight differences and potential issues. While Semantic Mutation Operators represent a vulnerability-based fault model, Syntactic Mutation Operators represent a syntax-based fault model.

4.5.1. Syntactic Mutation Operators

When using a mutation-based approach and verification techniques like model checkers to generate AATs, first a set of mutated models is generated. Then all models in this set have to be model checked to generate corresponding AATs. In the ideal case, one only wants to model check whose mutated models, that also generate an AAT, because (1) model checking is quite resource intensive, and (2) mutated models that do not generate an AAT cannot be used for test case generation in our approach. Furthermore, semantically annotated models require manual work from the Security Analyst. In contrast, Syntactic Mutation Operators can automatically be applied without annotations. For the efficiency evaluation, we therefore compare Semantic Mutation Operators with Syntactic Mutation Operators.
Syntactic Mutation Operators are based on the same set of General Mutation Operators (see Section 2.6.4) as the Semantic Mutation Operators. Nevertheless they differ in the following sense:

- Syntactic Mutation Operators are applied in the formal specification everywhere the ASLan++ syntax allows it. It is the syntactic structure of a formal specification that motivates the application of Syntactic Mutation Operators. As a reminder, Semantic Mutation Operators only apply a mutation operator to those parts of the specification where a source code vulnerability might occur in the implementation.

- A second difference is that Syntactic Mutation Operators lack an immediate and obvious connection to semantic information like corresponding vulnerabilities. Therefore, they can only exploit syntactic changes to determine the malicious parameters (see Section 2.6.4) whereas Semantic Mutation Operators do have access to semantic annotations. We will see in this section that this has consequences during the operationalization of AATs.

For this evaluation, we consider the following four Syntactic Mutation Operators: DetachVariable, RemoveFunctionFromAssignment, ReplaceVariableByRandomValue, and InvalidateIntroduceFact. The functionality is already discussed in Section 2.6.4. To discuss the reported AATs by applying Syntactic Mutation Operators, we make the following assumption:

Assumption 1. *SPaCiTE is not always able to determine which message parameter has to be used to inject malicious inputs. In this situation, SPaCiTE does not introduce an explicit 'VERIFY' action that tells the TEE to check if the malicious effect can be observed. If the 'VERIFY' action is missing, we assume that the check is performed after the last message of the AAT.*

We take the same three formal specifications as for the effectiveness evaluation (see Section 4.2) and apply both Syntactic and Semantic Mutation Operators. The results are reported in Table 4.9. The different columns represent the three formal specifications. For each specification we list the number of generated mutated models (mM), the number of AATs (A) and the ratio between the first two columns (mM/A). A row represents the different mutation operators that are applied (syntactic, semantic, first-order, higher-order). From this table we highlight the following observations that we later discuss in Chapter 5:

- Applying Syntactic Mutation Operators lead to more generated AATs than applying Semantic Mutation Operator. This means that certain traces are not tested following the semantics-based approach. Nevertheless, the difference in terms of the absolute number of AATs is lower than expected. While higher-order Semantic Mutation Operators generate 60 unique AATs from the Wackopicko specification, higher-order Syntactic Mutation Operators generate 94. This is a ratio of around 1.6. The ratio between first-order Syntactic and Semantic Mutation Operators is between 1.29 (18 vs. 14 AATs) and 2 (20 vs. 10 AATs).

- While all different mutation operator sets, except the set of higher-order Syntactic Mutation Operators, generate between 19 and 310, the set of higher-order Syntactic Mutation Operators generate between 3820 and 12'046 AATs. In the context of model-checking this difference is significant. Model checking thousands of models requires hours of computation. While model-checking 3851 models generated by the higher-order Syntactic Mutation Operators using the configuration given in Section 4.2 and eight parallel processes required 312 hours, model checking 248 specifications generated by the `XSSRemoveSanitizationUpToLimit` mutation operator required 29 hours.

- The ratio between generated mutated specifications and the fraction of mutated specifications that generate AATs is much smaller for Syntactic Mutation Operators (between 2% (94 vs. 3851 models) and 13% (20 vs. 158 models)). The same ratio is much better for Semantic Mutation Operators (between 12% (24 vs. 204 models) and 53% (10 vs. 19 models)).

Wackopicko. To compare the set of AATs of all four different combinations (first-order, higher-order, Syntactic Mutation Operator, Semantic Mutation Operator) in more detail, we first focus on the Wackopicko use case. We syntactically compare the AATs of each set and report the findings in Table 4.10. The table has to be read in the following way. Each row represents a specific set of mutation operators. The diagonal elements show the number of unique AATs that a specific set generates. The different columns show how many of the reported AATs are found by the different sets of mutation operators as well. E.g., the set of first-order Syntactic Mutation Operators generate 18 unique AATs. 14 out of these 18 AATs are generated by the first-order Semantic Mutation Operators as well. In a nutshell, we expected that Syntactic Mutation Operators subsume Semantic Mutation Operators. Tables 4.10 and 4.11 show that Syntactic Mutation Operators generate AATs that are challenging when determining the malicious input parameter. E.g., higher-order Syntactic Mutation Operators only generate 52 out of these 60 AATs generated by higher-order Semantic Mutation Operators (Table 4.10). Furthermore, Table 4.11 shows that Syntactic Mutation Operators generate 6 out of 10 AATs as well but SPaCiTE fails to determine the input parameter of 4 out of 10 AATs. In the following, we will analyze these issues.

	WebGoat			Wackopicko			Bank		
	mM	A	mM/A	# mM	# AATs	mM/A	# mM	# AATs	mM/A
Syntactic first-order									
InvalidateIntroduceFact	53	9	0.17	42	4	0.10	59	8	0.14
RemoveFctFromAssign.	8	0	0.00	36	10	0.28	23	0	0.00
DetachVariable	38	15	0.39	41	8	0.20	74	5	0.07
SetToRandomValue	59	0	0.00	56	0	0.00	122	0	0.00
unique artifacts	158	20	0.13	175	18	0.10	278	12	0.04
Syntactic higher-order (min=max=2)									
InvalidateIntroduceFact	1378			861	155	0.18	1711		
RemoveFctFromAssig.	28			630	300	0.48	253		
DetachVariable	703			820	275	0.34	2701		
SetToRandomValue	1711			1540	0	0.00	7381		
unique artifacts	3820			3851	94	0.02	12046		
Semantic first-order									
InvSQLByDetachingVar	4	4	1.00	0	0	0.00	7	4	0.57
InvSQLByRandom	4	0	0.00	0	0	0.00	7	0	0.00
SQLRemoveSanitization	3	3	1.00	15	10	0.67	13	6	0.46
XSSRemoveSanitization	8	3	0.38	23	4	0.17	17	0	0.00
unique artifacts	19	10	0.53	38	14	0.37	44	9	0.20
Semantic higher-order (min=max=2)									
InvSQLByDetachingVar	1	1	1.00	0	0	0.00	6	6	1.00
InvSQLByRandom	1	0	0.00	0	0	0.00	6	0	0.00
SQLRemoveSanitization	1	1	1.00	57	48	0.84	56	43	0.77
XSSRemoveSanitization	28	21	0.75	253	88	0.35	136	3	0.02
unique artifacts	31	13	0.42	310	60	0.19	204	24	0.12

Table 4.9.: Application of Syntactic and Semantic Mutation Operators with Different Configurations
(mM=# mutated Models; A=# AATs InvSQLByDetachingVar=InvalidateSQLConditionCheckByDetachingVariable; InvSQLByRandom=InvalidateSQLConditionCheckBySettingRandom)

From this Table 4.10 we conclude the following observations:

- The first-order Syntactic Mutation Operator generate 18 AATs that are all generated by the higher-order Syntactic Mutation Operators as well. This is not astonishing since

Set	Also generated by			
	Syn1.	Syn2.	Sem1.	Sem2.
First-order Syntactic	18	18	14	14
Higher-order Syntactic	18	94	14	52
First-order Semantic	14	14	14	14
Higher-order Semantic	14	52	14	60

Table 4.10.: Comparison of Syntactic vs. Semantic Mutation Operator (Syn1=first-order Syntactic Set; Syn2=Higher-order Syntactic Set; Sem1=first-order Semantic Set; Sem2=Higher-order Semantic Set)

first-order and higher-order mutation operators do not differ in terms of functionality but in terms of the number of mutation candidates that they consider for the same mutated formal specification. 14 out of these 18 AATs are generated by first-order and higher-order Semantic Mutation Operators as well. The rationale for first-order vs. higher-order Syntactic Mutation Operators is the same for first-order vs. higher-order Semantic Mutation Operators.

- The higher-order Syntactic Mutation Operators generate in total 94 AATs. In this situation, this set of AATs contains traces that are not generated by any other set. Around 19% of the 94 traces are generated by the first-order Syntactic Mutation Operators as well, even less are generated by first-order Semantic Mutation Operator (around 15%). In comparison, around 56% of the traces are also generated by the higher-order Semantic Mutation Operator.

- The first-order Semantic Mutation Operators generate in total 14 AATs. All of these AATs are generated by all other sets as well.

- Finally, the higher-order Semantic Mutation Operator generates 60 AATs in total, from which 14 AATs are also generated by the first-order Semantic Mutation Operators. Interestingly, the higher-order Syntactic Mutation Operators generate a subset only (52 out of 60). We will analyze and discuss the rationales in Chapter 5. Furthermore, the first-order Semantic Mutation Operators generate 14 out of the 60 AATs.

WebGoat. Analyzing the generated AATs by first-order Syntactic vs. Semantic Mutation Operators in Table 4.11 on the WebGoat formal specification shows similar issues. Table 4.11 has to be read in the following way: The first column lists all 20 AATs generated by the set of first-order Syntactic Mutation Operators. The third column shows all 10 AATs generated by the set of first-order Semantic Mutation Operators. Each row states, whether the trace in the first column is syntactically equivalent to the trace in the third column, or which of the three issues *Issue1*, *Issue2*, or *Issue3* occurs.

Issue1. Message parameter to be used for the injection can be determined but the type of the injection is missing.

Issue2. The message parameter for the injection cannot be determined.

Issue3. The ID to identify the acting client is considered malicious. This mutation has no real-word corresponding vulnerability.

Table 4.11 shows that 6 AATs generated by first-order Semantic Mutation Operators are syntactically equivalent to 6 AATs generated by first-order Syntactic Mutation Operators. AATs 0, 2, 8, and 17 generated by the first-order Syntactic Mutation Operators correspond to the remaining 4 AATs generated by the first-order Semantic Mutation Operators. For those four AATs, SPaCiTE is able to identify the correct message parameter, but since the violated security property is related to *Authentication*, the type of malicious input cannot be determined (Issue1). For 9 AATs generated by the first-order Syntactic Mutation Operators, SPaCiTE is not able to determine the message parameter (Issue2). Finally, the last AAT (AAT 7), an identifier that refers to the client performing the action in the browser is considered as malicious. Since there is no underlying real world vulnerability, this AAT is only generated by the set of Syntactic Mutation Operators (Issue3).

First-order Syntactic Mutation Operators		**First-order Semantic Mutation Operators**
0	Issue1	0
1	Issue2	
2	Issue1	8
3	Issue2	
4	syntactic equivalent	4
5	syntactic equivalent	2
6	syntactic equivalent	5
7	Issue3	
8	Issue1	7
9	Issue2	
10	Issue2	
11	Issue2	
12	Issue2	
13	Issue2	
14	Issue2	
15	Issue2	
16	syntactic equivalent	6
17	Issue1	1
18	syntactic equivalent	3
19	syntactic equivalent	9

Table 4.11.: Comparison of AATs Generated by First-Order Syntactic vs. First-order Semantic Mutation Operators

4.5.2. Efficiency Conclusion

In sum, comparing Semantic Mutation Operators (vulnerability-based fault model) vs. Syntactic Mutation Operators (syntax-based fault model) using the three models introduced in Section 4.2 shows that applying Syntactic Mutation Operators ‘only’ generate between 1.29 (18 vs. 14 AATs) and 2 (20 vs. 10 AATs) times as many AATs as applying Semantic Mutation Operators. The ratio between all mutated specifications and specifications that generate an AAT is very low for Syntactic Mutation Operators (between 2% (94 vs. 3851 models) and 13% (20 vs. 158 models)), and much higher for Semantic Mutation Operators (between 12% (24 vs. 204 models) and 53% (10 vs. 19 models)). This demonstrates that Syntactic Mutation Operators generate a lot of mutated specifications where the used model checker cannot find a trace that violates the specified security properties. Since only mutated specifications that generate an AAT are useful for test case generation, all the other mutated specifications consume computation power without generating test cases. Semantic annotations help to prevent some of these mutated models by providing information about the underlying used technology and the semantics of the annotated artifact. Finally comparing reported AATs generated by Syntactic and Semantic Mutation Operators highlights three issues with AATs generated by Syntactic Mutation Operators — (1) missing vulnerability information, (2) difficulties during determining which message parameter has to be used for the injection, and (3) AATs that cannot be instantiated. This has the consequence that the operationalization is not obvious and heuristics need to be applied to execute them. We will discuss such heuristics in Section 5.5. Since crucial operationalization information is missing, such heuristics often implement an over-approximation and decrease the efficiency of generating executable test cases. The improvements of the Semantic Mutation Operators are not for free since Semantic Mutation Operator require manual annotations. We will discuss the required effort in more details in Section 5.6.

5. Discussion and Future Work

5.1. Syntactic vs. Semantic Mutation Operators

Table 4.9 shows the application and model checking of three different models (columns) and different mutation operators (rows). From this table we conclude the following:

The ratio of the number of Abstract Attack Traces (AATs) generated by Syntactic Mutation Operators divided by the number of AATs generated by Semantic Mutation Operators is smaller than expected. Rationales for this observation are:

- Our considered formal specifications consist up to eleven different message parameters, indicated by P1 ... P11 on the x-axis in Table 5.1. On the y-axis we see all defined messages. An 'X' and a 'S' in the intersection of a message and a parameter P indicates that the parameter is used for an Cross-site Scripting (XSS) and Structured Query Language (SQL) injection respectively. E.g., the 'X' in the intersection of `login(P1,P2)` and `P1` means that a test case was generated that injected an XSS malicious input using the first parameter of the login message. The table shows that depending on the model, between 40% and 90% of all the modeled input parameters are involved in XSS related security properties and 100% are involved in SQL injection related security properties (see Table 5.1). Comparing these two numbers, one observes that the number of parameters involved in XSS attacks is significantly smaller than the number of parameters involved in SQL attacks. Having a closer look to the corresponding formal specifications, it turns out that all parameters that are sent back to the browser are involved in XSS attacks (percentage numbers in parentheses in Table 5.1). Therefore, a Syntactic Mutation Operator cannot exploit an additional message parameter for violating a security property since all of them are already considered by Semantic Mutation Operators.

- This behavior is supported by the fact that the considered formal specifications mainly focus on those message handlers, that are relevant in terms of XSS and SQL attacks. Communication that is unrelated to these vulnerabilities and message exchanges that are abstracted, are left out by these models. Therefore, most of the modeled parameters are relevant in terms of XSS and SQL vulnerabilities and are already covered by AATs generated by Semantic Mutation Operators.

- Important to stress is that a mutation only leads to a test case if a security property is violated. E.g., if we consider a message with 'x' different message parameters, but none of them are rendered in the browser, injecting XSS related malicious code will also never generate an AAT. Therefore, even if a Syntactic Mutation Operator is applied to parts of the specification related to these parameters, it is very unlikely that such mutations generate an AAT. Furthermore, by annotating the formal specification, the Security Analyst is eager to cover as many parts of the specification that correlate with the defined security properties. Therefore, it is very likely that the Security Analyst has ‘optimized’ the annotations for covering the security properties, meaning that all parts that are affected by the properties are also annotated. In such a situation, even if many more mutated models are generated by Syntactic Mutation Operators than by Semantic Mutation Operators, they either rarely violate a security property, or they generate an AAT that is also generated by a Semantic Mutation Operator.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
WebGoat											
login(P1,P2)	X, S	S									
viewProf(P3)			X, S								
editProf(P4,P5)				X, S	X, S						
	Parameters used for XSS: 80% (100%); Parameters used for SQL: 100%										
Wackopicko											
register(P1,P2,P3,P4,P5)	X, S	X, S	X, S	X, S	X, S						
upload(P6,P7,P8,P9,P10)						X, S	X, S	X, S	X, S	S	
preview(P11)											S
	Parameters used for XSS: 90% (100%); Parameters used for SQL: 100%										
Bank											
register(P1,P2,P3,P4,P5)	X, S	X, S	X, S	X, S	S						
loginAdmin(P6,P7)						S	S				
activateClient(P8)								S			
loginClient(P9,P10)									S	S	
	Parameters used for XSS: 40% (100%); Parameters used for SQL: 100%										

Table 5.1.: Parameter Used for Different Attacks (Syntax: X=XSS, S=SQL)

5.2. Comparison of Higher-Order Semantic Mutation Operator and Higher-Order Syntactic Mutation Operator

Table 4.10 shows that only a subset of the AATs generated by the Semantic Mutation Operators are also generated by the higher-order Syntactic Mutation Operators (52 and 60 in the last row). Since this is counter intuitive, we performed the following investigation. We identified those 8 AATs that Semantic Mutation Operators generate but are missed by the Syntactic Mutation Operators. In the next step, we collected the corresponding mutated models, from which these 8 AATs are generated. Afterwards, we collected the mutated models produced by the Syntactic Mutation Operators that contain the same corresponding mutations. These are the models where the Syntactic Mutation Operators mutated the same mutation candidates like the Semantic Mutation Operators. Finally, we analyzed the eight AATs generated from those models. It turns out that all those eight AATs suffer from the same issue, that we exemplary explain as follows.

Listing 5.2 shows the kind of AATs that we also expected from the Syntactic Mutation Operators. In a nutshell, the attacker injects malicious XSS data using the filename parameter of the `upload` message. Then the user-provided data is stored in the database. Using the `recent` message, the malicious data is read from the database and sent back to the client. In Section 2.3.3 we discussed a way to model such values that are retrieved by a message handler and stored in a database. Such values can be modeled with random nonces. Listing 5.3 shows the AAT of the mutated model where the higher-order Syntactic Mutation Operators mutated the same parts in the model as the higher-order Semantic Mutation Operators. It is the statement that sanitizes the filename stored in the database. As one can observe, both AATs consist of the same sequence of messages that are exchanged between the browser and the web application. Nevertheless it turns out that the information available to the Syntactic Mutation Operator is not sufficient to determine the parameter where the injection has to take place. The rationale is the following:

In the AAT returned by the model checker, the unique value `instance_filename_t (Filename_upload(162), n140(RandomValue2Client))` appears as part of the `upload` message. The value in the database is represented as `instance_filename_t( Filename_upload(162), n158(Nounce_toStore)).` Since the Syntactic Mutation Operator does not know the semantics that the sanitization of the database value is an inherited property of the sanitization of the filename part of the `upload` message, the injection parameter cannot correctly be determined. Therefore, the AAT generated by the Syntactic Mutation Operators does not contain the `INJECT` and `VERIFY` keywords (Listing 5.3). In such a situation, one could apply the heuristics that there is only one filename involved in the AAT to determine the injection parameter. At the current state of the work, such a heuristic is not implemented (yet). In contrast, the semantic annotation `Implicit` (see Listing 5.1) establishes the link between the filename part of the `upload` message (Filename_uploadServer) and the filename stored in the database (Filename_toStore). This information can be exploited by the Semantic Mutation Operators while determine the parameter to be used for the injection. Therefore, the result is the AAT in Listing 5.2. Although a syntactical comparison of the AATs concludes that the Semantic Mutation Operator generates an AAT that is not generated by the Syntactic Mutation Operator, it is actually the automatic identification process of the vulnerable parameters that fails because the Syntactic Mutation Operators lack important semantic information.

In sum, the AAT generated by the Syntactic Mutation Operators requires more effort during the instantiation of the test cases. Either additional automatic analysis techniques have to be used during the instantiation, or an over approximation is needed to generated test cases that use every parameter of the AAT for the injection.

Listing 5.1: Extract of the Wackopicko Model:
Sanitization of the Filename Stored in the Database

```
%@Semantics[ HTML, Sanitize, Implicit(Filename_uploadServer) ]
sanitize_XSS( Filename_toStore ) ;
```

Listing 5.2: Missed AAT by Higher-Order Syntactic Mutation Operators (wS=webServer; cl=client)

```
<cl> *->* wS : register( cl,username_client,firstname_client,
                         lastname_client,password_client )
wS *->* <cl> : welcome( username_client )
<cl> *->* wS : similarName
wS *->* <cl> : similarName_response(sanitize_SQL_username_t(username_client))
<cl> *->* wS : upload( Username_upload( 162 ),Tag_upload( 162 ), INJECT(HTML),
                       Title_upload( 162 ), Price_upload( 162 ) )
wS *->* <cl> : askForComment( Username_upload( 162 ),Title_upload( 162 ) )
<cl> *->* wS : preview( Comment_upload( 162 ) )
wS *->* <cl> : show_preview( Username_upload( 162 ),Title_upload( 162 ),
                             Comment_upload( 162 ) )
<cl> *->* wS : create
<cl> *->* wS : recent
wS *->* <cl> : VERIFY( cl )
```

Listing 5.3: Corresponding AAT by Higher-Order Syntactic Mutation Operators (wS=webServer; cl=client)

```
<cl> *->* wS : register( cl,username_client,firstname_client,lastname_client,
                         password_client )
wS *->* <cl> : welcome( username_client )
<cl> *->* wS : similarName
wS *->* <cl> : similarName_response(sanitize_SQL_username_t(username_client))
<cl> *->* wS : upload(Username_upload(162),Tag_upload(162),Filename_upload(162),
                      Title_upload( 162 ),Price_upload( 162 ) )
wS *->* <cl> : askForComment( Username_upload( 162 ),Title_upload( 162 ) )
<cl> *->* wS : preview( Comment_upload( 162 ) )
wS *->* <cl> : show_preview( Username_upload( 162 ),Title_upload( 162 ),
                             Comment_upload( 162 ) )
<cl> *->* wS : create
<cl> *->* wS : recent
wS *->* <cl> : recent_response( Username_upload( 162 ),Tag_upload( 162 ),
                                Filename_upload( 162 ),Title_upload( 162 ),
                                Comment_upload( 162 ) )
```

5.3. Comparison of First-order Syntactic Mutation Operator to First-order Semantic Mutation Operator

In this section we discuss results based on the comparison of first-order Syntactic Mutation Operators and first-order Semantic Mutation Operators applied to the Wackopicko formal specification.

The first-order Syntactic Mutation Operators generate four AATs that are not generated by Semantic Mutation Operators (line 1 in Table 4.10). These four AATs are generated by mutated models where syntactic mutations have been performed at the client side (browser). The formal specification of the server side of the web application still correctly sanitizes all user-provided input values. The applied mutation by the Syntactic Mutation Operators leads to the situation that the value received by the client side browser is not the same value anymore than used in the security property. The received value is not stored in the variable anymore, that is used by the security property. Therefore, these four AATs do not reflect real vulnerabilities.

5.4. Comparison of First-Order Semantic Mutation Operator to Higher-Order Semantic Mutation Operator

In this section we discuss results based on the comparison of first-order Semantic Mutation Operator and higher-order Semantic Mutation Operator. In Section 4.5 we showed that all generated AATs by first-order Semantic Mutation Operator are also generated by higher-order Semantic Mutation Operator. Considering all AATs generated by the higher-order Semantic Mutation Operators, we observe that first-order Semantic Mutation Operators miss all those AATs where more than one input is exploited. Additionally, also some AATs where only one input is exploited are not covered by the first-order Semantic Mutation Operator. Therefore, we provide the rationales for those AATs (AAT 2, 3, 14, 19, 27, and 28 in Table 4.7; AAT 11 in Table 4.6).

AAT 3 in Table 4.7 and AAT 11 in Table 4.6. First-order Semantic Mutation Operators do not generate the AATs of the multi-step stored XSS and multi-step stored SQL injection since they require more than one mutation in the specific formalization. These two vulnerabilities and their corresponding attacks are considered as difficult vulnerabilities since automatic vulnerability scanners do usually not find them. In contrast, SPaCiTE *does* generate and find these vulnerabilities thanks to higher-order mutation operators.

AAT 2 in Table 4.7. At first sight, it is peculiar that AAT 2 in Table 4.7 is not generated by the first-order Semantic Mutation Operators. The AAT consists of one malicious parameter only. Nevertheless AAT 2 represents a multi-step attack, where sanitization must be vulnerable in the `register` message handler and the `recent` message handler as well. Therefore, a first-order mutation operator is not sufficient to generate this AAT.

AAT 3, 14, 19, 27, and 28 in Table 4.7. The same argumentation holds for AAT 3, 14, 19, 27, and 28. There are two different message handler involved and therefore, sanitization must be invalidated twice to generate these AATs. Therefore, a first-order mutation operator is not sufficient.

All these examples show that higher-order mutation operators are essential to find (multi-step) stored XSS and SQL vulnerabilities. In general, the more complex a vulnerability is, the more is the tendency that higher-order mutation operators are required to find the vulnerability. This is also true for attacks that only require one malicious parameter. The above discussion shows that an AAT consisting of one malicious parameter only is no guarantee that first-order mutation operators are sufficient.

5.5. Operationalizing AATs from Syntactic Mutation Operators

In this section we illustrate on a technical level the effort required to operationalize AATs generated by Syntactic Mutation Operators. We discuss what kind of AATs are generated and possible strategies to operationalize them. This section shows that AATs generated from Syntactic Mutation Operators may lack crucial information needed for the operationalization. Applied strategies to operationalize them often lead to an over-approximation which influences the efficiency. In this section we discuss the different strategies and we will discuss the implications of them in the following Section 5.6.

Applying Syntactic Mutation Operators generates three different types of AATs. The first type are AATs that contain the two keywords INJECT and VERIFY together with information which type of malicious code is required for the injection (see Listing 5.4). Such an AAT is possible for a Syntactic Mutation Operators if the violated security property is named accordingly (XSS, SQL), although such a naming convention is equivalent to a semantic annotation of the specified security property. In such a situation the kind of malicious input can be learnt from the name of the violated property. They have the same form as AATs generated by Semantic Mutation Operators. The second type of AATs still contain the two keywords INJECT and VERIFY but they lack information about the type of malicious code (see Listing 5.6). The third type of AATs are traces without indications where to inject malicious code (see Listing 5.5). While the first type of AATs can be operationalized exactly in the same way as AATs generated by Semantic Mutation Operators, the other two types require more effort. To operationalize such AATs, the Security Analyst can apply different strategies:

Listing 5.4: Example AAT

```
<client> *->>* webServer : register(client,username_client,INJECT(SQL),
                                     lastname_client,password_client
                          )
webServer *->>* <client> : welcome( username_client )
<client> *->>* webServer : similarName
webServer *->>* <client> : VERIFY( client )
```

Listing 5.5: Example AAT

```
<client> *->>* webServer : register(client,username_client,firstname_client,
                                     lastname_client,password_client
                          )
webServer *->>* <client> : welcome( username_client )
<client> *->>* webServer : similarName
webServer *->>* <client> : similarName_response(
                               sanitize_SQL_username_t(username_client)
                          )
```

Listing 5.6: Example AAT

```
<tom> *->* webServer : login( username( tom ),INJECT( Unknown ) )
webServer *->* <tom> : VERIFY( tom )
```

Strategy 1: The Security Analyst operationalizes only AATs that contain information which parameter is used for which kind of attack. In particular, he does not instantiate those AATs where no indication is provided (see Listing 5.5). Applying strategy 1 has the consequence that mutated models are generated where the model checker finds counter examples, but due to the missing semantics of the mutation it is unclear how to operationalize them. Syntactic mutations that lead to such AATs can e.g., be:

- A mutation is performed at the client side.
- A mutation may allow the use of a value (e.g., a nonce or a dummy value) that violates a security property but the same value is not used for the communication between the client and the web application.
- Syntactic Mutation Operators may lead to so called dummy values used in the communication between the browser and the web application. When a dummy value should be sent to the web application, it is unclear what a concrete value should be. Similar, if a dummy value is sent back to the client, it is unclear how to verify such a dummy value.
- A mutation of an internal value (e.g., stored in a database) that cannot be linked to the corresponding value appearing in the communication because of the missing semantic annotation.

Since AATs like the one in Listing 5.5 are not operationalized when applying strategy 1, there are models so that Syntactic Mutation Operators are less effective than Semantic Mutation Operators. As an example, Semantic Mutation Operators applied on the Wackopicko formal specification generate the AAT given in Listing 5.4. It represents the stored SQL attack. The most similar AAT generated by the Syntactic Mutation Operators is the one given in Listing 5.5. It contains no injection information and therefore, the stored SQL injection test case is not generated using Syntactic Mutation Operators and applying strategy 1. Since a mutated model was generated that leads to an AAT it is unfair to ignore it just because not enough injection information is available. Therefore, one could treat each parameter as potentially vulnerable, as explained in the following strategy.

Strategy 2: The Security Analyst instantiates whose AATs where no indication is provided 'where' to inject 'what' with a brute force strategy. Since he knows that Semantic Mutation Operators are dedicated to XSS and SQL injections, a brute force strategy instantiates the AAT given in Listing 5.5 as follows:

- Inject XSS and SQL malicious code respectively using the client reference of the register message (first parameter).
- Inject XSS and SQL malicious code respectively using the username, first name, last name and password of the register message.

Considering AATs where the malicious parameter is identified but the type of injection is unknown (see Listing 5.6), a brute force strategy would inject XSS and SQL malicious code only using the identified parameter.

This brute force strategy leads to the situation that Syntactic Mutation Operators generate a super set of executable test cases generated by Semantic Mutation Operators. It contains all the test cases that the Semantic Mutation Operators generate but Syntactic Mutation Operators generate additional test cases that have no vulnerability motivation from the specification. To illustrate that, we apply the brute force strategy to the AAT given in Listing 5.5 and take the AAT that injects XSS malicious code using the password parameter of the register message. This AAT is not generated by the Semantic Mutation Operators since at model level, the password is never sent back to the client and displayed in the browser. Therefore, also no XSS related security property can be violated and it is very likely that also a penetration tester would not generate and execute such a test case.

5.6. Effectiveness and Efficiency

The chosen strategy for operationalizing AATs generated from Syntactic Mutation Operator influences the efficiency of the Semantic Mutation Operators compared to Syntactic Mutation Operators. In this section we apply strategy 2 described in Section 5.5 to the Wackopicko formal specification. We illustrate the consequences of applying Syntactic and Semantic Mutation Operators and speculate about the required effort. An accurate measurement is currently not possible and considered as future work. Furthermore, our evaluation is based on three case studies. They provide important insights and allow the drawing of tendencies. Nevertheless we are aware that a complete generalization is not possible from our evaluation data.

Effectiveness of SPaCiTE. In contrast to other tools like black-box vulnerability scanners, SPaCiTE generates test cases for non-trivial XSS and SQL vulnerabilities that require multi-step attacks. Executing such test cases on a real implementation can find the vulnerabilities also in the System Under Validation (SUV), if present. Such vulnerabilities are often not found by automatic black-box scanners. Concretely, SPaCiTE finds the stored XSS vulnerability exploitable with a multi-step attack in WebGoat and Wackopicko, as well as the stored SQL vulnerability exploitable with a multi-step attack in Wackopicko. ZAP did not find the same vulnerability in WebGoat, and the black-box scanners Acunetix, AppScan, Burp, Grendel-Scan, Hailstorm, Milescan, N-Stalker, NTOSpider, Paros, W3af, and Webinspect did not find the vulnerabilities in Wackopicko.

Effort for Syntactic and Semantic Mutation Operators. While effectiveness is usually a binary decision, efficiency is non-trivial. An absolute value is not possible due to the large number of dependencies and the complexity of measuring them. Nevertheless, we summarize the crucial components (annotation, model-checking, operationalization, and execution) and elaborate on their required effort. For the efficiency evaluation, we need approaches that find non-trivial XSS and SQL vulnerabilities but require different efforts. Therefore, we compare Semantic Mutation Operators with Syntactic Mutation Operators, both operating on abstract models.

Modeling. Modeling the web application is an important, not negligible, and crucial aspect in terms of efficiency. If models are not available, the Security Analyst must have ASLan++ knowledge to come up with a formal behavioral model. Depending on the skills of the Security Analyst, the modeling aspect is challenging and might require several hours. Nevertheless, such a formal specification is needed both for Syntactic and Semantic Mutation Operators.

Annotations. Syntactic and Semantic Mutation Operators differ in terms of annotation effort. While Semantic Mutation Operators require manual annotations, Syntactic Mutation Operators do not. For the Wackopicko specification, the Security Analyst needs to provide 38 annotations, the WebGoat specification contains 16 annotations, and the Bank specification contains 37 annotations. They all have a predefined syntax and do not require knowledge about vulnerabilities. Each annotation requires three inputs from the Security Analyst:

1. The Security Analyst needs to identify those parts of the formal specification that represent sanitization functionality.
2. For each of them, the Security Analyst needs to provide the technology the annotated part sanitizes against, since data can be sanitized against SQL, HTML, Javascript, etc.
3. The third input specifies whether input or internal data is sanitized, and whether the sanitization occurs with an explicit statement or whether it is implicitly given by the semantics of another sanitization part.

It is important to stress that such annotations can be provided by a Security Analyst that has a rough idea about the implementation. Knowing all source code level details is not necessary. The required effort obviously depends on the skills of the Security Analyst. From our expertise, we believe that this task can be completed within very few hours, if not one single hour.

Model-checking Mutated Models. For the Wackopicko web application, applying Semantic Mutation Operators to the annotated formal specification leads to 348 mutated formal specification. Model checking those specifications requires approximately 33 hours of computation power (see Figure 5.1) using the configuration specified in Section 4.4, eight parallel threads and a timeout of 2400 seconds.

In contrast to the Semantic Mutation Operators, Syntactic Mutation Operators do not require manual annotations but can be fully automatically applied to formal ASLan++ specifications. Applying the Semantic Mutation Operators to the Wackopicko specification generates in total 4026 mutated models. Model checking all of them requires approximately 320 hours of computational power (see Figure 5.1). For comparison, higher-order Syntactic Mutation Operators applied to the WebGoat specification generates 3820 mutated models and the Bank specification more than 12’000.

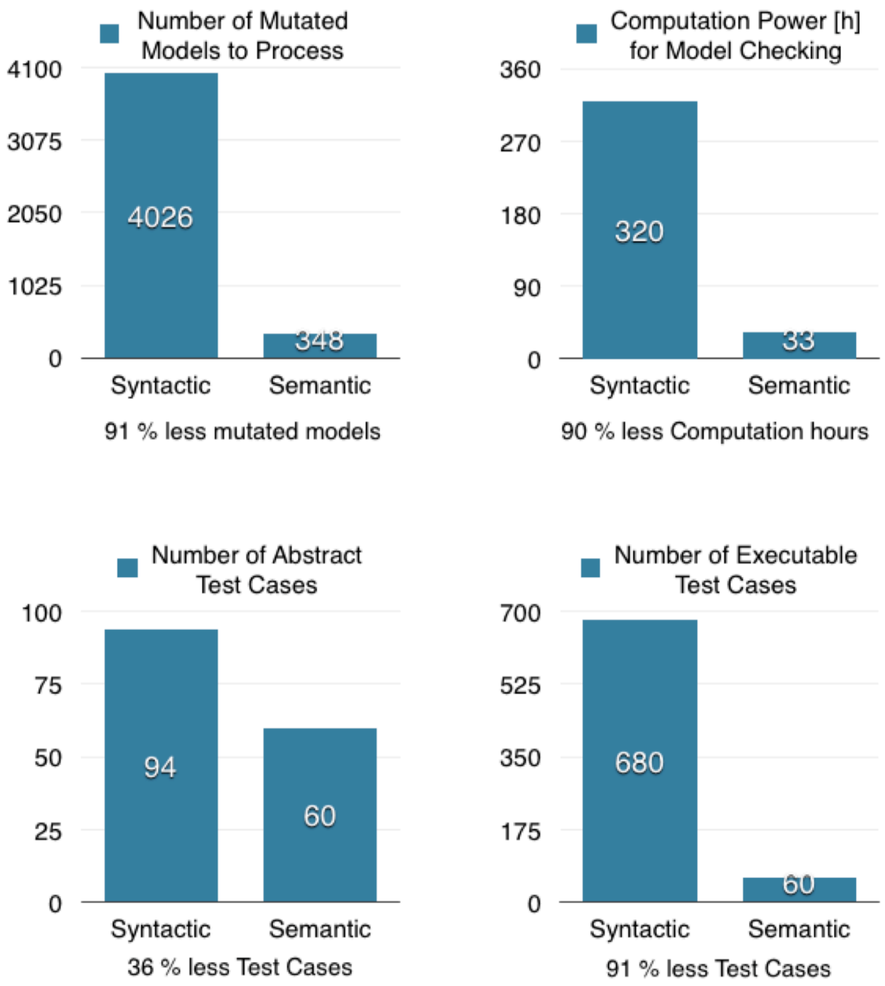


Figure 5.1.: Impact of Manual Annotation

	Modeling	**Anno-tations**	**Mutated models**	**Model checking [h]**	**AATs**	**Executable test cases**
Syntactic	✓		4026	320	94	680
Semantic	✓	✓	348	33	60	60

Table 5.2.: Impact of Manual Annotation

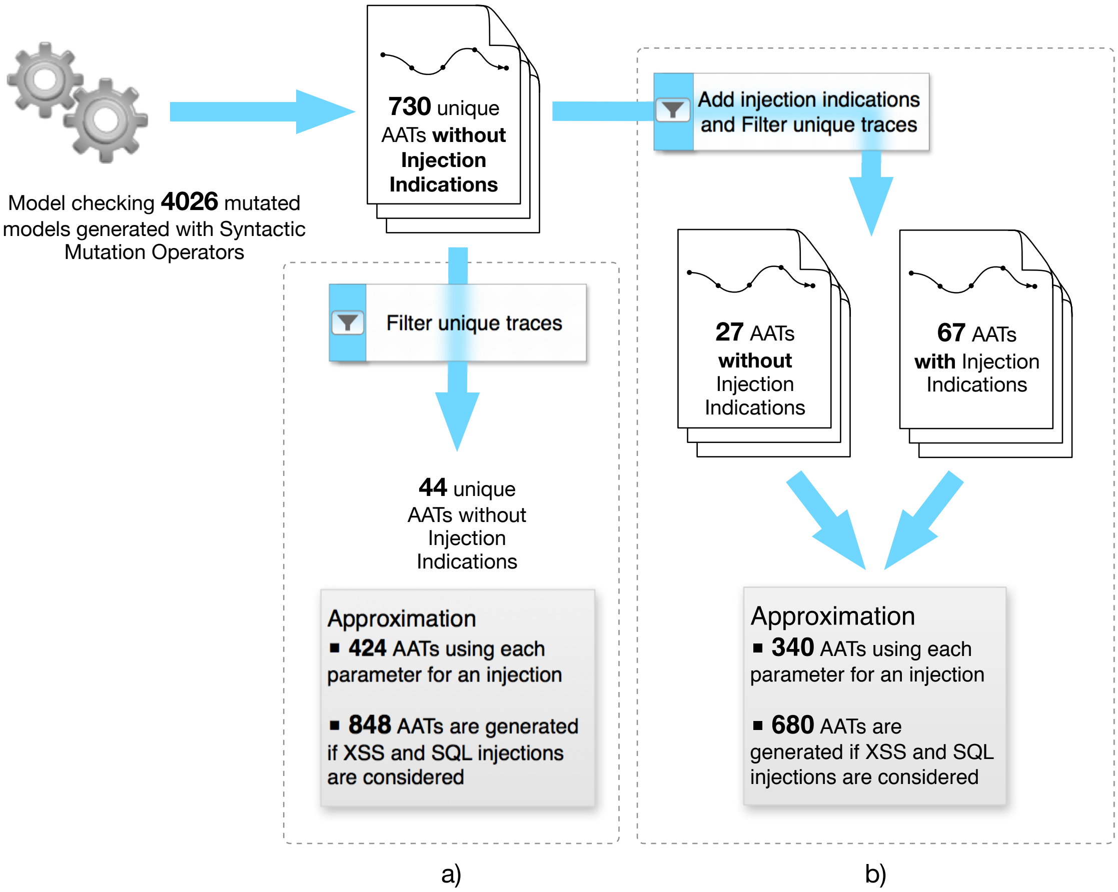

Model checking 4026 mutated models generated with Syntactic Mutation Operators
730 unique AATs without Injection Indications
Filter unique traces
44 unique AATs without Injection Indications
Approximation
▪ 424 AATs using each parameter for an injection
▪ 848 AATs are generated if XSS and SQL injections are considered
a)
Add injection indications and Filter unique traces
27 AATs without Injection Indications
67 AATs with Injection Indications
Approximation
▪ 340 AATs using each parameter for an injection
▪ 680 AATs are generated if XSS and SQL injections are considered
b)

Figure 5.2.: Operationalization of AATs Generated By Syntactic Mutation Operators

Operationalization. The operationalization of AATs generated both from Syntactic and Semantic Mutation Operators is possible with the same amount of manual activities, if generated from the same model. The required activity consists of providing a single Web Application Abstract Language (WAAL) mapping that can be reused for all AATs generated from the same formal specification.

In terms of the number of generated executable test cases, AATs generated from Semantic Mutation Operators differ from AATs generated from Syntactic Mutation Operators. In Section 5.5, we already discussed different strategies for the operationalization of AATs generated from Syntactic Mutation Operators applied to a detailed example. Please remember that AATs generated from Semantic Mutation Operators always contain injection information (using the mutation operators described in Section 2.6.1). For AATs generated from Syntactic Mutation Operators, this is not always the case (see Section 5.5). As a reminder, AATs as a direct output of a model checker do not contain the special keywords *inject* and *verify*. These keywords are added by post-processing the reported AATs. Depending on the used mutation operator, identifying the correct message parameters is not always possible. Figure 5.2 provides an approximation overview of the number of generated executable test cases for the Wackopicko formal specification using Syntactic Mutation Operators. Figure 5.2#a shows the situation where the AATs as a direct output of the model checker are operationalized without trying to determine the malicious input parameters at all (Strategy 1). Figure 5.2#b shows the situation where first the malicious parameters are determined before the AATs are operationalized.

Strategy 1. The model checker processes approximately 4000 mutated models and generates 730 AATs, where 44 traces are unique. The traces are online available [36]. In sum, they consist of 44 `register` messages with 5 parameters each, 34 `upload` messages with 5 parameters each, and 34 `preview` messages with 1 parameter each. Without further analyzing where to inject which kind of malicious code, every parameter is considered as a potential injection point. If split injections are neglected and two different types of malicious injections are considered (XSS and SQL) approximately 850 executable test cases 2*(44 * 5 + 34 * 5 + 34) are generated.

Strategy 2. While Strategy 1 is a blind brute force heuristic, it is recommended to try to determine the injection parameters. This leads to a total of 94 unique AATs. The traces are online available [36]. For 67 of them, the injection parameter can be determined. Generating two test cases for each of the 67 AATs (one for XSS and one for SQL), and applying Strategy 1 for the remaining 27 AATs reduces the number of executable test cases to approximately 680 test cases (see Figure 5.1), since the 27 AATs consist of 27 `register` messages with 5 parameters each, 23 `upload` messages with 5 parameters each, and 23 `preview` messages with 1 parameter each. This illustrates that as soon as we have AATs without injection information, a brute-force strategy very soon generates a huge test suite.

Execution. After the operationalization, the execution is a fully automatic process for our use cases. The test execution is handled by the TestNG framework, test data is automatically provided by TestNG data providers, test distribution among worker nodes is automatically handled by the Selenium Grid framework, and reseting the web application to a predefined initial state between the execution of two test cases is automatically possible for the VMWare vSphere Virtualization Environment [33].

Although the execution is fully automatic in our context, it is not in general. There are multiple reasons why manual help of the Security Analyst might be required.

- The discussed mutation operators in Section 2.6.1 allow the fully automatic execution of test cases at least to the point in the trace, where malicious data has to be injected. The rationale is that the trace till the injection point corresponds to a trace conforming the correct behavioral specification. Whether the remaining trace after the injection is still executable depends on how the implemented web application reacts upon the injection (error and exception handling, etc.).

- There exists mutation operators that lead to AATs where manual help of the Security Analyst is needed. E.g., a mutation operator might inject a vulnerability that allows reordering exchanged messages between the browser and the web application. As part of the SPaCIoS EU project, we have already implemented such mutation operators, but they still require future research. Therefore, they are not discussed as part of this Ph.D thesis.

As a summary, by manually providing approximately 40 semantic annotations, the number of mutated models can be massively reduced from around 4000 to 350, computation hours from 320 to 33, and the number of executable test cases from 680 to 60 (see Figure 5.1). Nevertheless these numbers have to be put in a bigger context. In terms of using model-based security test case generation for web applications, we see advantages and disadvantages both at the same time. While sophisticated attacks often consist of multiple steps, using verification techniques based on behavioral models definitely can be used for automatically generating such test traces. E.g., it is the strength of model checkers to find traces in a model that violates a defined security property. At the same time, attacks are not only sophisticated because they consist of multiple steps, but also because the input data has to follow a syntax and is encoded in different ways. In this respect, verification techniques in combination with behavioral models are too abstract as that encoding information could be part of such specifications. We believe that considering issues at the level of encoding too fast lead to the state explosion problem that model checker often suffer. We therefore recommend to put future research effort into combining model-based verification techniques with source code analysis techniques like e.g., symbolic execution. This would allow the combination of highlevel traces of a web application with low level syntax and encoding information for concrete, application dependent malicious inputs.

5.7. Multiple Malicious Input Parameters

In Section 4.4.1.1 we have seen that the `XSSRemoveSanitization` Semantic Mutation Operator generates three additional AATs that are not generated by the `XSSRemoveSanitizationUpToLimit[1][2]` and `XSSRemoveSanitization[1][1]` mutation operators. Therefore, we closely analyzed these three additional AATs. Unfortunately, not all of the additional AATs generated by the `XSSRemoveSanitization` are useful. To filter out non-useful AATs a double check on the AATs is needed. When multiple input parameters are exploited, one has to check if indeed all reported parameters contribute to the violation of the involved security property. Depending on how the security property is formulated, exploiting one parameter might already be sufficient to violate the security property. It might be the case that the security property (formula) covers both input parameters of a message in the same formula, but one parameter is sufficient to violate the formula. Unfortunately, the used model checker for this Ph.D thesis (Cl-Atse) reports all parameters of the violated formula, independent if they contribute to the violation or not. Therefore, a double check either needs an analysis of the violated property (formula), or an adaptation of the security property to make sure that the AAT is reported only if indeed all vulnerability affected parameters contribute to the property violation.

An example of a reported AAT that would have filtered out is AAT 8 in Table 4.3[1]. This AAT was generated from a mutated model where three mutation candidates were mutated — the `ID` and the `data` parameter of the `edit` message, and the returned `ID` parameter part of the `login` response. For this AAT the `data` parameter is not required to be mutated since the `data` parameter is not contained as part of the `login` response. Therefore, this AAT would have been filtered out during a second check. We want to stress that the double check depends on the violated security property. E.g., let's consider a security property in DNF, expressing an attack state. Furthermore, let's assume that each conjunctive clause of the DNF consists of one variable only. Therefore, one malicious value is sufficient to violate the security property. Nevertheless, the used model checker reports all values occurring in the violated security property. The value-tracking facts (see Section 2.6.4) restricts this set of values to those that are affected by the vulnerability injection. While this approach is sufficient for reflected and even multi-step stored attacks using only one malicious parameter, it is not for multi-value based attacks.

We described the issue with an example of two malicious parameters, but it is a general problem with any number ≥ 2 of injected vulnerabilities. In rare cases, the issue can be tackled in the AAT generation phase by setting the minimum and maximum number of mutations to the required mutations to discover the vulnerability while ignoring any reported AAT that only exploits one parameter. Another strategy to address this problem is a post-analysis of the violated security property to find the minimal set of values that are sufficient to violate the security property. Such a post-analysis is not implemented in the current version of SPaCiTE yet and is therefore considered as future work.

5.8. Syntactically Different But Semantically Equivalent AATs

If multiple agents / values are available, a vulnerability can be exploited using different combinations of such values. E.g., if multiple honest agents are defined in the formal specification, an AAT like AAT 6 in Table 4.3 is a counter example for each honest agent. If in addition multiple dishonest agents exist, an AAT like AAT 7 in Table 4.3 is a counter example for multiple combinations of a dishonest and an honest agent. Unfortunately the current version of the model checker does only report *one* security property violating trace. If multiple such combinations should be reported, either the model checker needs to be modified or the security properties have to be adapted. In contrast, the examples that we have considered during this Ph.D thesis showed that if an XSS or SQL vulnerability is present for an agent x with role y, it is also available for any other agent with the same role y. When it comes to a comprehensive comparison of AATs, two AATs might differ due to using different agents but the agents have the same role. Although such two AATs are syntactically different, they might be semantically equal with respect to XSS and SQL vulnerabilities. In sum, this observations allows one e.g., to group test cases together, that are equivalent modulo the user who performs the actions. It further highlights future work e.g., in terms of finding the minimal set of test cases to identify a vulnerability.

[1] A similar argumentation is applicable to AAT 10 in Table 4.3

5.9. Penetration Tester vs. SPaCiTE User

As we have discussed in Chapter 1, a popular way of testing web applications for security issues is penetration testing. This kind of testing requires that the tester is in the same mind set as an attacker. In addition the tester must have expertise in terms of exploits and how the system can be attacked. In contrast, the Security Analyst considered for this Ph.D thesis is required to be in the implementation mindset. We have seen that one of Security Analyst's task is to provide security annotations to a formal specification. These annotations provide semantics in terms of functionality and used technology during the implementation of the web application. The expertise to provide and execute attacks on the SUV is moved from the Security Analyst to the SPaCiTE tool. This implies that the SPaCiTE tool does not require the same level of expertise than a penetration tester must have. Together with the fact that SPaCiTE contributes with a modern IDE and the automation of e.g., applying mutation operators, and operationalization of AATs, the approach tends to be usable for a broader audience then pure penetration testers.

5.10. Structural Coverage Criteria

We have argued in Chapter 1 that coverage criteria are often proposed in literature for generating or validating test suites. We have seen that SPaCiTE injects vulnerabilities into the formal specification of a web application and uses verification tools to generate a test suite that covers these vulnerabilities. In particular, SPaCiTE does not generate test cases dedicated to a vulnerability v for parts of the model that do not suffer from the vulnerability v. This is an advantage of our approach since not every element of the specification is dedicated to security. Some artifacts are dedicated to the overall functionality of the web applications, other artifacts are introduced due to the fact that model checkers are used. Coverage criteria make sense if one has a model that only talks about explicit security aspect. For example the purpose of a firewall policy is authorization. Every rule was written because every rule captures one aspect of authorization. Because every rule contributes to security, achieving high coverage with a test suite for such models make sense. On the other hand if one considers a behavioral model that is a mixture of functionality and security mechanism, not every part of the model is of the same importance when security testing is performed. As an example, if one wants to test for reflected XSS attacks, all test cases that do not cover at least one user input and one response that contains the user provided input, do not make sense. The coverage criteria proposed in the literature do not focus on these aspects and are therefore not appropriate for security testing of models that mix behavioral and security aspects.

5.11. Vulnerabilities that Require Authentication and Sophisticated Front End Technologies

One practical issue with finding vulnerabilities especially for automatic tools is 'authentication' and sophisticated client-side technologies like Flash and JavaScript. If authentication blocks a tool to reach the state of an application where a vulnerability can be exploited,

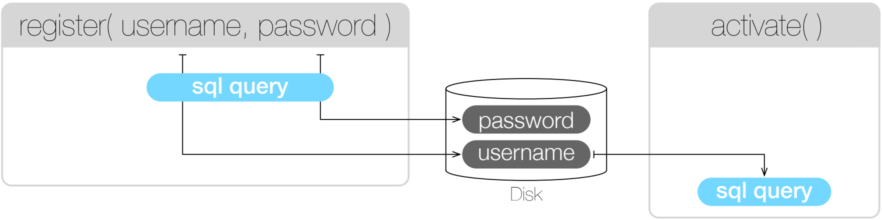


Figure 5.3.: Conceptual Register and ActiveClient Messages

the vulnerability will not be found. Also, if crucial webpage elements require the availability of Flash or JavaScript, a security tool needs to properly handle such technologies. While tools that directly operate on the protocol level, often suffer from these issues, SPaCiTE does properly handle them because the SPaCiTE-conform input models include behavioral aspects and covers them via the browser level. Therefore, authentication and sophisticated front end technologies do not raise issues for SPaCiTE.

5.12. Vulnerability Injection vs. Attack Injection

In this subsection we discuss an important difference and the corresponding consequences between vulnerability injection and attack injection. The bank application offers a method for registering a new account which is later activated by the administrator using the `activateClient` message. The conceptual information flow of that example is shown in Figure 5.3.

For this specific situation, applying the mutation operators proposed in Section 2.6.1 generates AATs that inject a vulnerable payload using the `username` parameter. The same parameter is then later used for the `activateClient` message. This kind of attack can be generated by solely injecting *vulnerabilities*. In particular, the mutation operator does not describe how this vulnerability is exploited. Due to the vulnerability the corresponding value is not sanitized against SQL and the attack is generated because that abstract value can be later used in an SQL query (`activateClient` message handler). As a reminder, since the modeling level is an abstraction of the real implementation level, the abstraction level is too high for the model checker to generate malicious payload. Due to that, the model checker is not able to determine the effect of a malicious value to other artifacts of the model (e.g., other variables) that then could violate a security property. This leads to the fact that AATs that require attack injections are missing. E.g., the following SQL vulnerability and attack behavior is not generated by SPaCiTE with vulnerability injections.

1. Inject a malicious SQL command using the password parameter of the `register` message in Figure 5.3.

2. The attack semantics of the used injection turns the stored `username` parameter malicious as well. E.g., this can be achieved if the malicious `password` parameter appends an additional INSERT command that overrides the username parameter in the database.

3. The overwritten `username` parameter in the database is used for the `activate Client` functionality.

To let SPaCiTE generate such kind of AATs, the following requirement needs to be fulfilled: The Semantic Mutation Operators not only need to inject vulnerabilities, but also attack effects. In the above example, the mutation operator needs to inject the effect that a malicious `password` parameter affects the sanitization property of another parameter like the username. Since the mutation operators as introduced in Section 2.6.1 reflect vulnerabilities, the above discussed kind of attacks are not generated by SPaCiTE so far.

5.13. Destructiveness of Attacks

A vulnerability can be exploited by an attack, that destroys data. In models that we have developed, the act of destroying the data violated defined security properties. A model checker then returns a counter example that ends in the state where the data was deleted. For the test generation, this test case needs to be prepended to a test sequence that indeed verifies that the data was deleted. The current version of SPaCiTE does not address these kind of attacks since it requires another iteration where the model checker has to generate a second trace. We consider this as future research that needs to be addressed.

5.14. Scalability

Unfortunately, the proposed approach as a whole does not scale well. The main component that prevents scalability is the verification technique (model checker). During modeling, the number of message parameters, the number of messages that can be exchanged between different agents, and the number of declared data values influence the time for model checking. In contrast, the Test Execution Engine (TEE) makes use of the Selenium Grid architecture. It is designed for scalability and makes use of a central hub that manages a set of nodes that execute the test cases decentralized. In addition, the TestNG framework used for the executable test cases contributes with dedicated features for concurrent test execution using thread pools.

5.15. Discussion on Future Larger Scale Evaluation Set-up

[This subsection was already discussed and published in the SPaCIoS deliverable 5.5 [200]]

In this section we discuss possibilities for setting up such experiments in the future, and conjecture some hypothesis that could be refuted/confirmed as a result of carrying on the experiments.

When a security tool like SPaCiTE is developed, it has to be evaluated in terms of effectiveness and efficiency. We will discuss a couple of strategies to compare a structural approach (e.g., SPaCiTE) to a non-structural approach (e.g., pen testing). For the remainder of this section, we use the tool SPaCiTE as a representative of a structural testing approach, and penetration testing as a representative non-structural approach.

For all strategies, the evaluation is performed in a time frame of x weeks. A supervisor selects a vulnerable application where he knows the actual vulnerabilities. The knowledge where the vulnerabilities are is not shared with the groups that perform the security testing. After the groups have finished their testing activities, it is the supervisor who analyses the result according to effectiveness and efficiency.

Strategy 1

For strategy 1, we select two equally educated groups of computer scientists where the first group performs penetration testing on a given application, and the second group uses SPaCiTE in order to test the same web application. For both groups the vulnerabilities are not previously known to the group members. After both groups have tested the same web applications for x weeks, the discovered vulnerabilities by each group are compared. An interesting research question in this context is: Given a fixed time frame and a fixed cost, does a structural or non-structural approach perform better according to effectiveness and/or efficiency metrics? The corresponding hypothesis is: For a short time frame, a non-structural testing approach has a better cost-benefit ratio than a structural approach. For a longer time frame, the opposite is true.

Strategy 2

The following strategies are more focused on the interleaving of structural and non-structural testing approaches. The first group performs pure penetration testing. The second group starts with penetration testing as well, but then switches to SPaCiTE. The intuition behind this strategy is that low hanging fruits are more easily handled with penetration testing, and more complex issues are better handled by a structural approach. Instead of continuing with penetration testing for finding more difficult vulnerabilities, a structural approach is followed.

Strategy 3

Similar to strategy 2, strategy 3 compares pure penetration testing with a mix of structural and non-structural approach. Again, one group performs pure penetration testing. The second group first starts with SPaCiTE and then switches to penetration testing as well. An interesting research question is whether penetration tester report different vulnerabilities if they first develop a model and the model checker potentially reports more complex issues.

Strategy 4

The next strategy allows a structural approach like SPaCiTE to abstractly highlight security issues that are then further investigated with penetration testing. Both groups start with penetration testing and after a certain time, both groups switch to a structural approach. The difference between the two groups is that the first group sticks to the structural approach once it switched to the structural approach. The second group can use the structural approach to get some new inspirations that are then further investigated with penetration testing.

Strategy 5

Strategies 1 - 4 all compare a model-based structural approach with a non-structural, non-model-based approach. Strategy 5 focuses on the usefulness and the level of formalism of the model. For this strategy 3 groups are needed. The first group performs a non model-based approach like penetration testing. The second group uses a informal or semi-formal testing approach. An example could be an approach based on UML models. In contrast, the last group uses SPaCiTE and formal ASLan++ models. The idea is to compare the reported vulnerabilities depending on the different kinds or the lack of models.

Strategy 6

Finally strategy 6 focuses more on the testing period that follows after first vulnerabilities have been reported and fixed. One group applies penetration testing and reports found vulnerabilities. After they have been fixed by the software developers, the corrected code has to be retested again to verify, whether the reported vulnerability is fixed and whether no additional vulnerabilities were introduced. The other group applies SPaCiTE and tries to benefit from the structural and model-based approach during the retesting phase. Therefore, strategy 6 is more focused on the effort during the retesting phase, and not primarily during the first testing phase.

5.16. Conclusion

As an overall conclusion, we highlight the following contribution. In a model-based context, non-trivial security vulnerabilities for web applications can be found with mutation operators and fault injections. Using ASLan++ formal specifications and fault injections generate test cases that find non-trivial security vulnerabilities that other tools like black-box vulnerability scanners often do not find. For sophisticated and non-trivial vulnerabilities, higher-order mutation operators are needed.

In Section 1.4 we discussed the sub hypothesis that in a model-based context, using Semantic Mutation Operators is more efficient than using Syntactic Mutation Operators. In Section 4.1 we raised the research question: *Are vulnerability-based fault models at ASLan++ model level more efficient in generating 'security-interesting' test cases than syntax-based fault models at the same model level?* To answer this question, we have seen in Section 4.5 and chapter 5 that Syntactic and Semantic Mutation Operators are both based on the same General Mutation Operators. Since the Syntactic Mutation Operators only operate on the syntax and do not need semantic annotations, they allow mutations at more locations in the model than Semantic Mutation Operators and therefore, they generate a bigger set of mutated specifications. In the context of injecting faults that means that Syntactic Mutation Operators inject faults at locations in the model that are implemented with technologies that actually cannot suffer from the injected fault.

Syntactic Mutation Operators generate more mutated models but the ratio of models that generate an AAT decreases. While Syntactic Mutation Operators generate up to 4000 mutated models but only 94 AATs for the Wackopicko specification, Semantic Mutation Operators generate approximately 350 mutated models and 60 AATs. This is because a mutated model alone does not generate a test case, but only if a security property is violated. If all message parameters are covered by the defined security properties in the formal specifications, Syntactic Mutation Operators do not generate massively more AATs. This demonstrates that the defined security properties already filter out all those mutations that do not affect the security properties. The fact that nevertheless more AATs (up to twice as many (see Table 4.9)) are generated than by using Semantic Mutation Operators is many fold. One reason is that in the context of XSS and SQL vulnerabilities, syntactically different AATs might have the same semantics. Considering the WebGoat application as an example, it is usually not important which concrete user profile is used for exploiting an XSS vulnerability, because all profiles are handled by the same source code while editing and viewing. For finding a vulnerability, a trace with *one* such profile is sufficient. Since Syntactic Mutation Operators are applied to more locations in the specification, multiple AATs are generated using different data values but exploiting the same vulnerability. To provide a minimal and therefore efficient set of test cases, this means that AATs need to be compared manually to exclude syntactically different but semantically equivalent AATs.

The effectiveness and efficiency of Syntactic Mutation Operators, and therefore the answer for the hypothesis, also depends on the strategy that the Security Analyst applies. Using a brute force strategy for AATs generated by Syntactic Mutation Operators results in a test suite that is at least as effective as a test suite generated by Semantic Mutation Operators. Nevertheless, Semantic Mutation Operators profit from semantic annotations that make the operationalization aspect of AATs more efficient since all AATs contain information where to inject what type of malicious code. By manually providing approximately 40 semantic annotations, the number of mutated models can be massively reduced from around 4000 to 350, computation hours from 320 to 33, and the number of executable test cases from 680 to 60 (see Figure 5.1). The question remains whether such a brute force strategy for AATs generated by Syntactic Mutation Operators fulfills the cost-effectiveness criteria in the definition of a '*good*' test case since it generates test cases that a penetration tester would not generate and execute. More concretely, the question whether it is more efficient to invest into computation power and execution resources or into the manual task of annotating formal specifications, needs to be answered for each concrete efficiency metrics separately. In a concrete situation, the Security Analyst has to judge whether the manual effort of annotating formal specifications with semantic keywords is worth preventing the generation of non-executable test cases or test cases without a vulnerability-based rationale.

6. Related Work

In this section we discuss related work to put our work into context. We first start with a paper that highlights open issues in the area of testing web applications. Afterwards, we discuss papers that follow the same goal but are not necessarily model-based in Section 6.1. Finally, we describe papers that are relevant due to the classification discussed in Appendix A.

An important motivational work was published by Doupé et al. [92]. It analyses existing black-box vulnerability scanners for web applications. The purpose of that analysis is an evaluation and an understanding how well automatic web vulnerability scanners perform in identifying security related vulnerabilities. Doupé et al. evaluated eleven different scanners on a realistic web application for photo-sharing and purchasing, called WackoPicko. All these scanners are characteristic for the ease of use and the high level of automation. Their task is to discover the web application and enumerate all reachable pages, including the necessary input vectors. The paper contributes with an extensive evaluation of such black-box scanners, an identification of challenges for scanners, the development of the WackoPicko web application, and a discussion why the previously identified scanners fail to detect certain vulnerabilities. The paper shows that state-of-the-art web application scanners fail to detect a significant number of vulnerabilities, as also reported in [1, 172, 202, 203, 226]. Many common and important vulnerabilities are not identified by the considered scanners. It turned out that the crawling task is very critical and challenging for the effectiveness of a scanner. The paper concludes with reasons for failing in terms of effectiveness:

- Scanner must navigate HTML frames, in particular, it should support well-known and widely used web technologies. This includes the handling of complex forms and rigorous input validation functions since they might prevent the scanner from following a particular link.

- Scanner must understand the state-based transactions to discover all possible states of a web application.

- Scanner must handle infinite number of pages. E.g., a web page that includes a calendar widget can easily lead to an infinite number of web pages (a different web page for each day).

- Scanner must be able to authenticate itself to access protected areas.

- Scanner must parse and understand client-side technologies like Javascript.

- Crawlers should not suffer from implementation errors. In particular, the HTML parser should be robust and sophisticated enough to identify a variety of different links on a web page.

Although we do not consider SPaCiTE as a security scanner, SPaCiTE addresses some of the above mentioned issues. The price is to be semi-automatic compared to the fully automatic black-box vulnerability scanners. E.g., SPaCiTE is able to navigate HTML frames and complex forms (including Javascript) with the help of the Selenium framework [26], the set of possible states is provided in form of a formal specification, and authentication is possible with the help of the Web Application Abstract Language (WAAL).

Our work is at the intersection between Model-based Testing (MBT) and penetration testing (pentesting). We rely on mutation operators to introduce implementation-level vulnerabilities into correct models and on model-checkers to generate attack traces. On one hand, security testing is usually performed by penetration testers that either use manual techniques, based on their knowledge and by following guidelines like the OWASP testing guide[1], or automated techniques thanks to penetration testing tools[2]. Such tools differ from our work as they do not rely on models for generating test cases. As already mentioned multiple times Doupé et al. [92] evaluated such 'point-and-click' pentesting tools and found that the crawling part (automatically discovering new pages of a web application) is a critical and challenging task for these tools that influences the overall ability to detect vulnerabilities by black-box web vulnerability scanners. The following list provides a summary of some vulnerability scanners. The first four are analyzed by Doupé et al. and are selected because they are not dominated by any other scanner analyzed by the same authors. Unfortunately they are all commercial tools and therefore, access to technical details is limited. The last one (KameleonFuzz) is developed by Duchene et al. [94]. These tools have the same goal as SPaCiTE, namely finding vulnerabilities in Web applications. Demonstrated by Doupé et al. [92], such tools often do not find non-trivial vulnerabilities. Therefore, the gap that SPaCiTE is addressing is finding non-trivial vulnerabilities that require multi-step attacks. SPaCiTE generates test cases for such vulnerabilities with formal abstract behavioral description. Since all of the described black-box tools below are based on dynamically learning the web application with a crawler, such tools do not, and do not need to address the abstraction gap between an abstract view of a web application and the corresponding implementation. Tools like Vera and IBT operate on models and they address the abstraction gap with mappings to protocol level messages. Since SPaCiTE is a model-based approach, it addresses this abstraction gap, compared to black-box scanners. In contrast to Vera or IBT, SPaCiTE addressed the abstraction gap with a mapping that does not require protocol level knowledge or API level programming.

Acunetix [5] is a commercial vulnerability scanner with an automatic JavaScript analyzer for AJAX and Web 2.0 applications. It can handle complex web technologies such as SOAP, XML, AJAX, and JSON. Acunetix tests a web application against many vulnerabilities, including XSS and SQL injections. During the crawling activity, the scanner builds the site's structure and enumerates all files. During the scanning phase, Acunetix emulates a hacker to attack the web application [4].

Burp [10] is a commercial integrated platform to test web applications. According to the documentation [10], it allows to combine advanced manual techniques with state-of-the-art automation. Among other components, it consists of a proxy to inspect and

[1] `https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf`

[2] `http://sectools.org`

modify traffic data, an application-aware spider, a scanner for automatically detect vulnerabilities, and an intruder tool for performing attacks. The intruder performs fuzzing, enumerating identifiers, brute-force attacks etc. to attack the web application.

N-Stalker [17] is a web security assessment tool. It makes use of the N-Stealth HTTP Security Scanner and is dedicated to vulnerabilities like XSS, SQL, buffer overflow and parameter tampering. Furthermore, it has access to a huge web attack signature database with 39'000 entries.

Webinspect [41, 40] is a commercial assessment tool offered by HP that can be used to identify known and unknown vulnerabilities in web applications. It is both automated and configurable. The tool consists of assessment agents to crawl the application, a security engine to evaluate the results, and audit engines to execute real exploits and to build the verdict.

KameleonFuzz was developed by Duchene et al. [94] and is a black-box fuzzer for XSS vulnerabilities in web applications. The tool specifically addresses the problem, how to fuzz a parameter of a request to a web application and how the effect can be observed. To solve these problems, KameleonFuzz makes use of a genetic algorithm which is guided with an attack grammar.

The crawling part of all these black-box vulnerability scanners is very crucial. One way to overcome this weakness is to use a white-box testing approach. Such approaches are often dedicated to applications written in a specific language.

Ardilla. The Ardilla tool [141] looks for SQL injections and XSS attacks in PHP applications. It creates SQL and XSS attack vectors by first creating sample inputs, then symbolically tracks those inputs by executing the source code and finally mutates the initial sample inputs to generate concrete exploits.

Apollo. The Apollo tool [61] uses a combination of concrete and symbolic execution and explicit-state model checking to generate test cases for dynamic Web applications. The tool implements these techniques for the PHP programming language, using a modified PHP interpreter. To build the verdict, Apollo monitors application crashes and validates the output according to the HTML specification.

SAGE. Godefroid et al. [111] developed SAGE, an automated white-box fuzzing testing approach. The tool is dedicated to the x86 instruction-level and can be applied to an arbitrary file-reading Windows application. SAGE takes a well-formed input, runs the application and records the actual execution. It collect all constraints, negates one by one and uses a constraint solver to produce new inputs (concolic executions). Since SAGE is dedicated to windows applications, it cannot directly be applied to web applications. Nevertheless since the tool seems to work extremely well, an adaptation to web application would make SAGE a competitive tool to SPaCiTE.

Finally, the following two model-based security test generation tools are developed during the SPaCIoS EU project by project partners.

VERA. Blome et al. [70] developed VERA, a flexible model-based vulnerability testing tool. It is based on extended finite state machines and operates without access to source code. Vera allows the Security Analyst to define attacker models. Such models separate the malicious inputs from the behavior description. The VERA framework consists of one or more instantiation library, that contains malicious input values, a configuration file with application specific information, and a model file which is instantiated and run on the SUV. Besides the fact that VERA abstracts away from low-level implementation details, the configuration file still allows to specify Cookie and Header information. VERA mainly differs to SPaCiTE because VERA is based on attacker models that require the Security Analyst to act like an attacker. Using SPaCiTE, this task is performed automatically by verification tools at ASLan++ models and the Security Analyst, if he provides the formal specification, needs to specify the security properties and is focused on 'what' he wants to achieve, but not 'how'.

IBT. This tool was developed by Armando et al. [60]. The IBT tool is dedicated to generating test cases from security protocol models. The IBT tool starts with models that are initially already insecure. Such models consist of transitions that are represented by rules. The operationalization of AATs is achieved in two steps. In the first step, each rule is associated (instrumented) with a Java source code fragment. In the second step, the TEE of the IBT tool executes the corresponding code fragments in the order given by the AAT. IBT differs to SPaCiTE since the latter also address the problem how to use initially correct specifications. Furthermore, SPaCiTE abstracts away from implementing the AATs directly at source code level.

On the other hand, formal models and model-checkers have been used for test case generation since at least 1998 [56], in a variety of contexts that has been surveyed elsewhere [106]. Most of this work concentrates on generating test cases that satisfy structural criteria on the model (e.g., state coverage, transition coverage, MC/DC coverage). As there is still no evidence of a strong relationship between such coverage criteria and fault detection effectiveness [154], we choose to rely on a domain-specific vulnerability-based fault model. Our work is closely related to mutation testing [90, 134]. Even though mutation testing usually aims at assessing the effectiveness of a test suite to detect small syntactic changes introduced into a program, it can also be used to generate and not assess test cases. This idea was successfully applied for specification-based testing from AutoFocus, HLPSL or SMV models in the security context [55, 82, 86, 227]. Our work differs in that we start by real vulnerabilities in web applications and correlate them with specific mutation operators. Moreover, we do not stop after test generation but also provide a semi-automatic way to execute the generated test cases on real implementations, in our case, a web application.

More recently, Armando et al. [60] have described work closely related to ours but for protocols instead of web applications. They start from an already insecure model described at the HTTP level. Therefore, this model can directly be used by model checkers to find attacks because the model checker will find a violating trace of the insecure model. The gap we want to address to this work is how to proceed when the initial model is correct because a model checker will not report any attack trace. When it comes to the execution of attack traces, Armando et al. provide an automatic testing approach that relies on a mapping from each abstract HTTP element in the model to HTTP messages suitable for the SUV. The fully automatic procedure is achieved at the price of describing the model at the HTTP level. When a web application is described in terms of abstract logical messages between agents, the work of Armando et al. does not address the additional abstraction layer between high level actions and HTTP messages. Since we focus on web applications we want to address the gap between high level application actions and HTTP messages.

Semantic Mutation Operators need additional information from the model that are beyond syntactical nature. This information might be introduced by annotating the model. Such an extension of an implementation level language to generate test cases is addressed by De Boer et al. [89]. They propose a Java-like specification language to describe the behavior of the test harness. The specification consists of expected traces which are used to synthesize a test environment. A trace is a sequence of method calls and returns (outgoing calls and returns, incoming calls and returns). Our work differs in the sense that their specification language is an extension of a language that is used for the implementation. Models are usually at a much higher level of abstraction. We extend/annotate ASLan++ that is a modeling language and therefore address the gap between a higher level modeling language (ASLan++) and an implementation language. ASLan++ is higher in the abstraction hierarchy than the language used for the implementation, and therefore, it introduces an additional layer of abstraction.

6.1. Papers With The Same Goal

The following papers have the same overall goal but differ significantly from our approach.

Appelt et al. [59] published an automated testing approach for SQL injection vulnerabilities. It is based on input mutation using several mutation operators. Their approach starts with a WSDL file of the web service and an existing test case for each operation the web service performs and should be tested. The mutation operators modify the test case to generate mutated test cases. The mutation operators are based on behavior-changing, syntax repairing and obfuscation techniques. To build a verdict, a database proxy is first trained and later monitoring all SQL queries to detect SQL vulnerabilities. First, Appelt et al.'s approach differs to our approach at the level it operates. While their approach directly operates on the implementation level, our approach operates first at an abstract model level and later bridges the gap to executable test cases. Furthermore, they start with an existing test suite and mutate that test suite to generate a new set of test cases. They claim that such initial test suites have to be manually created if they do not already exist and refer to external tools. Our approach tries to address the delta to come up with such an initial test suite by considering abstract behavioral models and mutation operators to use such models for test case generation. Furthermore, since Appelt et al. [59] directly start with executable test cases, their work does not, and does not need to address the issue of bridging the abstraction level.

Lebeau et al. [144] published a paper that is very close to our work. They propose a model-based vulnerability testing approach for web applications to automate model-based vulnerability testing. The goal is to improve both accuracy and precision of vulnerability testing. The input for the proposed approach consists of different UML specifications that describe the behavioral aspects of web application, as well as vulnerability test purposes. The approach itself consists then of four major steps — (1) formalizing test purposes from vulnerability patterns, (2) modeling behavioral aspects of the web application, (3) automatically producing abstract test cases, and finally (4) concretizing and executing test cases. The contribution consists, among others, of using vulnerability test patterns as test purposes, and the modeling activities dedicated to vulnerability testing. This work is extended in a follow up publication [216] where the described approach is applied to detect multi-step XSS vulnerabilities. The proposed approach is very close to our work. In both approaches, models are used as input that describe behavioral aspects of the web application. Furthermore, steps 2 - 4 of the approach are conceptually equivalent. Nevertheless step 3 is addressed differently. We consider our delta especially in step 1 of the approach where we focus on ASLan++ models that serve as input for model checking technologies, whereas Lebeau et al. [144] use different kind of UML diagrams (class, object, state diagrams). Furthermore, the high level behavioral model in our approach is independent of how the functionality is implemented at the beginning but is then annotated by the Security Analyst with technology dependent information to guide the test case specification. These annotations are used to mutate the model by injecting source code level vulnerabilities into the model. That differs from the proposed approach that considers a formalization of the environment in order to exploit vulnerabilities. Due to the different approach in step 1, also step 3 differs. We consider model checking techniques that try to find a way to exploit the injected vulnerabilities. This is done by checking if specified security properties are violated by the injected vulnerability. In the approach of the paper, the test patterns directly describe the behavior of the environment and therefore, security properties are only captured indirectly.

Fu et al. [108] propose an approach to detect SQL injection vulnerabilities. It introduces a static analysis framework called SAFELI to detect such attacks at compile time, using symbolic execution. Therefore, it operates on the source code of the application. By identifying so called hotspots, statements that send SQL queries, symbolic execution techniques and constraint solvers are used to generate concrete user input that exploit the SQL vulnerabilities. The most significant difference to our approach is the input artifact. Whereas Fu et al. [108] directly operate on executable code, our approach is based on an abstract representation, called a formal specification.

Martin and Xie [154] published a work about a fault model and mutation testing of access control policies. They evaluated the research question how strong the correlation between structural criteria and fault-detection capability is, and characteristics about different mutation operators. Martin and Xie [154] take as input XACML policies and consider three different test suites. The first test suite is generated randomly, the second one is the result of minimizing the first one according to a set of requirements, and the first test suite is generated according to a change-impact analysis of two policies that were mutated. Performing an experiment on eleven different policies, the author report the following major findings. For certain policies, there is a correlation between structural criteria and fault-detection capabilities. E.g., a test suite that does not cover policy or rule artifacts does also not kill mutants generated by mutating the same artifacts. At the other side, even if a 100% structural coverage is achieved by a test suite, this does not guarantee a good mutation killing rate. This all indicates that there is a correlation between structural coverage and fault-detection capability, but it is not strong enough to reach a satisfying level. Comparing Martin and Xie [154]'s work with our approach, it differs in the following sense. In terms of the initial input artifacts for the testing approach, Martin and Xie [154] focuses on XACML policies that can be directly used as configuration files for an access control policies. In our work, we initially start with a formal specification of a web application, that is specified at an abstract level and the abstraction gap needs to be addressed to test the corresponding SUV. That means that generated attack traces based on mutated models are at the same abstract level as the input model. To establish a correlation between abstract security issues and issues at the implementation level, these AATs need to be mapped to executable test cases. Due to the fact that different input artifacts are used, the corresponding vulnerabilities differ as well. While Martin and Xie [154] concentrates on faults while specifying an XACML policy, we focus on web vulnerabilities like XSS and SQL-related faults.

Hu and Ahn [124] published a work about enabling verification and conformance testing for access control models. The goal is to integrate automatic analysis and conformance testing for access control systems into the Assurance Management Framework (AMF). The proposed approach combines both formal verifications and conformance testing. Verification is used to check if the formal specification of a security model and the policy fulfill a given set of security properties. Conformance testing is used to check whether the implementation of the formal specification and the specification comply. For this purpose, test cases are automatically generated from the formal specification. The paper's contribution consists of an enhanced AMF framework for rigorous analysis and testing; and a methodology to combine model-based verification and model-based test generation. In terms of test case generation, the approach distinguishes between positive and negative test cases. Negative test cases are generated by taking formal access control model specifications into consideration that do not satisfy the constraint specification. Positive test cases are generated by verifying the access control model against the negated constraint specifications. The described publication differs from our approach in the following aspects: (1) Whereas Hu and Ahn focus on access control policies, we focus on web applications. This implies that our formal specification not exclusively consist of the access control part but includes the behavior description of the web application as well. Furthermore, the considered properties in our approach go beyond pure access control and capture dedicated web security aspects like XSS and SQL as well. When it comes to test generation from a formal specification, we assume for our approach that verification techniques do not report any counter examples. In contrast to Hu and Ahn, our considered formal specifications do not initially violate the specified security properties. Therefore, we need to mutate the model to generate negative test cases. Hu and Ahn face a similar problem for generating positive test cases, where they mutate the property instead of the model.

Clark et al. [85] published a paper about semantic mutation testing tool for C. They use semantic mutation operators to capture a different class of errors than what traditional mutation testing does. They focus on possible misunderstanding of the semantics of the description language. For their mutation operators, it is sufficient to focus on the syntax of the specification, including type information since problematic source code level expressions and statements can be identified syntactically. This differs from our work since e.g., defined ASLan++ facts or statements do not have a predefined semantics. They interplay with security properties and need to be annotated by semantic keywords to let Semantic Mutation Operators inject corresponding vulnerabilities. Furthermore, we use Semantic Mutation Operators to represent security-related vulnerabilities and try to exploit them to violate specified security properties, whereas Clark et al. use them to test for different functional behavior depending on used compiler and configurations. Since the Semantic

Mutation Operators for C directly apply on source code, no abstraction gap needs to be addresses. This is different in our approach where we assume that we do not have direct access to the source code. We apply Semantic Mutation Operator at an abstract model level and therefore, our approach has to address the gap between abstract and concrete test cases.

6.2. Papers about Model-Based Security Testing

Mouelhi et al. [166] focus on security policies. They propose a model-based approach for the specification, deployment and testing of security policies in Java applications. The approach starts with a generic security meta-model of the application. It captures the high level access control policy implemented by the application and is expressed in a dedicated security SDL. Before such a model is further used, the model is verified to check the soundness and adequacy of the model with respect to the requirements. Afterwards the model is automatically transformed to policy decision points (PDP). Since such PDPs are usually not generated from scratch but are based on existing frameworks, the output of the transformation is e.g. an XACML file that captures the security policy. This transformation step is essential in MBT since an identified security issue at model level does not automatically imply the same issue at implementation level, nor does a model without security issues automatically imply the same on the implementation. Mouelhi et al. make use of mutations at the model level to ensure that the implementation conforms to the initial security model. An existing test suite is executed on an implementation generated from a mutated security model. If such mutants are not detected by the existing test suite, it will be adapted to cover the mutated part of the security model as well. Finally the test objective is to check that the implementation (security policy) is synchronized with the security model. Mouelhi et al.'s approach and our approach share the commonality that both start with a formal high level specification of the SUV. Nevertheless there are major differences between the two approaches. Since Mouelhi et al. use mutation testing for evaluating and improving an existing test suite, we use mutation testing to generate test cases in the first run. In terms of the application domain, Mouelhi et al. are focused on access control policies, whereas our work is dedicated to web applications.

Woodraska et al. [228] published an approach for security mutation testing of the FileZilla FTP Server. As input, they consider the source code of the server and mutate the source code according to causes (e.g., design-level and implementation level defects) and consequences of vulnerabilities (e.g., STRIDE attacks). The mutation is a manual process where code is deleted, modified, and added. To perform this task, a comprehensive study of the source code was necessary. For this publication, the authors created 30 different mutants. The generated mutants are used to evaluate the quality of two test generation techniques based on threat models, represented as attack trees and attack nets [229]. Woodraska et al. [228] conclude that both test generation techniques have similar vulnerability detection rates but cannot detect vulnerabilities that are not covered by the threat models. This publication differs in many aspects from the approach described in this Ph.D thesis. First, we consider an abstract formal specification of the SUV whereas Woodraska et al. [228] directly operate on the source code. Second, our approach automatically applies mutation operator after the Security Analyst has annotated model blocks with their semantics. In this respect, both approaches need manual input, although our approach automates the application of the mutation operators. Third, mutation operators are used by Woodraska et al. for evaluating existing test case generators, whereas in our work, they are used to directly generate the test cases. Finally, the class of considered vulnerabilities differ due to different SUVs.

Tang et al. [204] published a model-guided security vulnerability discovery approach for network protocol implementation. Their approach is very related to our work since they introduce mutation analysis and model checking into fuzz testing. They concentrate on black-box security testing only and do not depend on source code and implementation details. The model-based approach starts with a formal specification of the protocol expressed using a parameterized extended finite state machine. By applying one mutation (at the model-level) to the original model, they generate a large number of mutants. The work differs in terms of the domain, where the approach is applied, the test case generation, and the involved artifacts. Tang et al. [204] focuses on protocol implementation and their dedicated vulnerabilities. The protocol level and the involved vulnerabilities significantly differ to web application and their typical vulnerabilities. Both their and our approach rely on mutation operators to let a model checker report counter examples that are interpreted as test cases. These abstract counter examples are mapped to input/output packets and a fuzzer is used to generated test cases for the real system. Our work differs because the applied mutation operators are not only used to let a model checker generate counter examples, but also during the test case instantiation. The applied mutation operators are used as a selection criteria for concrete, executable exploits. In addition, since we test web applications, we map abstract attack traces to executable actions in the browser instead of packages at the protocol level.

Ramakrishnan and Sekar [179] published a paper about model-based vulnerability analysis of computer systems. They focus on modeling different components of a system, compose them so that they represent different communication scenario and model-check the composition of them against a formally specified security property in LTL. The input models for the approach are either generated from source code, or e.g., the vendors of the different components provide them. Since Ramakrishnan and Sekar [179] focus on computer systems and not web services, the covered vulnerabilities (e.g., concurrency, file permissions, printing, etc.) are different from our approach (XSS attacks and SQL injections). Furthermore, in Ramakrishnan and Sekar [179]'s approach, vulnerabilities are mainly discovered due to the composition of different models. Compared to our approach, we inject vulnerabilities by applying Semantic Mutation Operators. That allows us to also use formal specifications for test case generation where initially verification techniques do not report counter examples. Finally, our work differs from Ramakrishnan and Sekar [179] since we do not stop at the abstract level but generate executable attack traces from the reported counter examples. This includes addressing the gap between abstract AATs and the SUV so that the implementation of the modeled system can be tested.

The paper *Model-based Security Vulnerability Testing* [183] proposes a fault-based approach to test case generation. The implementation uses state transition and program models taken from the design of the Spec-Explorer tool, specified in the corresponding Spec# language. The whole approach is based on three models — desired behavior model (which captures the key aspects of application) needed for testing process, an implementation model needed for exploits, and an attacker model needed for indicating which vulnerabilities are used by an exploit. The attack model works as a test purpose that captures what the attacker wants to do. It is a dedicated, separate model so that attacker behavior can change dynamically. In particular it captures the precondition for the attack. This helps to exclude all those scenarios, where the attack is not possible. For test case generation, the implementation model is combined with the attack model in order to provide an exact localization of security vulnerabilities. Although the approach allows that the behavioral specification can be incomplete or underspecified regarding a security issue, vulnerabilities are nevertheless caught by implementation and attack model. To find counter example using a constraint solver, the approach combines all three separate models and defines when a transition of the implementation is faulty with respect to a specification. This is called a faulty context. Such a faulty transition is considered as a negative test case. The approach tries to find out if a particular transition defined by the attacker model is present in the implementation model in a faulty context. The proposed approach is close to our work in the sense that both approaches are model-based and generate test cases to detect security vulnerabilities. To do so, both approaches focus on the application level, rather than the network level. Nevertheless we see a couple of deltas to our work: The paper approach focuses on modeling attacker knowledge so that attacker behavior can change dynamically. In our case, we use the built-in Delov-Yao attacker in the model checker. Since security vulnerabilities are closely related to an implementation, the paper's approach introduces an implementation model. In our case the implementation level details are introduced as fault models that describe source code level vulnerabilities. These vulnerabilities are covered as Semantic Mutation Operators that mutate the ideal behavioral specification. The model checker's task is to check if the introduced vulnerability can be used to violate the specified security property, independent if that vulnerability is part of the implementation. In contrast to Salas et al.'s approach, that allows to generate also test cases for perfect specifications and implementations.

Marback et al. [152] published a paper about security test generation using threat trees. This work is similar to our approach since it generates security related test cases for web applications from abstract models. Marback et al. start with building threat trees at the design level and then combine attack steps with data flow diagrams of the SUV. This is similar to our approach since it is an abstract view of the SUV and generated test cases from such threat trees are not automatically executable. The threat tree approach differs to our approach in the sense that the Security Analyst in our approach is focused on a behavioral model with semantics annotations about implementation level technologies and functionality. Then, vulnerabilities, properties, and verification techniques like model checkers are used to automatically generate attack traces. In Marback et al.'s approach, the Security Analyst is more focused on attacks directly and describes these decision-making processes of an attacker manually. For operationalizing test cases, Marback et al. [152] directly map model-level elements to implementation-level constructs. In our approach, we let the Security Analyst map model-level elements to abstract browser actions using a declarative Domain Specific Language (DSL). Thus, we address the gap to not bother the Security Analyst with implementation-level details.

Authors	Title	MSSec	SecME	TSC	MES	EM	EL
Mallouli et al. [150]	A formal framework to integrate timed security rules within a TEFSM-based system specification	SecP	n.s.	FB	Prot	Effe+ Effi	Exec
Mallouli et al. [149]	Modeling and Testing Secure Web-Based Systems: Application to an Industrial Case Study	SecP	n.s.	FB	Prot	Effe+ Effi	Exec
Martin et al. [155]	Assessing quality of policy properties in verification of access control policies	SecP+ FSecM	n.s.	FB	Prot	Ex+ Effe+ Effi	Abs
Zhou et al. [245]	Protocol Security Testing with SPIN and TTCN-3	SecP+ FSecM	A+T	FB	Pre	Ex	Abs+ Exec

Table 6.1.: Model of System Security = Properties & Test Selection Criteria = Fault-based ('MSSec' = Model of System Security; 'SecME' = Security Model of Environment; 'TSC' = Test Selection Criteria; 'MES' = Majurity of Evaluated System; 'EM' = Evidence Measures; 'EL' = Evidence Level; 'SecP' = Security Properties; 'FSecM' = Functionality of Security Mechanisms; 'A' = Attack Model; 'T' = Threat Model; 'FB' = Fault-Based; 'Prot' = Prototype; 'Pre' = Premature System; 'Ex' = Example Application; 'Effe' = Effectiveness; 'Effi' = Efficiency; 'Abs' = Abstract; 'Exec' = Executable; 'n.s.' = not specified)

6.3. Papers in the Context of Property- and Fault-based Test Selection

In 2015, we published a paper [104] in the Software Testing, Verification and Reliability (STVR) Journal which provides a taxonomy and a systematic classification in the context of model-based security testing. This work is discussed in details in Appendix A. As a summary, we performed a comprehensive literature survey. In this section we use the classification to discuss papers that use both property- and fault-based test selection criteria. The identified approaches are listed in Table 6.1 and are interesting, since our approach is based on (security) properties and fault-based test selection criteria as well. From a set of 119 papers, four papers fulfill these criteria. In the following we discuss these papers.

Mallouli et al. [150, 149] describe an approach to integrate timed security rules into TEFSM-based system specifications. An example of such a rule is e.g., that two successive travel requests to a Travel Reservation Web Application are separated by at least two minutes. The main contribution of this work are new algorithms to integrate timed rules into specifications, a correctness proof, and an industrial case study with France Telecom. The described approach takes as input a TEFSM functional description of the system, a set of security rules, and an implementation of the system. The security rules are then integrated, abstract test cases are generated with the TestGenIF tool, and abstract test cases are mapped to HTTP with tclwebtest. Our work differs in the sense that we assume the security properties to be given and part of the initial formal specification. As an additional input, we consider potential source-code level vulnerabilities whose abstract representation is injected into the formal specification. Test cases are then generated by checking if introduced vulnerabilities violate a specified security property.

Martin et al. [155] assess the quality of policy properties by verifying access control policies, usually given as XACML policies. Such properties are defined at a higher level than the policy specification. To verify the properties, a set of mutated policies is generation by seeding a single fault per mutated model. Afterwards they are checked if they still fulfill the policy properties. By verifying the mutated models, the properties are determined that interact with rules in the policy. If a mutated model does not violate the policy properties, then the quality of the properties is insufficient. In such a case, a rule is not covered by the property set. In contrast to the usual case where the request set to test the policy would be improved, this paper proposes to improve the property set. During this approach, the request set that let a mutated model violate the property set is considered as a test suite for the policy. The delta to our work is that we assume the specified properties to be validated, meaning they represent what the security expert indeed wants. Furthermore we don't focus on Access Control Mechanism but consider the full functionality of the application. In addition, our mutation operator capture a more general range of vulnerabilities and are not limited to access control vulnerabilities.

Zhou et al. [245] describe an approach for protocol security testing with the SPIN model checker and TTCN-3 for test case specification. The work focuses on specification security of communication protocols. It is a general method for protocol security testing that addresses effective detection of specification vulnerabilities and efficient testing of protocol implementations. The SPIN model checker is used to find specification flaws with respect to LTL security constraints, which are partitioned into system security (deadlock, invalid end, none-progress circle) and protocol requirements (confidentiality, integrity, authorization, authentication, availability, non-repudiation). To simulate malicious entities, the approach considers threat models. After the model checking phase, mapping rules translate counter examples of model checking to TTCN-3. The describe approach is similar to our work in the sense that both approaches start from a formal specification and end with executing test cases on a SUV. They both include the steps (1) model checking formal specification against security properties, (2) counter examples are considered as test cases (3) operationalization and execution of test cases on the SUV. Nevertheless the delta between the Zhou et al.'s and our approach is: (1) Zhou et al. consider protocols as SUV, whereas we consider web applications. (2) Besides the different levels where these SUVs are used in a software stack, also the types of vulnerabilities are different. In our work, we consider SQL injections and XSS vulnerabilities, that, in most of the cases, are not part of security protocols. Finally, our approach includes a mutation step where we modify the initial formal specification. This addresses the important issue that also test cases for an initially correct model can be generated.

Conclusion

In sum, all the above discussed papers show that the combination of formal verification techniques combined with penetration testing has not been addressed for web applications.

In this context the question how to use correct formal specifications and corresponding security properties, that do no generate AATs using verification tools like model checkers, for test case generation, is still an open issue. Therefore, this Ph.D thesis contributes to this gap with the methodology discussed in Chapter 2, including different mutation operators. Furthermore, the discussed papers also show that the gap from abstract to executable test cases is still an open issue, which we address with the intermediate browser level to simplify the operationalization of test cases and successfully handling sophisticated client-side browser technologies like JavaScript or Flash.

7. Conclusion

In this Ph.D thesis, we developed and evaluated a semi-automatic security testing approach with fault models and properties. Many web application surveys show that two of the most severe vulnerabilities for web applications are dedicated to Cross-site Scripting (XSS) and Structured Query Language (SQL) attacks. Therefore, the considered fault models reflect such issues and are used to generate test cases. A comprehensive survey of papers published between 1996 and 2013 in the area of model-based security testing has shown that only a few papers propose modeling source code level vulnerabilities at an abstract level and combine them with verification technologies like model checkers to generate test cases for web applications. Furthermore, Doupé et al. [92] published a paper that demonstrates that automatic security scanners do not find 8 out of 16 well-known vulnerabilities. Issues like reaching a sufficient depth in the web application, authentication, client-side technologies like Flash and Javascript are major issues for such tools. Therefore, we propose a model-based approach in this Ph.D thesis that starts with a formal specification of web applications. Model checking such formalizations either directly leads to a counter example, or the model checker does not find any issues, either because they are correct or long-running specifications[1]. While the first kind of models can directly be used for test case generation — by interpreting the counter example as a negative test case, the second kind of models are not immediately useful. In this Ph.D thesis, we focus on the latter kind of models. The literature suggests in such situation structural coverage criteria to generate test cases. Since such criteria are not optimal in the context of security testing and behavioral models of web applications, we propose Semantic Mutation Operators.

To make use of models that do not generate Abstract Attack Traces (AATs) for test case generation, either the specified security properties Φ or the model M needs to be mutated. If the underlying model M is correct, and therefore, $M \models \Phi$, mutating properties is not immediately useful for test case generation since all traces of the correct model M satisfy all original properties ($\forall \phi \in \Phi : M \models \phi$). If the security properties are mutated, the model checker can only report traces from the correct model that all satisfy the initial security properties. Therefore, the model M itself needs to be mutated to M' so that a trace in M' exists that violates at least one specified property ($\exists \phi \in \Phi : M' \not\models \phi$). Model checking tools might now report traces of the mutated model that violate an initial security property $\phi \in \Phi$.

We call such traces Abstract Attack Trace (AAT) that consists of a sequence of abstract messages exchanged between different entities. To operationalize such AATs they need to be instantiated. Since web applications are usually accessed with the help of a browser, we consider the browser level for this process. Executing test cases requires the help of the Security Analyst since the abstraction gap has to be bridged between the AAT and the System Under Validation (SUV). Asking the Security Analyst for a mapping of an AAT to a

[1] A long-running specification is a model, where the model checker does not find any issues in a reasonable time frame.

sequence of actions performed in the browser seems to be easier and more convenient than providing a mapping directly to protocol level messages because less code has to be written manually and technical API details can be neglected. In addition the mapping of AATs to executable source code consists of application-dependent and application-independent information. Therefore, we split the process of making AATs operational into two different steps by adding an additional intermediate level in between the AAT layer and the implementation layer. These three layers have different purposes. Layer 1 describes the AAT as a sequence of abstract messages as given by the output of the model checker. Therefore, the first mapping takes as input an AAT and maps it to a sequence of abstract browser actions, expressed in the Web Application Abstract Language (WAAL) language at the second layer. WAAL is a language that we developed for this Ph.D thesis to describe how exchanged messages between agents can be generated and verified in terms of actions a user performs in a web browser. Providing this mapping is a manual task but supported with a Domain Specific Language (DSL). The second step is mapping WAAL actions to executable API calls using a specific framework. In our case we make use of the Selenium framework [26] and provide a mapping from WAAL to selenium API calls. Once such a mapping is defined, it can be automatically applied and reused for any test case. Therefore, layer 3 describes the instantiated attack trace in terms of source code.

We evaluated our approach using three different formal specifications. While WebGoat is a web application already known from the beginning, Wackopicko and a Bank Application were only considered during the evaluation. The overall evaluation addressed effectiveness and efficiency aspects of SPaCiTE, that uses a vulnerability-based fault, compared to approaches that use a syntax-based fault model. To evaluate the effectiveness of SPaCiTE, we evaluated WebGoat with a vulnerability scanner called ZAP. Wackopicko was intensively analyzed by Doupé et al. [92], and the bank application was examined using several tools by Master students of the Technische Universität München. In the first part of the evaluation, we showed that SPaCiTE generates AATs that correspond to more difficult and sophisticated attacks like multi-step stored XSS and multi-step stored SQL attacks, attacks that automatic security scanner usually do not find. In the second part of the evaluation, we elaborated the efficiency of SPaCiTE by comparing four different sets of mutation operators — first-order, higher-order, Syntactic Mutation Operators, and Semantic Mutation Operators.

We are very well aware that the conclusion we draw is based on a few case studies. They provide important insights but the evaluation data does not allow to generalize these results.

In terms of effectiveness, we conclude that SPaCiTE does find the non-trivial stored XSS and stored SQL vulnerabilities in WebGoat and Wackopicko both at the abstract, as well as on the implementation level. In contrast, while ZAP fails to find the stored XSS vulnerability in WebGoat, black-box scanners like Acunetix, AppScan, Burp, Grendel-Scan, Hailstorm, Milescan, N-Stalker, NTOSpider, Paros, W3af, and Webinspect do not find these two vulnerabilities in Wackopicko. Therefore, the proposed Semantic Mutation Operators are powerful enough to find the stored XSS and stored SQL vulnerability both at the abstract model level, as well as at the implementation level.

In terms of efficiency, we conclude that applying higher-order Syntactic Mutation Operators to the three case studies generate an amount of mutated models that are impractical to model check. Higher-order Syntactic Mutation Operators generate up to 3800 and 12’000 mutated models for the Wackopicko and Bank specification respectively. At the same time, the Semantic Mutation Operators generate approximately 200 and 300 mutated models respectively. While model checking those 300 mutated models requires approximately 30 hours, model checking those 3800 models require 312 hours. These numbers do not generalize but they provide an intuition about the computation power required for model checking. Furthermore, applying Syntactic Mutation Operators generate a set of mutated models where between 2% (94 vs. 3851 models) and 13% (20 vs. 158 models) of the models generate an AAT. The ratio is much higher when Semantic Mutation Operators are applied, namely between 12% (24 vs. 204 models) and 53% (10 vs. 19 models). Comparing the number of generated AATs shows that Syntactic Mutation Operators generate between 1.3 (18 vs. 14 AATs) and 2 (20 vs. 10 AATs) times as many AATs as applying Semantic Mutation Operators. This has several reasons: All relevant input message parameters are already considered by the Semantic Mutation Operators so that no parameter is left that could be used by Syntactic Mutation Operators to violate specified security properties. Furthermore, injecting a vulnerability to a formal specification only leads to an AAT if a security property is violated. Depending on how intensively the Security Analyst annotated the formal specification, most/all of the possible AATs are already covered by Semantic Mutation Operators.

Whether Syntactic Mutation Operators perform more efficiently than Semantic Mutation Operators depends on the strategy how AATs generated by Syntactic Mutation Operators are handled. Illustrated on the Wackopicko formal specification, automatically applying higher-order Syntactic Mutation Operators generates around 4000 mutated models, requires 320 hours of model checking, and the generation of around 680 executable test cases. These numbers are based on the brute force approach for AATs for with it is unclear how to concretize and execute them. By manually applying around 40 semantic annotations, the number of mutated models can be reduced to 350, to 33 hours of model checking, and to the generation of 60 executable test cases. Therefore, the effort for the manual task of annotating a formal specification competes with the effort of handling a bigger test suite that also includes test cases without vulnerability-based rationales. This conclusion obviously does not generalize but nevertheless provides important insights.

As a summary of the lesson learnt, we want to stress the following conclusions:

1. We believe that the combination of behavioral models and security properties increase the success of finding vulnerabilities because the application logic can be respected. E.g., the test cases generated for the Bank application first register a new account and activate it before a client logs in with malicious data. Such test cases combine application logic with attacks. They increase the success of the attack compared to a test case that tries to login without first creating and activating an account.

2. A crucial quality characteristic of the formal specification is an accurate and careful data flow modeling. E.g., receiving a value and storing it in the database generates two unique copies of the same data value. Since storing the value might make the sanitization ineffective, it matters which of the two copies is sent back to the client.

3. To find non-trivial vulnerabilities in the evaluated three use cases, higher-order mutation operators (either syntactic or semantic) are required. To generate test cases that find reflected XSS and SQL vulnerabilities, it is not guaranteed that first-order mutation operators are sufficient. Similar, to generate test cases that find stored XSS and SQL vulnerabilities, it is not guaranteed that second-order mutation operators

are sufficient. The required order depends on the model, and not on the type of vulnerability.

4. The difference between Syntactic and Semantic Mutation Operators is the locations where the mutation operator is applied and the kind of value-tracking facts that are introduced. While the location where the model is mutated is mainly responsible for violating the specified security properties, the value-tracking facts are used during the operationalization of AATs. Therefore, Syntactic Mutation Operator generate much more mutated models but that set contains those mutated models that are also generated by Semantic Mutation Operators. Thus, Syntactic Mutation Operators eventually cover the same test cases as generated by Semantic Mutation Operators, but the computation overhead is massive.

5. By using WAAL as an intermediate language while operationalizing an abstract test case, the Security Analyst has to specify significantly less lines of codes (approximately 40% for the WebGoat and 50% for the Wackopicko use case) compared to providing the executable test case directly.

6. To get a more comprehensive test suite, attack injection is required. A pure vulnerability injection does not cover the effect of the malicious input. Knowing the semantics of such a malicious input reveals new test cases, that are missed by pure vulnerability injections.

7. While sophisticated attacks often consist of multiple steps, using verification techniques based on behavioral models definitely can be used for automatically generating such test traces. E.g., it is the strength of model checkers to find traces in a model that violates a defined security property. At the same time, attacks are not only sophisticated because they consist of multiple steps, but also because the input data has to follow a syntax and is encoded in different ways. In this respect, verification techniques in combination with behavioral models are too abstract as that encoding information could be part of such specifications. We believe that considering issues at the level of encoding too fast lead to the state explosion problem that model checker often suffer. We therefore recommend to put future research effort into combining model-based verification techniques with source code analysis techniques like e.g., symbolic execution.

Finally, the overall Ph.D thesis shows, that executable test cases for security-interesting non-trivial vulnerabilities can be generated using the combination of verification techniques and penetration techniques. Using fully automatic model checkers to find abstract security issues is beneficial and can reveal non-trivial traces that violate specified security properties. Nevertheless we realized that the effort of providing appropriate models, model-checking them and addressing all the technical challenges when it comes to an automated approach is very challenging. In particular, model-checking does in general not scale and therefore, the more complex the web application gets, the more details need to be abstracted that either later increases the effort to add them again to gain executable test cases, or prevent the model checker to find vulnerabilities. At the same time, we often realized that the Security Analyst already gets a clear understanding of the relevant components and their test cases during the modeling phase.

Appendix

A. A Taxonomy and Systematic Classification for Model-Based Security Testing

[The content of this appendix is in press of the Software Testing, Verification and Reliability (STVR) Journal.]

As part of the literature study, we performed a comprehensive and systematically survey of existing model-based approaches, written together with Michael Felderer, Philipp Zech, Ruth Breu, and Alexander Pretschner. Because of the observation that only very few classifications for Model-based Security Testing (MBST) approaches exist we published a taxonomy and a classification of 119 MBST relevant papers in [104]. Although the classification is reported in terms of papers, we focused on approaches rather than individual papers since multiple papers for the same approach might have been published. Besides getting an overview of published work, the goal of that survey study is to better understand which areas of MBST are well-understood and evaluated and which areas are not studied enough and are therefore potentially interesting for future research. Using five digital libraries, we collected 119 MBST relevant papers and classified them according to our proposed taxonomy. The classification provides a state of the art overview of MBST approaches between 1996 and 2013. Finally we highlighted some interesting observations based on our classification and discussed potential future research directions. Since MBST is an active research domain, our taxonomy and classification helps to clarify key issues in this domain.

We start in Appendix A.1 with a discussion of already existing classifications for Model-based Testing (MBT), security testing, and MBST and a motivation why further MBST dedicated classifications are desirable. In Appendix A.2 we propose our own taxonomy and apply it to a set of 119 systematically collected papers in Appendix A.3. We report results of the performed classification in Appendix A.4 and discuss them in Appendix A.5. The survey is then used in Chapter 6 as a basis to discuss related work that is important for this Ph.D thesis.

A.1. Existing Classifications

We see our work as a complement to already existing classification approaches [214, 91] by providing classification criteria specific for security aspects and evidence criteria of the approaches. Using the above criteria, we systematically collected 119 papers published in one of the following digital libraries — IEEE Digital Library[1], ScienceDirect[2], Springer Link[3], ACM Digital Library[4], and Wiley[5]. Each publication was then categorized by assigning it to our proposed criteria.

A.1.1. Existing Classifications for MBT

MBT is an active research area and has a big potential to improve test processes in industry [215, 186, 113]. Several MBT classifications already exists:

Utting et al. [214] provide a broad taxonomy for MBT. The taxonomy consists of three general classes:

- 'Model specification' with the sub categories: scope, characteristics, and paradigm. The scope is a binary value and specifies whether the inputs only or the expected input-output behavior of the System Under Validation (SUV) is captured. The characteristics is related to timing issues, nondeterminism, continuous or event-discrete nature of the model. Finally, the last category captures the paradigm and notation used to describe the model (e.g., state-based, transition-based, history-based, functional, operational, stochastic, data-flow notations).

- 'Test generation' with the sub categories: test selection criteria and technologies. Test selection criteria specify which test cases are generated and include e.g., structural model coverage, data coverage, requirements-based coverage, ad-hoc, random and stochastic, and fault-based criteria. The technology category then captures, how the corresponding test cases are identified, e.g., by using random generation, search-based algorithms, (bounded) model checking, symbolic execution, deductive theorem proving, or constraint solving.

- 'Test execution' with the sub category: online/offline. While test cases can be adapted upon the output of the SUV for online testing, they are generated and fixed strictly before test execution for offline testing.

This taxonomy is extended and complemented by Zander et al. [240]. They extend the category 'test generation'with 'result of the test generation', as well as the category 'model specification' with 'MBT basis'. The latter extension captures the elements of a software engineering process which serve as a basis for the MBT process. Furthermore, Zander et al. [240] complement the taxonomy with the category 'test evaluation' which contains the sub categories 'specification' and 'technology'.

Dias-Neto and Travassos [91] published a systematic review and classification. Its main goal is to support the selection of MBT techniques for software projects. The classification consists of the following dimensions: 'type of experimental evidence', 'testing level (unit, integration, system, regression)', 'use of supporting tools', 'model to represent the software to test (UML, non-UML)', 'software execution platforms (embedded, distributed, web services,

[1] http://ieeexplore.ieee.org

[2] http://www.sciencedirect.com

[3] http://link.springer.com

[4] http://portal.acm.org

[5] http://onlinelibrary.wiley.com

web application)', and 'type of non-functional requirements (security, reliability, efficiency, usability)'. Dias-Neto and Travassos [91] complement Utting et al.'s taxonomy because the review is very detailed (it identifies 48 different MBT modeling notations) compared to the first one, which groups them into seven modeling paradigms. At the same time, Utting et al.'s taxonomy allows a higher level way of classifying existing and future MBT approaches.

Anand et al. [57] performed an orchestrated survey of methodologies for automated software test case generation. Anand et al. distinguish three MBT approaches and three types of modeling notations. MBT approaches are categorized into *axiomatic Finite State Machines (FSMs)* and *Labeled Transition Systems (LTSs)*. Modeling notations are categorized into *scenario oriented notations*, *state oriented notations*, and *process oriented notations*. Comparing Anand et al.'s survey with Utting et al.'s one, the categories of the first one can be subsumed under the categories *paradigm* and *test selection criteria*.

Hierons et al. [121] published an extensive survey on using formal specifications to support testing. The survey distinguishes three types of specification languages — *state-based specification languages*, *algebraic languages*, and *hybrid* ones. The paper discusses on how specification languages can be integrated into testing processes by concrete MBT approaches. The discussed types of specification languages can be subsumed under the *model specification* category in Utting et al.'s taxonomy.

Finally, Hartman et al. [118] provide several decision criteria to choose an appropriate language for test behavior and test selection modeling. The decision criteria is based on technological and economical criteria, as well as the distinction between model-driven and model-based testing. The first category distinguishes between *visual vs. textual languages*, *proprietary vs. standard languages*, and *commercial vs. open source tools*. The latter category is divided into *UML-based vs. not UML-based testing*, *online vs. offline testing*, *system vs. test-specific language*, and *domain-specific vs. generic languages*. Except the domain-specificity, the non-technological criteria are covered by Utting et al.'s taxonomy.

As a summary for the discussion of existing work in the area of MBT classifications, we conclude the following: Utting et al.'s taxonomy is well suited to classify MBT approaches on a technological-independent level and to identify trends on MBT approaches. The following three arguments support this statement:

1. Utting et al.'s taxonomy provides high-level criteria.

2. The taxonomy does not consider technological details.

3. The taxonomy subsumes many other classifications as shown above.

A.1.2. Existing Classifications for Security Testing

In this section we want to discuss three classification papers in the area of security testing.

Tian-yang et al. [205] published a list of major methods of security testing. It contains eight different categories: *formal security testing*, *model-based security testing*, *fault-injection-based security testing*, *fuzz testing*, *vulnerability scanning testing*, *property-based testing*, *white box-based security testing*, and *risk-based security testing*. As the listed categories might suggest, Tian-yang et al.'s publication is rather ad-hoc and focuses only on a few approaches.

Shahriar and Zulkernine [192] propose seven different main comparison criteria for security vulnerability testing — *vulnerability coverage*, *source of test cases*, *test case generation method*, *testing level*, *test case granularity*, *tool automation*, and *target application*. The category tool automation is further split into *test case generation*, *oracle generation*, and *test case execution*. Shahriar and Zulkernine [192] used these comparison criteria to classify 20 informally collected approaches. Since the goal is to support security practitioners to select an appropriate approach, the classification mixes abstract and technological criteria.

Finally, Bau et al. [64] classifies eight different security vulnerability testing approaches according to the following criteria — *class of vulnerability that is tested*, *effectiveness against the target vulnerability*, and *relevance of the vulnerability* in productive systems. The goal of this classification is twofold. First, it provides evidence to support the selection of security vulnerability approaches. Second, the survey assesses the potential impact of future research.

To conclude this section, security testing involves the definition and evaluation of attacks. That requires selecting a specific testing approach based on some classification criteria. Existing classifications show that risk-based, fault-injection, fuzz testing, vulnerability coverage, and evidence of an approach are very relevant aspects in this context. Nevertheless more specific testing techniques are necessary. In particular, testing approaches that go beyond functional testing of security mechanisms should be much more addressed in future research.

A.1.3. Existing Classifications for MBST

In this appendix we want to discuss two classification papers in the area of MBST. Although a lot of MBST papers are published, only very few and ad hoc classifications exists.

Felderer et al. [102] provides an extension of Potter and McGraw [173]'s classification by adding model-based approaches. Nevertheless the model-based aspects are restricted to the integration of *risk* into the model. Other model-based aspects are missing. In addition to the *risk* category, Felderer et al. [102] also adds the category *automated test generation*. This category is further divided into the subcategories *complete*, *partial*, and *missing*. Since the classification also includes non model-based approaches, this survey is more general and not fully dedicated to MBST.

Finally, Schieferdecker et al. [187] published a classification that claims that in MBST three different kinds of input models are needed. In their work, they distinguish between *architectural and functional models*, *threat, fault, and risk models*, and *weakness and vulnerability models*. The first category, the architectural and functional models, deal with requirements and implementations of the SUV. An example are access and usage control models. Such models express the expected or desired behavior. For security testing, we not only need the expected behavior but also an understanding, what can go wrong. This knowledge is provided by the second category, the threat, fault, and risk models. Examples of such models are fault and attack models (see CORAS [146]), or fault and attack tree analysis [217]. Finally, the knowledge for the last category, the weakness and vulnerability models, is collected in Common Vulnerabilities and Exposures (CVE) databases like 'The ultimate security vulnerability datasource' [11].

To conclude the section about existing classifications for MBST, only a few ad hoc classifications of published MBST approaches exists. Furthermore, Schieferdecker et al.'s classification does not explicitly cover *test selection criteria*. Due to that, an alignment with Utting et al. [214]'s taxonomy is difficult. In addition, *models of system security* and *security models of environment* is not distinguished neither. Finally, *evidence aspects of MBST* are also not part of Schieferdecker et al. [187]'s taxonomy. Therefore, the goal of our contribution [104] is to address the above mentioned issues in order to be more comprehensive and to better align the classification with Utting et al. [214]'s work.

A.2. Classification Criteria for Model-Based Security Testing

In principle, a MBST approach can be classified using a MBT taxonomy, since both are model-based artifacts. As we have discussed in Appendix A.1.1 Dias-Neto and Travassos [91] consider *security* as a non-functional category and classify 16 MBST approaches. Nevertheless, existing MBST classifications are missing important and crucial categories like *risk-based testing*, *fault-injection*, *fuzz testing*, *vulnerability coverage*, and *evidence*, to name a few. Our survey paper addresses these issues by providing a comprehensive taxonomy that allows to systematically classify existing work specifically dedicated to MBST. An overview over all criteria are shown in Figure A.1.

Our taxonomy is based on two main categories — *filter criteria* and *evidence criteria*.

A.2.1. Filter Criteria

As a model usually describes an infinite set of traces, filter criteria are used to select a relevant finite subset of traces of the SUV. They formally define the security testing objectives and highlight what is modeled. We distinguish between *model of system security*, *security model of the environment*, and *explicit test selection criteria*.

A.2.1.1. Model of System Security

System security models are bound to the SUV since they describe parts of the system. To be more fine grained, we further consider *security properties*, *vulnerability models*, and *functionality of security mechanisms* as potential models of system security. These three types of models are described as follows:

a) **Security Properties** are well-known characteristics like the CIA properties. An approach in this category must at least explicitly define one of the following properties: confidentiality, integrity, availability, authentication, authorization, or non-repudiation.

b) **Vulnerability Models** describe characteristics of the system, a property that is part of the system itself. Without a vulnerability, an exploit cannot be successfully executed on the SUV. Usually such models are expressed in term of deviation from the expected, correct behavior [178].

c) **Functionality of Security Mechanisms** describe concrete means that are used by the SUV to preserve a specific security property. Typical models are access control and usage control models. E.g., a abstract policy can be interpreted as the model that describes the functionality of the security mechanism.

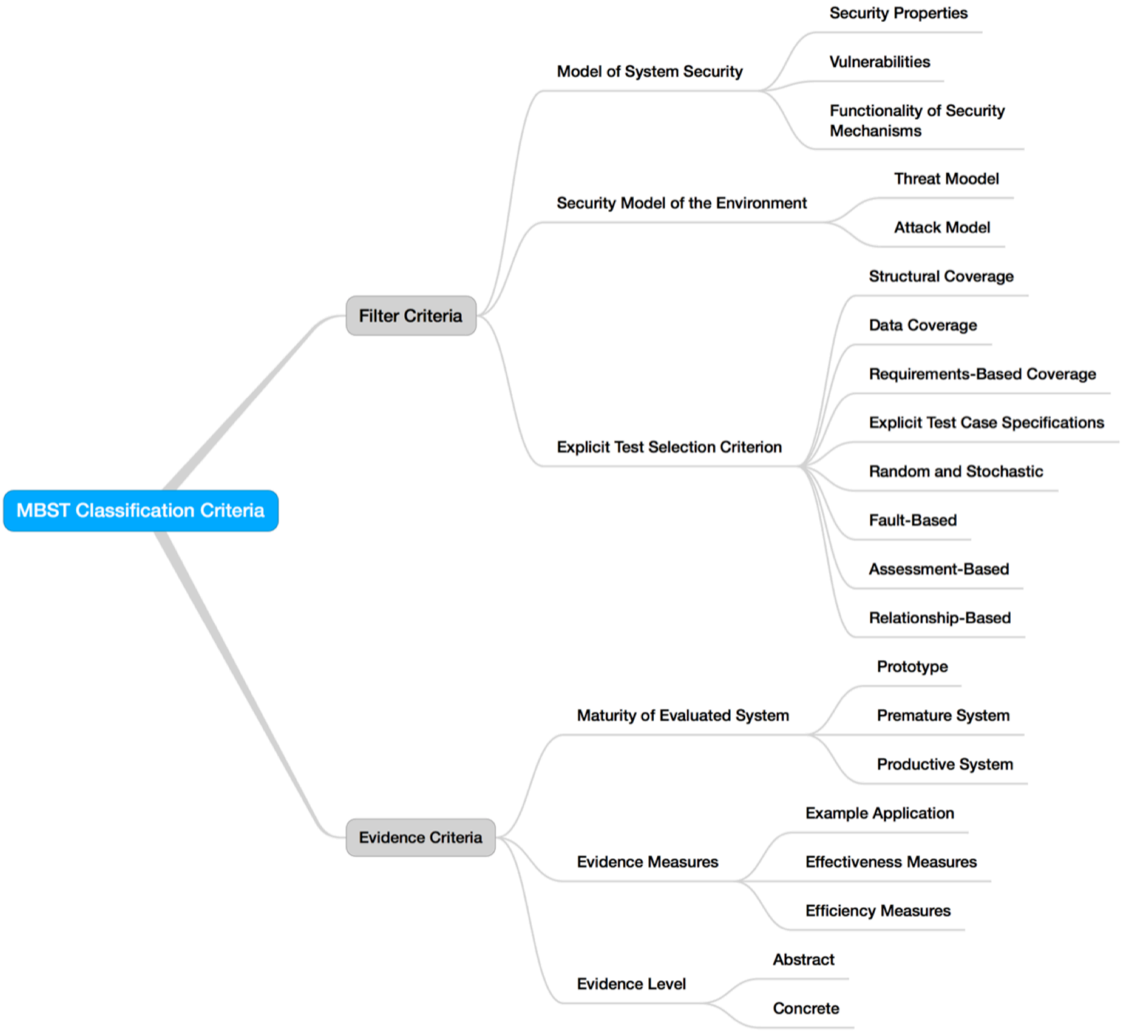


Figure A.1.: MBST Classification Criteria

The above discussed models of system security are not mutual exclusive, meaning that a specific MBST approach might make use of several models. Therefore, we treat the category *model of system security* as multi-select.

A.2.1.2. Security Model of the Environment

Security models of the environment describe technical, organizational, and operating context security aspects of the SUV. In contrary to vulnerability models, they focus on causes or potential consequences of a system behavior. E.g., modeling an exploit of a vulnerability is part of the environment, whereas modeling the vulnerability consists of modeling the SUV. In our survey we distinguish the following two sub categories — *threat model*, and *attack model*:

- **a) Threat Model.** These kind of models describe a potential cause of an incident. In terms of an attack tree, the threat is the root node that motivates an attack [133].
- **b) Attack Model.** These models describe sequences of actions to exploit vulnerabilities. Threats may be considered as their causes. Attack models may be testing strategies such as string or behavior fuzzing, possibly on the grounds of grammars, or syntax models, that generate test cases.

The relationship between threats and attacks can be illustrated by attack trees: An attack tree represents a conceptual view on how an asset might be attacked. It describes a combination of a threat and corresponding attacks. Usually, the top level event (root node) of an attack tree represents the threat. In contrast, each cut of an attack tree, with the root node removed from this cut, is an attack. Several non-root nodes of the tree exploit vulnerabilities. The children of a node express conditions that need to be fulfilled in order to exploit the vulnerability at the parent node. If all nodes in a cut (including the root node) are made true, the corresponding attack is possible and leads to the described threat. Note that vulnerabilities are inherently bound to a SUV while attacks are inherently bound to a SUV's environment. Test selection is done using both.

A.2.1.3. Explicit Test Selection Criteria

As we have previously discussed in the introduction, the set of relevant test cases must be sufficiently small in order to be useful for testing activities. The category *Explicit test selection criteria* captures further approaches to cut down the number of relevant test cases defined by the model. They directly describe the basis for test case selection. In our taxonomy, we distinguish the following sub categories:

- **a) Structural Coverage.** Test cases are selected based on some structural criteria of the model. E.g., a test suite is constructed so that every node or every edge is covered.
- **b) Data Coverage.** An approach that uses data coverage to generate test cases is focused on the large data range of a values and specifies which subset of values are considered for the test case generation.

c) Requirements-based Coverage. An approach selects test cases according to requirement-based coverage, if informal requirements are the crucial metrics. This requires that it is possible to explicitly map elements of the model to the given informal requirements.

d) Explicit Test Case Specifications. Papers classified as *explicit test case specification* present an explicit and formal notation of which test cases are considered for testing. E.g., explicit test case specifications can be used to characterize specific paths or scenarios that have to be covered during testing.

e) Random and Stochastic. This criteria directly or indirectly involves probabilities. E.g., usage profiles are an example of a stochastic model.

f) Fault-based. Selection criteria based on faults capture the idea that source code contains faults either introduced intentionally or unintentionally. An approach that uses fault-based selection criteria correlates the test suite to faults.

g) Assessment-based. Test selection can be based on results of an assessment of the artifacts part of a test case. Typically, artifacts are assigned to measured or estimated values gained during an assessment and can be of different kind. For instance, a popular estimated value is risk which *estimates probabilities and consequences of certain threats*. Other metrics for assessment might include costs, priorities, or severities, as well as size, complexity or change measures. Therefore, assessment-based criteria select tests on the basis of these values assigned to artifacts.

h) Relationship-based. Test cases can also be selected based on the difference of two models. Such differences may happen due to different abstraction levels of the models or due to the different versions of the same model (see regression testing).

A.2.2. Evidence Criteria

Evidence criteria capture the applicability and usability of an MBST approach, as well as the actual state in research. Since such evidence data might be missing in a publication, the *evidence criteria* is optional in our taxonomy. Evidence is not something that is specific for MBST. Dias-Neto and Travassos [91] already present evidence categories for MBT. Since the selection of applications and a suitable testing approach is very complex, we extend existing work by presenting orthogonal evidence criteria dedicated to MBST. We consider the following sub categories — *Maturity of evaluated system*, *Evidence measures*, and *Evidence level*.

A.2.2.1. Maturity of Evaluated Systems

An approach can be evaluated using different systems. This category captures the type of the system used for the evaluation and allows the following three sub categories — *Prototype*, *Premature System*, *Productive System*.

a) Prototype. This category represents software in a very early development state and include e.g., sample applications. They have well-known limitations e.g., according to security. Prototypes were developed without industrial pressure.

b) Premature System. A premature system is developed with stakeholders in mind, but is not in a state yet where it performs valuable tasks on a regular basis.

c) Productive System. A productive system performs valuable tasks for stakeholders on a regular basis.

A.2.2.2. Evidence Measures

Evidence measures are metrics for qualitative and quantitative assessments. Since a SUV can be evaluated qualitatively and quantitatively at the same time, this category is multi-select. It contains the following sub categories:

a) Example Application. The approach is applied to a small application in order to demonstrate its feasibility.

b) Effectiveness Measures. Effectiveness metrics are used to decide if the expected results are achieved by the approach. They express the effect of the test suite used during the evaluation. E.g., the effectiveness can be represented as the number of faults found in the SUV.

c) Efficiency Measures. These measures relate test cases, faults, or tested model elements to a notion of required time or cost.

A.2.2.3. Evidence Level

The evidence level expresses at which level test cases are evaluated. Using abstract models as input for a test generation approach, corresponding test cases stays at the same abstraction level. To test actual systems, these Abstract Attack Traces (AATs) need to be made operational. Since not every approach turn AATs operational, we distinguish between abstract and executable test case generation. Since both sub categories do not exclude each other, it is a multi-select criterion.

a) Abstract. Test cases are not executable. Therefore, the effect of the approach on the SUV cannot be measured.

b) Executable. Test cases can be executed against a SUV and therefore, also their effect can be considered. Whether intermediate abstract test cases are generated first is up to the concrete testing approach.

A.3. Systematic Selection and Classification of Publications on MBST

In this section we describe how we systematically collected published work that is later used for our classification. In addition we also discuss threats to validity in terms of publication bias, threats to the identification and classification of publications.

A.3.1. Paper Selection

To provide clear criteria, which papers are considered for this survey, we initially fixed the following requirements:

- The considered model of the MBST approach must be explicit and processable by security tools. Therefore, pure penetration testing and fuzz testing approaches are excluded for this survey.
- Test cases are generated based on these models and are therefore generated systematically. The security test models provide guidance for an effective specification of security test objectives, security test cases, and their automated generation and evaluation.
- We focus on active testing. That means that the considered approaches have to be dynamic and intrusive — stimuli are sent to the SUV and responses of the system are observed [187]. In particular, static analysis and monitoring approaches are excluded since they are not intrusive.
- Finally, approaches that address traditional robustness, safety or trust properties are excluded for this survey study as well.
- The paper must have a length of at least four pages so that we have enough information for the classification. Therefore, e.g., extended abstracts are excluded.
- The paper must be written in English.
- The paper must be primary literature and peer-reviewed.
- Finally, the paper must be published between 1996 and 2013, both including. The year 1996 was chosen, because it is the year of the first MBST paper identified by Dias-Neto and Travassos [91].

The search strategy in order to collect relevant peer-reviewed primary publications is based on an automatic search in the following five digital libraries — IEEE Digital Library, ScienceDirect, Springer Link, ACM Digital Library, and Wiley. These libraries cover the most relevant publications in software and security engineering [76].

In order to measure the quality of the search string for the automatic search, we initially defined a reference database of 76 manually selected relevant papers of all five digital libraries. These papers were selected by three experts in the MBST area. The goal of this reference database is that the result of the automatic search must at least contain all papers of that database. To come up with our search string, we started with the search string defined by Dias-Neto and Travassos [91] and iteratively improved it by adapting it to concepts related to (security) models, security, and testing. The final search string below was used on the fields: *title*, *abstract*, and *keywords*.

```
( "model based" OR automata OR "state machine" OR
  "specification based" OR policy OR policies OR
  "threat model" OR mutation OR risk OR fuzzing )
AND ( security OR vulnerability OR privacy OR cryptographic )
AND ( test OR testing )
```

Performing the automatic search, 5928 papers are returned. Before we classify them, we applied some suitable inclusion and exclusion criteria in three different phases (Figure A.2). Papers that could not be excluded with enough evidence in phase x stayed in the loop and entered the next phase $x+1$.

Phase 1. In phase 1, we excluded papers that do not satisfy the criteria defined in Appendix A.3.1 based on the title of the publication. This reduced the number of papers from 5928 to 660.

Phase 2. In this phase, we excluded irrelevant papers that do not satisfy our criteria based on the abstract. After this phase, 324 papers are remaining.

Phase 3. Finally, the full text of the publication was considered and excluded if necessary. After this phase, 119 papers are remaining that we use as a basis for our classification.

A.3.2. Paper Classification

For the classification, the collected 119 papers were divided among three researchers and classified against the taxonomy introduced in Appendix A.2. Upcoming issues were discussed collaboratively in group sessions. To avoid mistakes during the classification, each paper classification was reviewed by at least one other researcher to apply the 4-eyes principle. The final classification is publicly available[6] as an interactive table, and appended to this thesis in Appendix B.2.

A.3.3. Threat of Validity

Performing a classification always faces several threats of validity. Therefore, we want to discuss the following three threats:

Publication Bias. In our approach only papers that were published are considered. There is the danger that approaches with a negative research outcome are missed. We consider this threat as moderate, since the digital libraries take a broad range of workshops, conferences, and journals into consideration. Nevertheless there is a trade-off between as many publications as possible and reliable information. We decided to exclude gray literature (technical reports, work in progress, unpublished, or not peer-reviewed papers) [142] in favor of quality.

[6] http://qe-informatik.uibk.ac.at/mbst-classification/

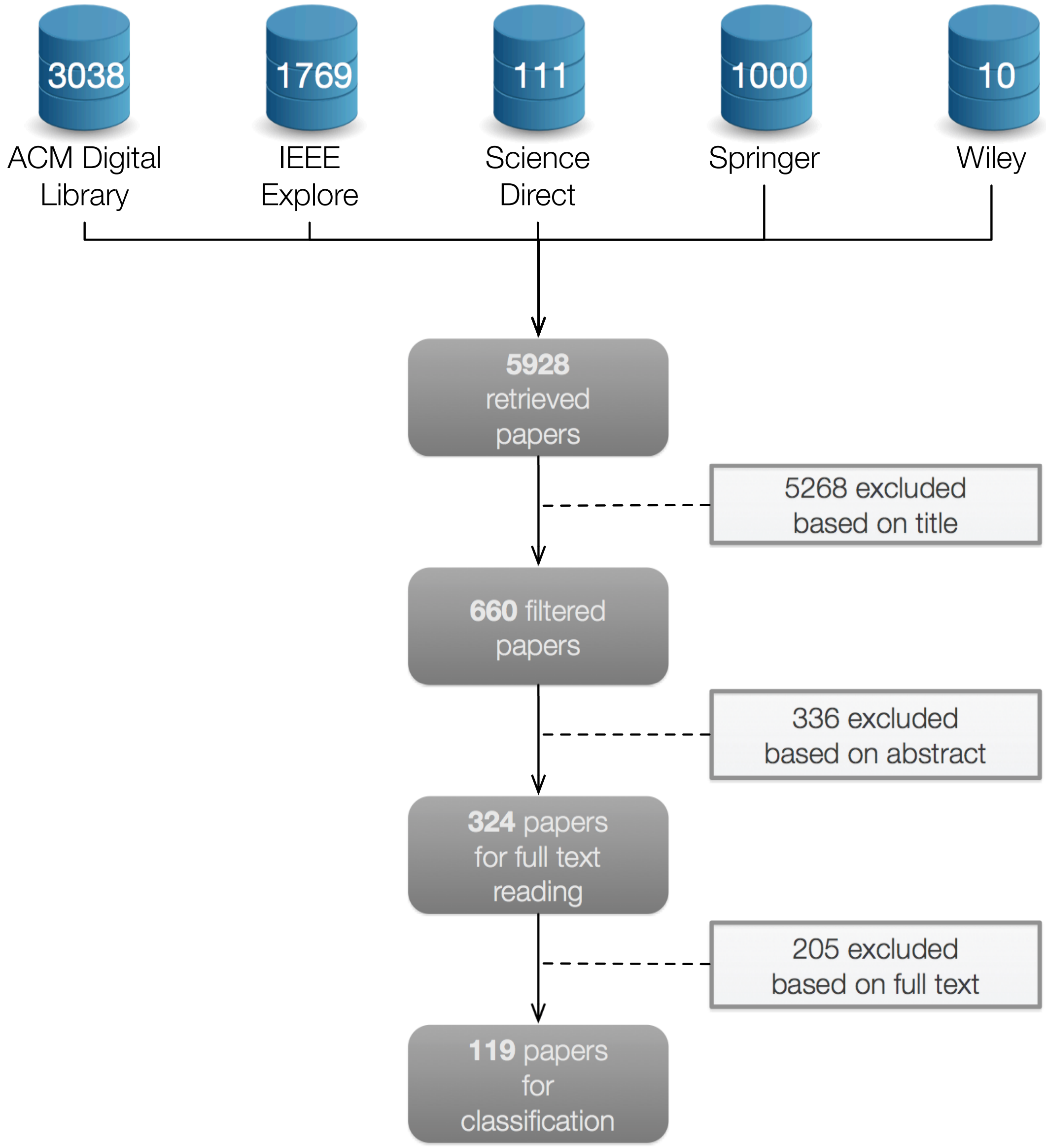

3038
ACM Digital Library
1769
IEEE Explore
111
Science Direct
1000
Springer
10
Wiley
5928 retrieved papers
5268 excluded based on title
660 filtered papers
336 excluded based on abstract
324 papers for full text reading
205 excluded based on full text
119 papers for classification

Figure A.2.: Paper Selection Procedure

Threats to the Identification of Publications. We are aware of the fact that it is impossible to know all existing relevant publications. Nevertheless we iteratively improved the search string according to a reference database. We stopped at a point where further optimizations of the search string lead to a decreasing precision of found papers. Although we followed a comprehensive study, it is not complete. However we consider single missing publications as a moderate threat because the classification aims for validating the taxonomy and characterize the state of the art.

Threats to the Classification of Publications. A high number of primary publications encloses the threat of misclassification. To address this issue, we considered the following counter measures:

- The taxonomy is clearly defined (see Appendix A.2).
- The classification was performed by three researchers all experts in the MBST domain.
- The whole classification process was independently reviewed by at least another author of the survey.

A.4. Results of the MBST Classification

In this section we present the results of the MBST classification, based on four different tables (Tables A.1 to A.4). They consider the following comparison criteria:

- Model of System Security vs. Security Model of Environment.
- Test selection criteria vs. Security Models.
- Maturity of Evaluated System vs. Evidence Level.
- Security Models vs. Evidence Criteria.

In general, we highlight the following two findings:

- As a first result, we observe that the proposed taxonomy allows to classify all 119 papers. Therefore, the successful classification of all papers indicates the adequacy of the taxonomy.
- Since the collection of papers were done in a systematic and comprehensive way, the survey provides a good overview of the state of the art of MBST.

Number of MBST Approaches per Type of Security Model of the Environment. In Table A.1 the different columns show all combinations of Model of System Security whereas the different rows show all combinations of Security models of environments. The table shows that the most common type of *Model of System Security* is *Functionalities of Security mechanisms* (65 papers). Out of these 65 papers, 42 approaches exclusively use functionalities of security mechanisms models and do not combine them with *Security models of environment*. Both findings are not so surprising considering that access control models are very popular in the literature. Comparing ‘Models of System Security' and 'Security models of Environments', the first is much more popular since 84 out of 119 papers do not consider *Security models of Environment* at all. From the 35 papers that do consider *Security models of Environment*, 29 make use of attacker model.

Number of Publications Reporting a Specific Type of Test Selection Criterion. In Table A.2, the different columns show the types of models, whereas different rows represent types of test selection criteria. The table shows that the dominant test selection criteria is *structural criteria* because many models in MBST are graphs, like in MBT. *Structural criteria* are mainly used in combination with *Functionality of Security Models* but also with *Security Properties* and *attack models*. Compared to *structural criteria*, *data coverage* is very rarely used (14 papers). If it is used, the most common combination is with *functionality of Security Models*. Although not as much used as *structural criteria*, *fault-based* and *explicit test case specification* are still criteria often found in state of the art approaches. In contrast, *requirement-based*, *random and stochastic-based*, *assessment-based*, and *relationship-based* criteria are very rarely used and provide room for future work.

Evidence Reported in MBST Publications per Maturity of Evaluated System. In terms of maturity level of the system to be tested, the most used type are prototypes (see Table A.3). 81 papers of whose you use prototypes to evaluate the approach, show the success of the approach based on examples. Compared to that, 25 uses effectiveness measures, and 19 papers use efficiency measures. This observation corresponds with the difficulty of the 3 categories. Usually one starts with some example scenario and only later show more systematically the effectiveness. Once the effectiveness is evaluated, it is put into a specific context which leads to an efficiency evaluation. Furthermore, papers that evaluate the approach based on premature or productive systems also tend to use effectiveness and efficiency measures instead of example evaluations. Whereas 60% of the prototypes are used for evaluation by examples, it is exactly the opposite for pre-mature and productive applications. 60% of them are used for evaluations in terms of effectiveness and efficiency. Differentiating between effectiveness and efficiency, the latter is used at least.

Number of Publications per Type of Security Model. Table A.4 puts types of evidence in relation to types of security models. Further it is a refinement of Table A.3. One can observe that except one single exception [127], evidence is always specified. This underlines the importance of providing an evaluation of the proposed approach. Table A.3 already shows that premature systems are almost never used for an evaluation strategy. This table provides further inside knowledge on that issue. E.g., premature systems are never used in combination with vulnerability models. If a paper makes use of attack models, productive systems are the dominant type of systems used for evaluation. An explanation could be that attack models describe sequences of steps that an attacker has to perform. Complex systems like productive systems are more likely to provide an environment, where such a sequence of steps is possible to execute. The most common types of security models that are evaluated with effectiveness and efficiency measures are *functionalities of security mechanisms*. Considering the ratio between evaluating the approach at abstract vs. executable level, one can observe that this ratio is highest for security property models. The ratio is lowest for vulnerability models. One explanation for this could be that security properties are often abstract and can be evaluated statically, whereas vulnerabilities often require an executable system.

		Model of System Security							
		SecP	V	FSecM	SecP+V	SecP+FSecM	V+FSecM	n.s.	**Sum**
Sec. Model of Env.	T	1	0	3	0	0	0	2	**6**
	A	2	0	1	0	0	0	20	**23**
	T+A	1	1	0	0	1	0	3	**6**
	n.s.	15	7	42	2	9	9	0	**84**
	Sum	**19**	**8**	**46**	**2**	**10**	**9**	**25**	**119**

Table A.1.: Number of MBST Approaches per Type of Security Model of the Environment ('T' = Threat Model; 'A' = Attack Model) and Model of System Security ('SecP' = Security Properties; 'V'= Vulnerabilities; 'FSecM' = Functionality of Security Mechanisms), where '+' represents a combination of different types of security models of the environment or models of system security and 'n.s' stands for 'not specified'

A.5. Discussion of the Results

The goal of this survey is a classification schema for the state of the art in MBST. Based on the presented results in Appendix A.4, we selectively discuss the relationship between 'security properties' and 'vulnerabilities', the use of coverage criteria by existing publications, and the feasibility and ROI of MBT.

a) Security Properties and Vulnerabilities. Comparing research approaches based on security mechanisms, security properties, and vulnerabilities, the first two categories are much more often chosen than the last kind of approach. At one side, this sounds surprising since the goal of security testing is finding potential faults — and faults are very much correlated to vulnerabilities. At the other side, it turns out that fault-based approaches are very often performed manually and are therefore not based on an explicit model. There is no obvious reason for this and one can only speculate. It seems that vulnerabilities are hard to formally specify in an explicit model. Comparing security mechanisms vs. security properties, more published approaches focus on the first one. At one side, this seems strange, since from a theoretical point of view, the difference between testing an access control mechanism and an authentication property is very hard. Both concepts define a set of traces that are intersected with the set of traces defined by the SUV. At the other side, one can speculate that performing function testing a security mechanism is easier than testing an abstract security

		Model of System Security					
		SecP	V	FSecM	T	A	**Sum**
Test Selection Criteria	SC	13	7	38	7	16	**81**
	DC	0	1	10	2	1	**14**
	RC	3	0	4	2	0	**9**
	TCS	9	1	9	1	6	**26**
	RS	1	1	5	0	0	**7**
	FB	6	9	12	3	5	**35**
	AB	0	1	0	0	3	**4**
	RB	2	2	2	0	1	**7**

Table A.2.: Number of Publications Reporting a Specific Type of Test Selection Criterion (‘SC’ = Structural Coverage; ‘DC’ = Data Coverage; ‘RC’ = Requirements Coverage; ‘TCS’= Explicit Test Case Specifications; ‘RS’ = Random and Stochastic; ‘FB’ = Fault-Based; ‘AB’ = Assessment-Based; ‘RB’ = Relationship-Based) per Type of Security Model (‘SecP’ = Security Properties; ‘V’= Vulnerabilities; ‘FSecM’ = Functionality of Security Mechanisms; ‘T’ = Threat Model; ‘A’ = Attack Model)

property. This origins in the fact that a security property affects every trace defined by the SUV, whereas the security mechanism is restricted to one dedicated security component. Nevertheless, it is not always clear that a security property enforced by a security mechanism holds system-wide. To bring some lights into the approaches that deal with security properties, we performed a further analysis on the 31 papers that are classified as *property-based*. It turns out that the majority of those paper focus on confidentiality rather than integrity or availability properties. This opens possibilities for further research since a vulnerability in general can be exploited to violate a set of different security properties.

As a summary, we see room for future research that (1) explicitly addresses vulnerabilities, (2) further investigates the relationship between functional testing a security mechanism and guaranteeing system-wide properties, and (3) directly addresses system-wide security properties.

b) Coverage Criteria. In section Appendix A.4 we observed that coverage criteria are very popular as test selection criteria. This is very likely based on the fact that generating test cases based on coverage criteria can be highly automated. Nevertheless, there is an ongoing discussion whether such generated test cases have a good potential fault detection rate [178, 224, 115, 132, 161, 147, 120, 95]. Usually, coverage criteria are defined independent of underlying fault-models, the success of test case strategies based on coverage criteria depends on the distribution of the faults in the system. In general, this distribution is not known a priori. At the other side, coverage criteria can also be used without an explicit model in order to identify vulnerable parts of an

Maturity of Evaluated System			Evidence Measures			
			Abs	Exec	Abs + Exec	**Sum**
	Prototype					
		Ex	36	22	23	**81**
		Effe	3	15	7	**25**
		Effi	3	9	7	**19**
		Sum	**42**	**46**	**37**	**125**
	Premature					
		Ex	1	0	1	**2**
		Effe	0	2	1	**3**
		Sum	**1**	**2**	**2**	**5**
	Production					
		Ex	3	3	1	**7**
		Effe	2	7	0	**9**
		Effi	1	0	0	**1**
		Sum	**6**	**10**	**1**	**17**

Table A.3.: Evidence Reported in MBST Publications per Maturity of Evaluated System (‘Premature’ = Premature System; ‘Productive’ = Productive System), Evidence Measures (‘Ex’ = Example Application; ‘Effe’ = Effectiveness Measures; ‘Effi’ = Efficiency Measures) and Evidence Level (‘Abs’ = Abstract; ‘Exec’ = Executable), where ‘+’ represents a combination of evidence levels

	Prot	Pre	Prod	Ex	Effe	Effi	Abs	Exec	Abs+Exec
SecP	27	3	1	25	5	5	17	5	9
V	17	0	2	13	9	4	4	11	4
FSecM	56	3	6	49	19	13	24	24	16
T	10	1	1	11	3	3	5	4	3
A	21	2	5	25	4	2	10	10	8
Sum	**131**	**9**	**15**	**123**	**40**	**27**	**60**	**54**	**40**

Table A.4.: Number of Publications per Type of Security Model ('SecP' = Security Properties; 'V'= Vulnerabilities; 'FSecM' = Functionality of Security Mechanisms; 'T' = Threat Model; 'A' = Attack Model) Reporting Evidence per Maturity of Evaluated System ('Prot' = Prototype; 'Pre' = Premature System; 'Prod' = Productive System), Evidence Measures ('Ex' = Example Application; 'Effe' = Effectiveness Measures; 'Effi' = Efficiency Measures) and Evidence Level ('Abs' = Abstract; 'Exec' = Executable), where '+' represents a combination of evidence levels

application (e.g., fuzz testing [112]). Therefore, future research should contribute to a better understanding which kind of coverage criteria target an effective and efficient security testing in the context of MBST.

c) Feasibility and ROI of Model-Based Testing. The literature only provides rough calculations about the ROI of model based testing [113, 160]. This origins from the fact that the ROI in this context has first not been sufficiently understood so far, and second, is very hard to measure. This corresponds with the observation made in the paragraph 'Evidence Reported in MBST Publications per Maturity of Evaluated System' in Appendix A.4, that efficiency evaluation is rarely considered in publications. In the context of re-usability, building and managing models is more beneficial than building and managing test suites directly. This benefit has a practical price:

- Finding the correct abstraction level.
- Modeling knowledge.
- Efficiency of test case generation out of models.
- Debugging help for models.
- Synchronization of models with source code of applications.
- Embedding the modeling activities into the overall software development process.
- Developing models must be cost efficient. Otherwise, the models will never exist. This holds for MBT and MBST.
- More generally, since both MBT and MBST require models, all concerns for MBT are also valid concerns for MBST.

Finally, future research has to address the issue which kind of models are beneficial for MBST. In particular, it has to elaborate whether general security models can be build that can be re-used in different contexts.

A.6. Conclusion

In the first part of this Ph.D thesis, a taxonomy for MBST and a classification of systematically collected state of the art MBST publications was conducted. The taxonomy is based on a comprehensive analysis of existing classification schemes for both MBT and security testing. The extension that this taxonomy provides is in terms of two main categories — *filter criteria* and *evidence criteria*. Filter criteria capture the idea how test cases are selected — either by different types of security model and/or explicit test selection criteria. They define a finite subset of all the traces of a SUV that are interesting in the context of security testing. Evidence criteria capture the idea how the proposed security testing approach is evaluated in terms of the maturity of the evaluated system, the used evidence measures, and the evidence level. Using a systematic search in five digital libraries, we finally collected 119 MBST relevant papers and classified them according to our proposed taxonomy. The classification provides a state of the art overview of MBST approaches between 1996 and 2013. Finally, we highlighted some interesting observations based on our classification and discussed potential future research directions. Since MBST is an active research domain, our taxonomy and classification helps to clarify key issues in this domain.

B. MBST Classification

B.1. Abbreviations for Filter and Evidence Criteria

	Abbreviation	Meaning
Model of System Security	MSSec	
	SecP	Security Properties
	V	Vulnerabilities
	FsecM	Functionality of Security Mechanisms
	n.s.	not specified
Security Model of Environment	SecME	
	A	Attack Model
	T	Threat Model
	n.s.	not specified
Test Selection Criteria	TSC	
	SC	Structural Coverage
	DC	Data Coverage
	RC	Requirements Coverage
	TCS	Explicit Test Case Specifications
	RS	Random and Stochastic
	FB	Fault-Based
	AB	Assessment-Based
	RB	Relationship-Based
	n.s.	not specified

Table B.1.: Abbreviations for Filter Criteria

B.2. MBST Classification

Table B.3 shows the result of the classification as described in Appendix A. The table is sorted by author names and applying the abbreviations from Table B.1 and Table B.2 of Appendix B.1.

	Abbreviation	Meaning
Maturity of Evaluated System	MES	
	Prot	Prototype
	Pre	Premature System
	Prod	Productive System
	n.s.	not specified
Evidence Measures	EM	
	Ex	Example Application
	Effe	Effectiveness Measures
	Effi	Efficiency Measures
	n.s.	not specified
Evidence Level	EL	
	Abs	Abstract
	Exec	Executable
	n.s.	not specified

Table B.2.: Abbreviations for Evidence Criteria

	Filter Criteria			**Evidence Criteria**		
Paper	**MSSec**	**SecME**	**TSC**	**MES**	**EM**	**EL**
Abassi and Fatmi [50]	SecP+FSecM	n.s.	TCS	Prot	Ex	Abs
Abbassi and El Fatmi [51]	SecP	n.s.	TCS	Prot	Ex	Abs
Al-Azzani and Bahsoon [52]	n.s.	A	TCS	Prot	Ex	Abs
Al-Shaer et al. [53]	FSecM	n.s.	RS	Prot	Effe+Effi	Abs+Exec
Allen et al. [54]	V	n.s.	SC	Prot	Ex+Effe	Exec
Antunes and Neves [58]	FSecM	n.s.	RS	Prot	Ex	Abs+Exec
Armando et al. [60]	SecP+FSecM	n.s.	SC	Pre	Effe	Exec
Bartel et al. [63]	V	n.s.	AB	Prot	Ex	Abs+Exec
Belhaouari et al. [65]	FSecM	n.s.	SC+DC+RS	Prot	Ex+Effi	Exec
Bertolino et al. [67]	FSecM	n.s.	DC	Prot	Effe+Effi	Exec
Bertolino et al. [68]	FSecM	T	FB	Prot	Ex+Effe+Effi	Abs+Exec
Beyer et al. [69]	SecP	n.s.	SC	Prot	Effe+Effi	Abs+Exec
Blome et al. [70]	n.s.	A	TCS	Prot	Ex	Exec
Bortolozzo et al. [71]	FSecM	n.s.	FB	Prod	Effe	Exec
Botella et al. [72]	FSecM	n.s.	SC+RC	Prot	Ex+Effi	Abs+Exec
Bozic and Wotawa [74]	n.s.	A	SC	Prot	Ex	Exec
Bracher and Krishnan [75]	SecP	n.s.	RC+TCS	Prot	Ex	Abs

Brucker et al. [79]	FSecM	n.s.	SC+DC	Prot	Effi	Abs
Brucker and Wolff [77]	FSecM	n.s.	SC+DC	Prot	Effi	Abs
Brucker et al. [78]	FSecM	n.s.	SC+DC	Prod	Ex	Abs
Brucker et al. [80]	FSecM	n.s.	SC+DC	Prot	Ex	Abs
Büchler et al. [82]	SecP	n.s.	FB	Prot	Ex	Abs
Büchler et al. [83]	SecP	n.s.	FB	Prot	Ex+Effe+Effi	Abs+Exec
Chen et al. [84]	SecP+FSecM	n.s.	SC	Prot	Ex	Abs+Exec
Dadeau et al. [86]	FSecM	n.s.	FB	Prot	Ex	Abs+Exec
Darmaillacq et al. [87]	FSecM	n.s.	SC	Prot	Ex	Abs
Darmaillacq et al. [88]	FSecM	n.s.	TCS	Prot	Ex	Abs+Exec
El Kateb et al. [96]	SecP+FSecM	n.s.	TCS	Prot	Ex	Abs+Exec
El Maarabani et al. [97]	FSecM	n.s.	SC	Prot	Ex	Exec
Elrakaiby et al. [98]	FSecM	n.s.	FB	Prot	Ex+Effe	n.s.
Falcone et al. [99]	FSecM	n.s.	SC	Prot	Ex	Exec
Faniyi et al. [100]	n.s.	A	TCS	Prot	Ex	Abs
Felderer et al. [103]	SecP	n.s.	SC+RB	Prot	Ex	Abs
Fourneret et al. [105]	SecP+FSecM	n.s.	SC	Prot	Ex	Abs
Gilliam et al. [110]	SecP+V	n.s.	n.s.	Prot	Ex	Abs
Hanna et al. [116]	SecP	n.s.	SC	Prot	Ex	Abs
Hanna et al. [117]	V	n.s.	FB+DC	Prot	Effi	Exec
He et al. [119]	n.s.	A+T	SC	Prot	Ex	Abs
Yu et al. [238]	FSecM	n.s.	SC	Prot	Ex	Abs
Hsu et al. [123]	FSecM	T	SC+DC	Prot	Ex	Abs+Exec
Hu and Ahn [124]	SecP+FSecM	n.s.	TCS	Prot	Ex	Abs
Hu et al. [125]	FSecM	n.s.	SC	Prot	Ex	Abs
Hu et al. [126]	SecP+FSecM	n.s.	SC	Prot	Ex	Abs+Exec
Huang and Wen [127]	n.s.	A	SC	n.s.	n.s.	n.s.
Hwang et al. [128]	FSecM	n.s.	SC+RS	Prot	Effe+Effi	Exec
Hwang et al. [129]	SecP	n.s.	SC	Prot	Ex	Abs+Exec
Hwang et al. [130]	FSecM	n.s.	SC+Random	Prot	Effe+Effi	Exec
Julliand et al. [135]	SecP+FSecM	n.s.	TCS	Prot	Ex	Abs
Julliand et al. [136]	FSecM	n.s.	SC+DC	Prod	Effe	Abs
Jürjens [137]	SecP	T	SC+RC	Prot	Ex	Abs
Jurjens and Wimmel [138]	FSecM	T	RC	Prot	Ex	Abs
Jürjens and Wimmel [139]	FSecM	n.s.	TCS	Prot	Ex	Abs
Kam and Dean [140]	n.s.	A	FB	Pre	Effe	Exec
Le Traon et al. [143]	V+FSecM	n.s.	SC+FB	Prot	Effe	Exec
Traon et al. [206]	V+FSecM	n.s.	SC+FB	Prot	Ex+Effe	Exec
Lebeau et al. [144]	FSecM	n.s.	SC+RC	Prot	Ex	Abs
Li et al. [145]	FSecM	n.s.	SC	Prot	Ex	Abs
Mallouli and Cavalli [148]	SecP	n.s.	TCS	Prot	Effe	Abs

Mallouli et al. [149]	SecP	n.s.	FB	Prot	Effe+Effi	Exec
Mallouli et al. [150]	SecP	n.s.	FB	Prot	Effe+Effi	Exec
Mammar et al. [151]	SecP	n.s.	SC+RC	Prot	Ex	Exec
Marback et al. [153]	n.s.	T	SC+DC	Prod	Ex+Effe	Exec
Martin and Xie [154]	V+FSecM	n.s.	SC	Prot	Effe+Effi	Exec
Martin et al. [155]	SecP+FSecM	n.s.	FB	Prot	Ex+Effe+Effi	Abs
Masood et al. [156]	FSecM	A	SC	Prot	Effe+Effi	Abs+Exec
Masson et al. [157]	SecP+V	n.s.	TCS	Prot	Ex	Abs
Morais et al. [162]	n.s.	A	SC	Prot	Ex	Abs+Exec
Morais et al. [163]	n.s.	A+T	SC	Prot	Ex+Effi	Exec
Mouelhi et al. [164]	V+FSecM	n.s.	FB	Prot	Ex	Exec
Mouelhi et al. [165]	FSecM	n.s.	FB	Prot	Ex	Exec
Mouelhi et al. [166]	V+FSecM	n.s.	SC+FB	Prot	Effe	Exec
Mouelhi et al. [167]	V+FSecM	n.s.	RB	Prot	Ex+Effe	Exec
Nguyen et al. [169]	V+FSecM	n.s.	FB	Prot	Ex	Exec
Noseevich and Petukhov [170]	SecP	n.s.	SC	Prot	Ex	Abs+Exec
Pari-Salas and Krishnan [171]	FSecM	n.s.	SC	Prot	Ex	Abs
Pretschner et al. [176]	FSecM	n.s.	DC	Prot	Ex+Effe	Exec
Rekhis et al. [180]	FSecM	n.s.	TCS	Prot	Ex	Abs
Rosenzweig et al. [181]	FSecM	n.s.	SC	Prot	Ex	Abs
Saidane and Guelfi [182]	SecP	A+T	SC	Prot	Ex	Exec
Salas et al. [183]	SecP	A	TCS	Prot	Ex	Abs
Salva and Zafimiharisoa [184]	V+FSecM	n.s.	RB	Prot	Ex+Effe+Effi	Abs+Exec
Savary et al. [185]	V	n.s.	FB	Prod	Effe+Effi	Abs
Schneider et al. [189]	n.s.	A	FB	Prot	Ex	Exec
Schneider et al. [188]	n.s.	A	SC	Prod	Ex	Abs+Exec
Senn et al. [191]	FSecM	n.s.	SC	Prod	Ex	Exec
Shahriar and Zulkernine [193]	n.s.	A	SC	Prod	Effe	Exec
Shu and Lee [195]	SecP	n.s.	RB	Pre	Effe	Abs+Exec
Shu and Lee [194]	SecP	A	SC	Prod	Ex	Abs
Shu et al. [196]	V	n.s.	SC	Prod	Effe	Exec
Singh et al. [197]	SecP	n.s.	RS	Prot	Ex	Abs
Stepien et al. [201]	n.s.	A	RB	Prot	Ex	Abs+Exec
Tang et al. [204]	V	n.s.	RS	Prot	Ex	Exec
Traore and Aredo [207]	FSecM	n.s.	RC	Prot	Ex	Exec
Tuglular and Belli [208]	V+FSecM	n.s.	SC	Prot	Ex	Abs+Exec
Tuglular and Gercek [209]	FSecM	n.s.	SC	Prot	Ex	Abs+Exec
Tuglular and Gercek [210]	FSecM	n.s.	SC	Prod	Effe	Exec
Tuglular et al. [211]	FSecM	n.s.	SC	Prot	Ex	Exec
Turcotte et al. [212]	V	n.s.	FB	Prot	Ex	Abs+Exec
Wang et al. [219]	n.s.	A+T	SC+TCS	Prot	Ex	Abs

Wang et al. [220]	FSecM	n.s.	SC	Prot	Ex+Effe	Exec
Weber et al. [221]	FSecM	n.s.	SC	Prot	Ex+Effe	Abs
Wei et al. [222]	n.s.	A	SC	Prot	Ex	Abs+Exec
Whittle et al. [225]	n.s.	A	SC	Prot	Ex	Abs
Wimmel and Jürjens [227]	V	A+T	FB	Prot	Ex	Abs
Xu et al. [232]	FSecM	n.s.	SC	Prot	Ex+Effe	Abs+Exec
Xu et al. [231]	n.s.	T	SC	Prot	Effe	Exec
Xu et al. [230]	FSecM	n.s.	SC	Prot	Effe	Exec
Yan and Dan [233]	FSecM	n.s.	SC	Prot	Ex	Abs
Yan et al. [234]	FSecM	n.s.	TCS	Prot	Ex	Abs
Yang et al. [236]	FSecM	n.s.	TCS	Pre	Ex	Abs
Yang et al. [237]	n.s.	A	DC	Prot	Ex	Exec
Yang et al. [235]	n.s.	A	SC	Prod	Ex+Effe	Exec
Yu et al. [239]	FSecM	n.s.	DC	Prod	Effe	Exec
Zech [241]	n.s.	A	SC+AB	Prot	Ex	Abs+Exec
Zech et al. [243]	n.s.	A	AB	Prot	Ex	Abs+Exec
Zech et al. [242]	n.s.	A	SC+AB	Prot	Ex	Abs
Zhang et al. [244]	n.s.	A	FB	Prot	Ex	Exec
Zhou et al. [245]	SecP+FSecM	A+T	FB	Pre	Ex	Abs+Exec
Zulkernine et al. [246]	n.s.	A	TCS	Prod	Ex	Abs

Table B.3.: Classification of Selected Model-Based Security Testing Publications

C. Syntax Definitions

C.1. Grammar Web Application Abstract Language (WAAL) Mapping

Listing C.1: WAAL Mapping Grammar

```
grammar spacite.Waalmapping with org.eclipse.xtext.common.Terminals

generate waalmapping "http://www.Waalmapping.spacite"

WaalmappingModel:
  simulatedAgents=SimulatedAgents
  agentConfigurations=AgentConfigurationsSection
  (functions=FunctionSection)?
  (messagestoactions=MessagesActionSection)?
;

AgentConfigurationsSection:
  {AgentConfigurationsSection}
  name='AGENTS_CONFIGURATIONS' open='{'
  agents+=Agent*
  close='}'
;

Agent:
  'Agent:' agentName=STRING '{'
    ('httpBasicAuthentication_Username:' httpBasicUsername=STRING
     'httpBasicAuthentication_Password:' httpBasicPassword=STRING)?
    'BrowserIP:' browserIP=STRING 'BrowserPort:' browserPort=INT
    'Sessions:' sessions=Sessions
    'Driver:' driver=('REMOTEWEBDRIVER' | 'RC' )
    'Platform:' platform=('ANDROID' | 'ANY' | 'LINUX' | 'MAC' | 'UNIX' |
                          'VISTA' | 'WIN8' | 'WIN8_1' | 'WIDNOWS' | 'XP')
    'Browser:' browser=('firefox' | 'chrome')
    'Browser Version:' browserVersion=BrowserVersion
    ('Init:' '{'
      initActions+=(AbstractGeneratingAction | AbstractVerifingAction)*
     '}'
    )?
  '}'
;

Sessions:
  '{' names+=Session ( ',' names+=Session )* '}'
;
Session:
  '{' names+=EndPoint ( ',' names+=EndPoint )* '}'
;
```

```

EndPoint:
  name=STRING
;

BrowserVersion:
  version=('ANY' | STRING)
;

SimulatedAgents:
  'SIMULATED_AGENTS' open='{' listOfAgents=ListOfAgents close='}';

ListOfAgents:
  agents+=STRING (',' agents+=STRING)*
;

MessagesActionSection:
  'MESSAGES_TO_ACTIONS' '{'
    (Messages2actionsMappings+=Messages2ActionsMapping)+
  '}'
;

Messages2ActionsMapping:
  'Message@Actions' '{'
    abstractPart=AbstractPart
    waalPart=WaalPart
  '}'
;

AbstractPart:
  'abstract' '{'
    abstractMessage=STRING
  '}'
;

WaalPart:
  {WaalPart}
  'waal(generation)' '{'
    waalactionsGeneration+=(AbstractGeneratingAction | AbstractVerifingAction)*
  '}'
  'waal(verification)' '{'
    waalactionsVerification+=(AbstractGeneratingAction | AbstractVerifingAction)*
  '}'
;

FunctionSection:
  'MAPPINGS' open='{' (listOfFunctions+=Function)* close='}'
;

Function:
  name=ID ':' (valuePair+=ValuePair)+
;

ValuePair:
  ('abstract:' input=STRING)? 'concrete:' output=STRING
;
```

```


// WAAL actions:

AbstractGeneratingAction:
  'Generate::'
  'Element:' element=BrowserElement
  'Action:' action=GA
  ('Additional action:' additionalAction=AdditionalAction)?
  ('Postcondition:' postcondition=Postcondition)?
;

AbstractVerifingAction:
  'Verify::'
  'Element:' element=BrowserElement
  'Condition:' condition=Condition
;

GA:
  ClickAction | TypeAction | SelectAction | SwitchTo | GoToURL |
  WaitForElement | FollowAction | SelectFile
;

ClickAction:
  name='Click' (implicit+=ClickActionImplicit)*
;

ClickActionImplicit:
  'ImplicitElement:' implicitElement=STRING
  'ImplicitAttribute:' implicitAttribute=STRING
  'ImplicitValue:' implicitValue=InputValue
;

TypeAction:
  name='Type' value=InputValue ('DefaultValue' defaultValue=STRING)?
;

SelectAction:
  name='Select' value=InputValue
;

SwitchTo:
  name='SwitchTo'
;

GoToURL:
  name='GoToURL' url=STRING
;

WaitForElement:
  name='WaitFor'
;

FollowAction:
  name='Follow'
;
```

```

SelectFile:
  name='SelectFile' value=STRING
;

AdditionalAction:
  AcknowledgeAlert | CancelAlert
;

AcknowledgeAlert:
  {AcknowledgeAlert}
  'AcknowledgeAlert'
;

CancelAlert:
  {CancelAlert}
  'Cancel Alert'
;

Postcondition:
  'Element:' element=BrowserElement
  'Condition:' condition=Condition
;

Condition:
  IsDisplayed | IsNotDisplayed | IsPresent | IsNotPresent | IsContained |
  IsNotContained | ValueHasChanged | PageIsLoaded | VerifyMaliciousEffect
;

IsDisplayed:
  {IsDisplayed}
  'ElementIsDisplayed'
;

IsNotDisplayed:
  {IsDisplayed}
  'ElementIsNotDisplayed'
;

IsPresent:
  {IsPresent}
  'ElementIsPresent'
;

IsNotPresent:
  {IsNotPresent}
  'ElementIsNotPresent'
;

IsContained:
  'TextIsContained:' value=STRING
;

IsNotContained:
  'TextIsNotContained:' value=STRING
;
```

```

ValueHasChanged:
  {ValueHasChanged}
  'ValueHasChanged'
;

PageIsLoaded:
  {PageIsLoaded}
  'PageIsLoaded'
;

/*
 * ELEMENTS
 */
BrowserElement:
  BrowserWindow | WebpageContent | PageElement
;

BrowserWindow:
  locator=ByBrowser
;

WebpageContent:
  locator=ByWebpageContent
;

PageElement:
  PageElementButton | PageElementInputButton | PageElementTextfield |
  PageElementDropdown | PageElementLink | PageElementSelectableList |
  Div | Span | PageElementImage | PageElementTextArea | GenericElement
;

PageElementButton:
  'Button:' locator=Locator
;

PageElementInputButton:
  'InputButton:' locator=Locator
;

PageElementTextfield:
  'Textfield:' locator=Locator
;


PageElementDropdown:
  'Dropdown:' locator=Locator
;

PageElementLink:
  'Link:' locator=Locator
;

PageElementSelectableList:
  'SelectableList:' locator=Locator
;
```

```

Div:
  'DIV:' locator=Locator
;

Span:
  'SPAN:' locator=Locator
;

GenericElement:
  'GenericElement:' locator=Locator
;

PageElementImage:
  'Image:' locator=Locator
;

PageElementTextArea:
  'TextArea:' locator=Locator
;

/*
 * LOCATOR
 */
Locator:
  ByID | ByXPath | ByCSS | ByLinkText | ByPartialLinkText | ByName |
  ByVisibleText | ByAttribute | ByWebpageContent | ByBrowser
;

ByID:
  'ByID' '(' arg=STRING ')'
;

ByXPath:
  'ByXPath' '(' arg=STRING ')'
;

ByCSS:
  'ByCSS' '(' arg=STRING ')'
;

ByLinkText:
  'ByLinkText' '(' arg=STRING ')'
;

ByPartialLinkText:
  'ByPartialLinkText' '(' arg=STRING ')'
;

ByVisibleText:
  'ByVisibleText' '(' arg=STRING ')'
;

ByName:
  'ByName' '(' arg=STRING ')'
;
```

```

ByAttribute:
  'ByAttribute' '(' 'Attribute:' att=STRING ',' 'Value:' value=STRING ')'
;

ByWebpageContent:
  arg = 'WebpageContent'
;

ByBrowser:
  arg = 'Browser'
;

InputValue:
  Parameter | Injection | StrValue | VariableValue
;

Parameter:
  '$' value=INT
;

Injection:
  'INJECT' '(' type=Injectiontype (',' position=IntValue)? ')'
;

StrValue:
  value=STRING
;

IntValue:
  value=INT
;

VariableValue:
  value=ID
;

Injectiontype:
  type=('value' | 'HTML'| 'JAVASCRIPT' | 'SQL')
;

VerifyMaliciousEffect:
  action='VERIFY_MALICIOUS_EFFECT'
;

terminal SL_COMMENT:
  '//' !('\n'|'\r')* ('\r'? '\n')?
;
```

D. WAAL Mappings

D.1. WAAL Mapping for WebGoat

Listing D.1: WAAL Mapping for WebGoat

```
SIMULATED_AGENTS { "tom", "jerry" }

AGENTS_CONFIGURATIONS {
  Agent: "tom" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
    }
  Agent: "jerry" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
    }
}

MAPPINGS {
  username:
  abstract:" tom " concrete:"\"Tom Cat (employee)\""
  abstract:" jerry " concrete:"\"Jerry Mouse (hr)\""

  pwd:
  abstract:" tom " concrete:"\"tom\""
  abstract:" jerry " concrete:"\"jerry\""
}

MESSAGES_TO_ACTIONS {

  Message@Actions {
    abstract {
      // profile(username_t, text)
    "profile"
  }
  waal(generation) {
    // waal actions to generate the above message
  }
  waal(verification) {
    // waal actions to verify the above message
    Verify::
    Element: WebpageContent
    Condition: TextIsContained: "Credit Card"
  }
}
Message@Actions {
  abstract {
    // editProf(username_t, text)
    "editProf"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element:InputButton:ByAttribute(Attribute:"value",Value:"EditProfile")
    Action:Click

    Generate::
    Element:Textfield:ByName("address1")
    Action:Type $5 DefaultValue "Default"

    Generate::
    Element:InputButton:ByAttribute(Attribute:"value",Value:"UpdateProfile")
    Action:Click
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // deleteProf(username_t)
    "deleteProf"
  }
  waal(generation) {
    // waal actions to generate the above message
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // viewProf(username_t)
    "viewProf"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element:SelectableList:ByName("employee_id")
```

```
      Action: Select $4

      Generate::
      Element: InputButton: ByAttribute(Attribute:"value",Value:"ViewProfile")
      Action:Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // login(username_t, password_t)
      "login"
    }
    waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element: Browser
      Action: GoToURL "http://guest:guest@172.16.1.12/WebGoat/attack"

      Generate::
      Element: InputButton:
        ByAttribute(Attribute:"value", Value: "Start WebGoat")
      Action: Click

      Generate::
      Element: Link: ByVisibleText ("Cross-Site Scripting (XSS)")
      Action: Follow

      Generate::
      Element: Link: ByVisibleText ("Stage 1: Stored XSS")
      Action: Follow

      Generate::
      Element: Dropdown: ByName("employee_id")
      Action: Select $4

      Generate::
      Element:Textfield:ByName("password")
      Action:Type $5 DefaultValue "Default"

      Generate::
      Element:InputButton:ByAttribute(Attribute:"value",Value:"Login")
      Action:Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // retE2L
      "retE2L"
    }
    waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element:InputButton:ByAttribute(Attribute:"value",Value:"ListStaff")
      Action:Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // listStaff(username_t)
      "listStaff"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element: WebpageContent
      Condition: TextIsContained: "Staff Listing Page"
    }
  }

  Message@Actions {
    abstract {
      // create
      "VERIFY"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element:Browser
      Condition:VERIFY_MALICIOUS_EFFECT
    }
  }
}
```

D.2. WAAL Mapping for Wackopicko

Listing D.2: WAAL Mapping for Wackopicko

```
SIMULATED_AGENTS { "i", "client", "clientDishonest" }

AGENTS_CONFIGURATIONS {

  Agent: "client" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
    }

    Agent: "i" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
    }
}

MAPPINGS {

username_client:
abstract: "" concrete: "\"abc\""
firstname_client:
abstract: "" concrete: "\"abc\""
lastname_client:
abstract: "" concrete: "\"abc\""
password_client:
abstract: "" concrete: "\"abc\""

Username_upload:
abstract: " 162 " concrete: "\"abc\""
Tag_upload:
abstract: " 162 " concrete: "\"tag\""
Filename_upload:
abstract: " 162 " concrete: "\"test.png\""
Title_upload:
abstract: " 162 " concrete: "\"title\""

Username_Server:
abstract: " 152 " concrete: "\"abc\""
```

```
Lastname_Server:
abstract: " 152 " concrete: "\"abc\""
}


MESSAGES_TO_ACTIONS {

  Message@Actions {
    abstract {
      // welcome(username_t)
      "welcome"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element: WebpageContent
      Condition: TextIsContained: "Hello"
    }
  }
  Message@Actions {
    abstract {
      // preview(comment_t)
      "preview"
    }
    waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element:TextArea:ByID("comment-box")
      Action:Type $4 DefaultValue "Default"

      Generate::
      Element:InputButton:ByAttribute(Attribute:"value",Value:"Preview")
      Action:Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // askForComment(username_t, title_t)
      "askForComment"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element: WebpageContent
      Condition: TextIsContained: "Add your comment"
    }
  }
```

```
Message@Actions {
  abstract {
    // recent
    "recent"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element: Link: ByXPath("html/body/div[1]/div[2]/div[1]/ul/li[3]/a/span")
    Action: Follow

    Generate::
    Element:Image:ByXPath("(.//img)[1]")
    Action:Click
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // register(agent, username_t, firstname_t, lastname_t, password_t)
    "register"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element: Browser
    Action: GoToURL "http://172.16.1.12/WackoPicko/"

    Generate::
    Element: Link: ByVisibleText ("Create an account")
    Action: Follow

    Generate::
    Element:Textfield:ByName("username")
    Action:Type $5 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("firstname")
    Action:Type $6 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("lastname")
    Action:Type $7 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("password")
    Action:Type $8 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("againpass")
    Action:Type $8 DefaultValue "Default"

    Generate::
    Element:InputButton:ByAttribute(Attribute:"value",Value:"Create Account!")
    Action:Click
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // upload(username_t, tag_t, filename_t, title_t, price_t)
    "upload"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element: Link: ByXPath("html/body/div/div[2]/div[1]/ul/li[2]/a/span")
    Action: Follow

    Generate::
    Element:Textfield:ByName("tag")
    Action:Type $4 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("name")
    Action:Type $5 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("title")
    Action:Type $6 DefaultValue "Default"

    Generate::
    Element:Textfield:ByName("price")
    Action:Type $7 DefaultValue "Default"

    Generate::
    Element:InputButton:ByName("pic")
    Action:SelectFile "/tmp/test.png"

    Generate::
    Element:InputButton:ByAttribute(Attribute:"value",Value:"Upload File")
    Action:Click
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // show_preview(username_t, title_t, comment_t)
    "show_preview"
  }
  waal(generation) {
    // waal actions to generate the above message
  }
  waal(verification) {
    // waal actions to verify the above message
    Verify::
```

```
    Element: WebpageContent
    Condition: TextIsContained: "A Preview of what your comment"
  }
}
Message@Actions {
  abstract {
    // recent_response(username_t, tag_t, filename_t, title_t, comment_t)
    "recent_response"
  }
  waal(generation) {
    // waal actions to generate the above message
  }
  waal(verification) {
    // waal actions to verify the above message
    Verify::
    Element: WebpageContent
    Condition: TextIsContained: "Recently uploaded pictures"

  }
}
Message@Actions {
  abstract {
    // similarName
    "similarName"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element: Link: ByXPath("html/body/div[1]/div[4]/ul/li[1]/a")
    Action: Follow
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // similarName_response(username_t)
    "similarName_response"
  }
  waal(generation) {
    // waal actions to generate the above message
  }
  waal(verification) {
    // waal actions to verify the above message
    Verify::
    Element: WebpageContent
    Condition: TextIsContained: "Users with similar names to you"
  }
}
Message@Actions {
  abstract {
    // create
    "create"
  }
  waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element:InputButton:ByAttribute(Attribute:"value",Value:"Create")
      Action:Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }

  }
  Message@Actions {
    abstract {
      // create
      "VERIFY"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element:Browser
      Condition:VERIFY_MALICIOUS_EFFECT
    }
  }
}
```

D.3. WAAL Mapping for Bank Application

Listing D.3: WAAL Mapping for Bank Application

```
SIMULATED_AGENTS { "i", "webBrowser" }

AGENTS_CONFIGURATIONS {
  Agent: "i" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
      Init: {
        Generate::
        Element: Browser
        Action: GoToURL "http://172.16.1.21"
      }

  }
  Agent: "webBrowser" {
      httpBasicAuthentication_Username: ""
      httpBasicAuthentication_Password: ""
      BrowserIP: "127.0.0.1"
      BrowserPort: 4444
      Sessions: { {"webServer"} }     // e.g., { {"user1"}, {"user2"} }
      Driver: REMOTEWEBDRIVER
      Platform: LINUX
      Browser: firefox
      Browser Version: ANY
      Init: {
        Generate::
        Element: Browser
        Action: GoToURL "http://172.16.1.21"
      }
    }
}

MAPPINGS {
  username1:
  abstract: "" concrete: "\"matt\""
  firstname1:
  abstract: "" concrete: "\"Matt\""
  lastname1:
  abstract: "" concrete: "\"buechler\""
  email1:
  abstract: "" concrete: "\"buechler@cs.tum.edu\""
  password1:
  abstract: "" concrete: "\"12345678\""
  username_admin:
  abstract: "" concrete: "\"admin\""
  password_admin:
  abstract: "" concrete: "\"admin\""
}

MESSAGES_TO_ACTIONS {

  Message@Actions {
    abstract {
      // ack_activateClient
      "ack_activateClient"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element: WebpageContent
      Condition: TextIsContained: "Activate Clients"
    }
  }
  Message@Actions {
    abstract {
      // ack_login(firstname_t, lastname_t, email_t)
      "ack_login"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element: WebpageContent
      Condition: TextIsContained: "Overview"
    }
  }
  Message@Actions {
    abstract {
      // loginClient(username_t, password_t)
      "loginClient"
    }
    waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element: Textfield:
        ByXPath("html/body/section[1]/form/table/tbody/tr[1]/td[2]/input")
      Action: Type $4 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[1]/form/table/tbody/tr[2]/td[2]/input")
      Action: Type $5 DefaultValue "defaultValue"

      Generate::
      Element: InputButton: ByAttribute(Attribute:"value",Value:"Login")
      Action: Click
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // activateClient(username_t)
    "activateClient"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element: Button: ByVisibleText("New Clients")
    Action: Click

    Generate::
    Element: Button:
      ByXPath("html/body/section[10]/table/tbody/tr/td[5]/button")
    Action: Click
    ImplicitElement: "html/body/section[10]/table/tbody/tr/td[1]"
    ImplicitAttribute: "innerHTML"
    ImplicitValue: $4
  }
  waal(verification) {
    // waal actions to verify the above message
  }
}
Message@Actions {
  abstract {
    // ack_register
    "ack_register"
  }
  waal(generation) {
    // waal actions to generate the above message
  }
  waal(verification) {
    // waal actions to verify the above message
    Verify::
    Element: WebpageContent
    Condition: TextIsContained: "Login"
  }
}
Message@Actions {
  abstract {
    // loginAdmin(username_t, password_t)
    "loginAdmin"
  }
  waal(generation) {
    // waal actions to generate the above message
    Generate::
    Element: Textfield:
      ByXPath("html/body/section[1]/form/table/tbody/tr[1]/td[2]/input")
    Action: Type $4 DefaultValue "defaultValue"

    Generate::
      Element: Textfield:
        ByXPath("html/body/section[1]/form/table/tbody/tr[2]/td[2]/input")
      Action: Type $5 DefaultValue "defaultValue"

      Generate::
      Element: InputButton: ByAttribute(Attribute:"value",Value:"Login")
      Action: Click

      Generate::
      Element: Button: ByXPath("html/body/nav/button[6]")
      Action: Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // register(username_t, firstname_t, lastname_t, email_t, password_t)
      "register"
    }
    waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element: Link: ByVisibleText("register")
      Action: Click

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[1]/td[2]/input")
      Action: Type $4 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[2]/td[2]/input")
      Action: Type $5 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[3]/td[2]/input")
      Action: Type $6 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[4]/td[2]/input")
      Action: Type $7 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[5]/td[2]/input")
      Action: Type $7 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[7]/td[2]/input")
      Action: Type $8 DefaultValue "defaultValue"

      Generate::
      Element: Textfield:
        ByXPath("html/body/section[2]/form/table/tbody/tr[8]/td[2]/input")
      Action: Type $8 DefaultValue "defaultValue"

      Generate::
      Element: InputButton:
        ByAttribute(Attribute:"value",Value:"Register")
      Action: Click
      Additional action:AcknowledgeAlert
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // logout
      "logout"
    }
    waal(generation) {
      // waal actions to generate the above message
      Generate::
      Element: Button: ByID("logoutButton")
      Action: Click
    }
    waal(verification) {
      // waal actions to verify the above message
    }
  }
  Message@Actions {
    abstract {
      // listPendingAccounts(username_t, firstname_t, lastname_t, email_t)
      "listPendingAccounts"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element: Button:
        ByXPath("html/body/section[10]/table/tbody/tr/td[5]/button")
      Condition: ElementIsDisplayed
    }
  }

  Message@Actions {
    abstract {
      // create
      "VERIFY"
    }
    waal(generation) {
      // waal actions to generate the above message
    }
    waal(verification) {
      // waal actions to verify the above message
      Verify::
      Element:Browser
      Condition:VERIFY_MALICIOUS_EFFECT
    }
  }
}
```

E. Makefiles

E.1. Makefile for Selenium Grid Hub

Listing E.1: Makefile for Selenium Grid Hub

```
DL=curl --remote-name
URL=http://selenium-release.storage.googleapis.com/2.42
install:
  $(DL) $(URL)/selenium-server-standalone-2.42.2.jar

run:
  java
    -jar selenium-server-standalone-2.42.2.jar
    -role hub
    -maxSession 50
    -DPOOL_MAX 512
```

E.2. Makefile for Selenium Grid Node

Listing E.2: Makefile for Selenium Grid Node

```
DL=curl --remote-name
URL=http://selenium-release.storage.googleapis.com/2.42
install:
        $(DL) $(URL)/selenium-server-standalone-2.42.2.jar

register:
        java
          -jar selenium-server-standalone-2.42.2.jar
          -role node
          -hub http://localhost:4444/grid/register
          -maxSession=50
          -browser "maxInstances=50,browserName=firefox"
```

List of Acronyms

SUV	System Under Validation	190
AAT	Abstract Attack Trace	197
WAAL	Web Application Abstract Language	215
DSL	Domain Specific Language	184
TEE	Test Execution Engine	165
XSS	Cross-site Scripting	183
SQLI	SQL Injection	62
AST	Abstract Syntax Tree	74
ZAP	OWASP Zed Attack Proxy	116
LTS	Labeled Transition System	191
FSM	Finite State Machine	191
MBT	Model-based Testing	189
MBST	Model-based Security Testing	189
GA	Generation Action	76
VA	Verification Action	76
BA	Browser Actions	87
RA	Recovery Actions	87
DNS	Domain Name System	3
ASLR	Address Space Layout Randomization	8
IDS	Intrusion Detection System	9
CVE	Common Vulnerabilities and Exposures	192
SQL	Structured Query Language	183
AST	Abstract Syntax Tree	74
HLPSL	High Level Protocol Specification Language	95

Bibliography

[1] Anantasec: Web vulnerability scanners evaluation (january 2009). http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html. Accessed: 2015-03-16.

[2] Cert/cc. http://www.cert.org/stats/cert_stats.htm.

[3] Cve-2015-2237. http://www.cvedetails.com/cve/CVE-2015-2237/. Accessed: 2015-03-28.

[4] Acunetix web vulnerability scanner overview. http://www.acunetix.com/support/docs/wvs/overview/, . Accessed: 2015-06-29.

[5] Acunetix. https://www.acunetix.com/, . Accessed: 2015-06-22.

[6] Appscan. http://www-03.ibm.com/software/products/de/appscan. Accessed: 2015-06-22.

[7] Bank aslan++ model. http://models.spacite.matt-buechler.com/bank.aslanpp.

[8] Betster (php betoffice) authentication bypass and sql injection. http://www.securityfocus.com/archive/1/archive/1/534816/100/0/threaded, . Accessed: 2015-03-28.

[9] Betster : online betting for your office. http://betster.sourceforge.net/, . Accessed: 2015-03-28.

[10] Burp. http://portswigger.net/burp/. Accessed: 2015-06-22.

[11] Cve. http://www.cvedetails.com. Accessed: 2015-03-28.

[12] Grendle-scan. http://sourceforge.net/projects/grendel/. Accessed: 2015-06-22.

[13] Imperva web application attack report, july 2013. http://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed4.pdf. Accessed: 2014-06-04.

[14] Junit. http://junit.org/. Accessed: 2015-05-18.

[15] Milescan. http://www.milescan.com/. Accessed: 2015-06-22.

[16] mysql_real_escape_string. http://php.net/manual/de/function.mysql-real-escape-string.php. Accessed: 2014-09-08.

[17] N-stalker. http://www.nstalker.com/. Accessed: 2015-06-22.

[18] Ntospider. http://www.rapid7.com/products/appspider/. Accessed: 2015-06-22.

[19] Nunit. http://www.nunit.org/. Accessed: 2015-05-18.

[20] Owasp broken web applications project. https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project, . Accessed: 2015-04-08.

[21] Owasp top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10, . Accessed: 2015-03-12.

[22] Xss filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, . Accessed: 2015-05-22.

[23] Cve details - owncloud. http://www.cvedetails.com/product/22262/Owncloud-Owncloud.html?vendor_id=11929, . Accessed: 2014-07-08.

[24] Owncloud. https://owncloud.org/, . Accessed: 2014-07-08.

[25] Paros. http://sourceforge.net/projects/paros/. Accessed: 2015-06-22.

[26] Seleniumhq browser automation. http://seleniumhq.org, . Accessed: 2014-07-06.

[27] Selenium element locators. https://selenium.googlecode.com/git/docs/api/py/selenium/selenium.selenium.html, . Accessed: 2015-05-21.

[28] Selenium webdriver. http://docs.seleniumhq.org/docs/03_webdriver.jsp, . Accessed: 2014-07-06.

[29] Bypassing chrome's anti-xss filter. http://blog.securitee.org/?p=37. Accessed: 2015-04-17.

[30] How to escape special character in mysql. http://stackoverflow.com/questions/881194/how-to-escape-special-character-in-mysql, . Accessed: 2014-10-31.

[31] Mysql 5.7 reference manual :: Language structure :: Literal values :: String literals. http://dev.mysql.com/doc/refman/5.7/en/string-literals.html, . Accessed: 2014-10-29.

[32] Testng. http://testng.org/doc/index.html. Accessed: 2015-04-06.

[33] Vmware vsphere. http://www.vmware.com/products/vsphere. Accessed: 2015-06-27.

[34] W3af. http://w3af.org/. Accessed: 2015-06-22.

[35] Wackopicko aslan++ model. http://models.spacite.matt-buechler.com/wackopicko.aslanpp, .

[36] Wackopicko: Traces generated by Syntactic Mutation Operators. spacite.matt-buechler.com, . Accessed: 2015-06-29.

[37] Wackopicko. http://www.aldeid.com/wiki/WackoPicko, . Accessed: 2015-04-18.

[38] Webgoat aslan++ model. http://models.spacite.matt-buechler.com/webgoat.aslanpp, .

[39] Category:owasp webgoat project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, . Accessed: 2015-04-08.

[40] Webinspect. http://resources.infosecinstitute.com/webinspect/, . Accessed: 2015-06-29.

[41] Webinspect. http://www8.hp.com/de/de/software-solutions/webinspect-dynamic-analysis-dast/, . Accessed: 2015-06-29.

[42] Website security statistics report, may 2013. https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf. Accessed: 2014-06-05.

[43] Wordpress.org. https://wordpress.org/. Accessed: 2015-05-21.

[44] Template expressions (xtend). https://eclipse.org/xtend/documentation/203_xtend_expressions.html#templates. Accessed: 2015-05-21.

[45] Cross-site scripting (xss). https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29, . Accessed: 2015-04-05.

[46] Types of cross-site scripting. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting, . Accessed: 2015-04-05.

[47] Xtend. https://eclipse.org/xtend, . Accessed: 2015-05-21.

[48] Xtext - language development made easy. https://eclipse.org/Xtext, . Accessed: 2015-05-21.

[49] Owasp zed attack proxy. https://www.owasp.org/index.php/ZAP. Accessed: 2014-08-23.

[50] Ryma Abassi and SGE Fatmi. Using security policies in a network securing process. In *Telecommunications (ICT), 2011 18th International Conference on*, pages 416–421. IEEE, 2011.

[51] Ryma Abbassi and Sihem Guemara El Fatmi. Towards a test cases generation method for security policies. In *Telecommunications, 2009. ICT'09. International Conference on*, pages 41–46. IEEE, 2009.

[52] Sarah Al-Azzani and Rami Bahsoon. Using implied scenarios in security testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, pages 15–21. ACM, 2010.

[53] Ehab Al-Shaer, Adel El-Atawy, and Taghrid Samak. Automated pseudo-live testing of firewall configuration enforcement. *Selected Areas in Communications, IEEE Journal on*, 27(3):302–314, 2009.

[54] William H Allen, Chin Dou, and Gerald A Marin. A model-based approach to the security testing of network protocol implementations. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 1008–1015. IEEE, 2006.

[55] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *ICECCS*, pages 212–221, 2001.

[56] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98*, pages 46–54. IEEE Computer Society, 1998.

[57] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[58] João Antunes and Nuno Fuentecilla Neves. Using behavioral profiles to detect software flaws in network servers. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 1–10. IEEE, 2011.

[59] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2610403. URL `http://doi.acm.org/10.1145/2610384.2610403`.

[60] Alessandro Armando, Giancarlo Pellegrino, Roberto Carbone, Alessio Merlo, and Davide Balzarotti. From model-checking to automated testing of security protocols: bridging the gap. In *TAP'12: Proceedings of the 6th international conference on Tests and Proofs*, pages 3–18, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30472-9. doi: 10.1007/978-3-642-30473-6_3. URL `http://www.ai-lab.it/armando/pub/tap2012.pdf`.

[61] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *TSE*, 36(4):474–494, 2010.

[62] Avantssar. Deliverable D2.3 (update) ASLan++ specification and tutorial. `http://www.avantssar.eu/pdf/deliverables/avantssar-d2-3_update.pdf`, 2008.

[63] Alexandre Bartel, Benoit Baudry, Freddy Munoz, Jacques Klein, Tejeddine Mouelhi, and Yves Le Traon. Model driven mutation applied to adaptative systems testing. In

Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pages 408–413. IEEE, 2011.

[64] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[65] Hakim Belhaouari, Pierre Konopacki, Régine Laleau, and Marc Frappier. A design by contract approach to verify access control policies. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 263–272. IEEE, 2012.

[66] E. Bernard, Fabrice Bouquet, A. Charbonnier, Bruno Legeard, Fabien Peureux, Mark Utting, and E. Torreborre. Model-based testing from UML models. In *MBT'2006, Model-based Testing Workshop, INFORMATIK'06*, volume P-94 of *LNI, Lecture Notes in Informatics*, pages 223–230, Dresden, Germany, October 2006. ISBN 978-3-88579-188-1. ISBN 978-3-88579-188-1.

[67] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. Automatic xacml requests generation for policy testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 842–849. IEEE, 2012.

[68] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, Eda Marchetti, Fabio Martinelli, and Paolo Mori. Testing of polpa authorization systems. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 8–14. IEEE, 2012.

[69] Dirk Beyer, Adam J Chlipala, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335. IEEE Computer Society, 2004.

[70] Abian Blome, Martin Ochoa, Keqin Li, Michele Peroli, and Mohammad Torabi Dashti. Vera: A flexible model-based vulnerability testing tool. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 471–478. IEEE, 2013.

[71] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs# 11 security tokens. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 260–269. ACM, 2010.

[72] Julien Botella, Fabrice Bouquet, Jean-Francois Capuron, Franck Lebeau, Bruno Legeard, and Florence Schadle. Model-based testing of cryptographic components–lessons learned from experience. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 192–201. IEEE, 2013.

[73] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 45–48, Leipzig, Germany, May 2008. ACM Press. ISBN 978-1-60558-030-2. doi: http://doi.acm.org/10.1145/1370042.1370052. URL `http://doi.acm.org/10.1145/1370042.1370052`.

[74] Josip Bozic and Franz Wotawa. Xss pattern for attack modeling in testing. In *Automation of Software Test (AST), 2013 8th International Workshop on*, pages 71–74. IEEE, 2013.

[75] Shane Bracher and Padmanabhan Krishnan. Enabling security testing from specification to code. In *Integrated Formal Methods*, pages 150–166. Springer, 2005.

[76] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[77] Achim D Brucker and Burkhart Wolff. Test-sequence generation with hol-testgen with an application to firewall testing. In *Tests and Proofs*, pages 149–168. Springer, 2007.

[78] Achim D Brucker, Lukas Brügger, and Burkhart Wolff. Model-based firewall conformance testing. In *Testing of Software and Communicating Systems*, pages 103–118. Springer, 2008.

[79] Achim D Brucker, Lukas Bruügger, Paul Kearney, and Burkhart Wolff. Verified firewall policy transformations for test case generation. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 345–354. IEEE, 2010.

[80] Achim D Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. An approach to modular and testable security models of real-world health-care applications. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, pages 133–142. ACM, 2011.

[81] M. Büchler. Security testing with fault-models and properties. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 501–502, March 2013. doi: 10.1109/ICST.2013.74.

[82] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Security mutants for property-based testing. In *Tests and Proofs*, pages 69–77. Springer, 2011.

[83] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 253–262. IEEE, 2012.

[84] Zhe Chen, Shize Guo, and Damao Fu. A directed fuzzing based on the dynamic symbolic execution and extended program behavior model. In *Proceedings of the 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 1641–1644. IEEE Computer Society, 2012.

[85] John A. Clark, Haitao Dan, and Robert M. Hierons. Semantic mutation testing. *Sci. Comput. Program.*, 78(4):345–363, April 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2011.03.011. URL `http://dx.doi.org/10.1016/j.scico.2011.03.011`.

[86] Frédéric Dadeau, P-C Héam, and Rafik Kheddam. Mutation-based test generation from security protocols in hlpsl. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 240–248. IEEE, 2011.

[87] Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Test generation for network security rules. In *Testing of Communicating Systems*, pages 341–356. Springer, 2006.

[88] Vianney Darmaillacq, J Richier, and Roland Groz. Test generation and execution for security rules in temporal logic. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 252–259. IEEE, 2008.

[89] F S De Boer, M B Bonsangue, A Grüner, and M Steffen. Java test driver generation from object-oriented interaction traces. *Electronic Notes in Theoretical Computer Science*, 243:33–47, 2009.

[90] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program Mutation: A New Approach to Program Testing. In *Infotech State of the Art Report, Software Testing*, pages 107–126, 1979.

[91] Arilo C Dias-Neto and Guilherme H Travassos. A picture from the model-based testing area: Concepts, techniques, and challenges. *Advances in Computers*, 80:45–120, 2010.

[92] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14214-1, 978-3-642-14214-7. URL `http://dl.acm.org/citation.cfm?id=1884848.1884858`.

[93] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006. ISBN 0321444426.

[94] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *CODASPY*, pages 37–48, San Antonio, Texas, USA, 2014. ACM. doi: 10.1145/2557547.2557550.

[95] A. Dupuy and N. Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*, volume 1, pages 1B6/1–1B6/7 vol.1, 2000.

[96] Donia El Kateb, Yehia El Rakaiby, Tejeddine Mouelhi, and Yves Le Traon. Access control enforcement testing. In *Automation of Software Test (AST), 2013 8th International Workshop on*, pages 64–70. IEEE, 2013.

[97] Mazen El Maarabani, Iksoon Hwang, and Ana Cavalli. A formal approach for interoperability testing of security rules. In *Signal-Image Technology and Internet-Based Systems (SITIS), 2010 Sixth International Conference on*, pages 277–284. IEEE, 2010.

[98] Yehia Elrakaiby, Tejeddine Mouelhi, and Yves Le Traon. Testing obligation policy enforcement using mutation analysis. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 673–680. IEEE, 2012.

[99] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. j-post: a java toolchain for property-oriented software testing. *Electronic Notes in Theoretical Computer Science*, 220(1):29–41, 2008.

[100] Funmilade Faniyi, Rami Bahsoon, Andy Evans, and Rick Kazman. Evaluating security properties of architectures in unpredictable environments: A case for cloud. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 127–136. IEEE, 2011.

[101] Dan Farmer and Wietse Venema. Improving the security of your site by breaking into it, 1993.

[102] M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. In *The Third International Conference on Advances in System Testing and Validation Lifecycle(VALID 2011)*, pages 109–114, 2011.

[103] Michael Felderer, Berthold Agreiter, and Ruth Breu. Evolution of security requirements tests for service–centric systems. In *Engineering Secure Software and Systems*, pages 181–194. Springer, 2011.

[104] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: A taxonomy and systematic classification. *Software Testing, Verification and Reliability*, 2015. accepted for publication.

[105] Elizabeta Fourneret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jurjens, and Parvaneh Yousefi. Model-based security verification and testing for smart-cards. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 272–279. IEEE, 2011.

[106] G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: a survey. *STVR*, 19(3):215–261, 2009.

[107] Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, November 2006. ISSN 0163-5948. doi: 10.1145/1218776.1218787. URL `http://doi.acm.org/10.1145/1218776.1218787`.

[108] Xiang Fu, Xin Lu, B. Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 87–96, July 2007. doi: 10.1109/COMPSAC.2007.43.

[109] M. Gegick and L. Williams. Toward the use of automated static analysis alerts for early identification of vulnerability- and attack-prone components. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, pages 18–18, July 2007. doi: 10.1109/ICIMP.2007.46.

[110] David P Gilliam, John D Powell, John C Kelly, and Matt Bishop. Reducing software security risk through an integrated approach. In *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, pages 36–42. IEEE, 2001.

[111] P. Godefroid, M.Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008. 16 pages.

[112] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012. ISSN 1542-7730. doi: 10.1145/2090147.2094081. URL `http://doi.acm.org/10.1145/2090147.2094081`.

[113] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

[114] Fanglu Guo, Yang Yu, and Tzi cker Chiueh. Automated and safe vulnerability assessment. In *Computer Security Applications Conference, 21st Annual*, pages 10 pp.–159, Dec 2005. doi: 10.1109/CSAC.2005.11.

[115] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 25(5):661–674, 1999.

[116] Aiman Hanna, Hai Zhou Ling, Jason Furlong, Zhenrong Yang, and Mourad Debbabi. Targeting security vulnerabilities: From specification to detection (short paper). In *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, pages 97–102. IEEE, 2008.

[117] Aiman Hanna, Hai Zhou Ling, XiaoChun Yang, and Mourad Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. In *On the Move to Meaningful Internet Systems: OTM 2009*, pages 815–832. Springer, 2009.

[118] Alan Hartman, Mika Katara, and Sergey Olvovsky. Choosing a test modeling language: A survey. In *Hardware and Software, Verification and Testing*, pages 204–218. Springer, 2007.

[119] Ke He, Zhiyong Feng, and Xiaohong Li. An attack scenario based approach for software security testing at design stage. In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, volume 1, pages 782–787. IEEE, 2008.

[120] M.P.E. Heimdahl, M.W. Whalen, A. Rajan, and M. Staats. On mc/dc and implementation structure: An empirical study. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 5.B.3–1–5.B.3–13, 2008.

[121] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):9, 2009.

[122] Dominik Holling, Alexander Pretschner, and Matthias Gemmar. 8cage: lightweight fault-based test generation for simulink. In *ASE'14*, pages 859–862, 2014.

[123] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pages 114–123. IEEE, 2008.

[124] Hongxin Hu and GailJoon Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 195–204, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-129-3. doi: 10.1145/1377836.1377867. URL `http://doi.acm.org/10.1145/1377836.1377867`.

[125] Vincent C Hu, Evan Martin, JeeHyun Hwang, and Tao Xie. Conformance checking of access control policies specified in xacml. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 275–280. IEEE, 2007.

[126] Vincent C Hu, D Richard Kuhn, and Tao Xie. Property verification for generic access control models. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 2, pages 243–250. IEEE, 2008.

[127] Bo Huang and Qiaoyan Wen. An automatic fuzz testing method designed for detecting vulnerabilities on all protocol. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 2, pages 639–642. IEEE, 2011.

[128] JeeHyun Hwang, Tao Xie, Fei Chen, and Alex X Liu. Systematic structural testing of firewall policies. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 105–114. IEEE, 2008.

[129] JeeHyun Hwang, Tao Xie, Vincent Hu, and Mine Altunay. Acpt: A tool for modeling and verifying access control policies. In *Policies for Distributed Systems and Networks (POLICY), 2010 IEEE International Symposium on*, pages 40–43. IEEE, 2010.

[130] JeeHyun Hwang, Tao Xie, Fei Chen, and Alex X Liu. Systematic structural testing of firewall policies. *Network and Service Management, IEEE Transactions on*, 9(1):1–11, 2012.

[131] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, December 1990.

[132] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE*, pages 435–445, 2014.

[133] ISO/IEC. *ISO/IEC 27005:2011 Information technology – Security techniques – Information security risk management*, 2011.

[134] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[135] Jacques Julliand, Pierre-Alain Masson, and Regis Tissot. Generating security tests in addition to functional tests. In *Proceedings of the 3rd international workshop on Automation of software test*, pages 41–44. ACM, 2008.

[136] Jacques Julliand, Pierre-Alain Masson, Régis Tissot, and Pierre-Christophe Bué. Generating tests from b specifications and dynamic selection criteria. *Formal aspects of computing*, 23(1):3–19, 2011.

[137] Jan Jürjens. Model-based security testing using umlsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104, 2008.

[138] Jan Jurjens and Guido Wimmel. Formally testing fail-safety of electronic purse protocols. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 408–411. IEEE, 2001.

[139] Jan Jürjens and Guido Wimmel. Specification-based testing of firewalls. In *Perspectives of System Informatics*, pages 308–316. Springer, 2001.

[140] Ben WY Kam and Thomas R Dean. Linguistic security testing for text communication protocols. In *Testing–Practice and Research Techniques*, pages 104–117. Springer, 2010.

[141] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *ICSE*, pages 199–209, 2009.

[142] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33:2004, 2004.

[143] Yves Le Traon, Tejeddine Mouelhi, Franck Fleurey, and Benoit Baudry. Language-specific vs. language-independent approaches: embedding semantics on a meta-model for testing and verifying access control policies. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 72–79. IEEE, 2010.

[144] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. In *SECTEST'13, 4-th Int. Workshop on Security Testing. In conjunction with ICST'13, 6-th IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 445–452, Luxembourg, Luxembourg, March 2013. IEEE Computer Society Press. doi: 10.1109/ICSTW.2013.58. URL `http://dx.doi.org/10.1109/ICSTW.2013.58`.

[145] Keqin Li, Laurent Mounier, and Roland Groz. Test generation from security policies specified in or-bac. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 255–260. IEEE, 2007.

[146] Mass Soldal Lund, Bjornar Solhaug, and Ketil Stølen. *Model-driven risk analysis: the CORAS approach*. Springer, 2011.

[147] Y.K. Malaiya, M.N. Li, J.M. Bieman, and R. Karcich. Software reliability growth with test coverage. *Reliability, IEEE Transactions on*, 51(4):420–426, 2002.

[148] Wissam Mallouli and Ana Cavalli. Testing security rules with decomposable activities. In *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE*, pages 149–155. IEEE, 2007.

[149] Wissam Mallouli, Mounir Lallali, Gerardo Morales, and Ana R Cavalli. Modeling and testing secure web-based systems: Application to an industrial case study. In *Signal Image Technology and Internet Based Systems, 2008. SITIS'08. IEEE International Conference on*, pages 128–136. IEEE, 2008.

[150] Wissam Mallouli, Amel Mammar, and Ana Cavalli. A formal framework to integrate timed security rules within a tefsm-based system specification. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*, pages 489–496. IEEE, 2009.

[151] Amel Mammar, Wissam Mallouli, and Ana Cavalli. A systematic approach to integrate common timed security rules within a tefsm-based system specification. *Information and Software Technology*, 54(1):87–98, 2012.

[152] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. Security test generation using threat trees. In Dimitris Dranidis, Stephen P. Masticola, and Paul A. Strooper, editors, *AST*, pages 62–69. IEEE, 2009. ISBN 978-1-4244-3711-5. URL `http://dblp.uni-trier.de/db/conf/icse/ast2009.html#MarbackDHKX09`.

[153] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258, 2013.

[154] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 667–676, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242663. URL `http://doi.acm.org/10.1145/1242572.1242663`.

[155] Evan Martin, JeeHyun Hwang, Tao Xie, and Vincent Hu. Assessing quality of policy properties in verification of access control policies. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 163–172. IEEE, 2008.

[156] Ammar Masood, Arif Ghafoor, and Aditya P Mathur. Conformance testing of temporal role-based access control systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(2):144–158, 2010.

[157] Pierre-Alain Masson, Jacques Julliand, Jean-Chritophe Plessis, Eddie Jaffuel, and Georges Debois. Automatic generation of model based tests for a class of security properties. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 12–22. ACM, 2007.

[158] Gary McGraw. Software security. *Security & Privacy, IEEE*, 2(2):80–83, 2004.

[159] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN 0321356705.

[160] Michael Mlynarski, Baris Güldali, Gregor Engels, and Stephan Weißleder. Model-based testing: Achievements and future challenges. *Advances in Computers*, 86:1–39, 2012.

[161] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE.

[162] Anderson Morais, Eliane Martins, Ana Cavalli, and Willy Jimenez. Security protocol testing using attack trees. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 690–697. IEEE, 2009.

[163] Anderson Morais, Ana Cavalli, and Eliane Martins. A model-based attack injection approach for security validation. In *Proceedings of the 4th international conference on Security of information and networks*, pages 103–110. ACM, 2011.

[164] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Mutation analysis for security tests qualification. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 233–242. IEEE, 2007.

[165] Tejeddine Mouelhi, Franck Fleurey, and Benoit Baudry. A generic metamodel for security policies mutation. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 278–286. IEEE, 2008.

[166] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In *Model Driven Engineering Languages and Systems*, pages 537–552. Springer, 2008.

[167] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Transforming and selecting functional test cases for security policy testing. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 171–180. IEEE, 2009.

[168] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.

[169] Phu H Nguyen, Mike Papadakis, and Iram Rubab. Testing delegation policy enforcement via mutation analysis. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 34–42. IEEE, 2013.

[170] George Noseevich and Andrew Petukhov. Detecting insufficient access control in web applications. In *SysSec Workshop (SysSec), 2011 First*, pages 11–18. IEEE, 2011.

[171] Percy A Pari-Salas and Padmanabhan Krishnan. Testing privacy policies using models. *Information Technology papers*, 2008.

[172] H. Peine. Security test tools for web applications. Technical Report 048.06, Fraunhofer IESE, Jan 2006.

[173] B. Potter and G. McGraw. Software Security Testing. *IEEE Security & Privacy*, 2004.

[174] A. Pretschner. Defect-based testing. In *Irlbeck, M., Peled, D., Pretschner, A.: Dependable Software Systems Engineering*, NATO Science for Peace and Security Series - D: Information and Communication Security, pages 224–245. IOS Press, 2015. doi: 10.3233/978-1-61499-495-4-224.

[175] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-based tests for access control policies. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 338–347, April 2008. doi: 10.1109/ICST.2008.44.

[176] Alexander Pretschner, Tejeddine Mouelhi, and Yves Le Traon. Model-based tests for access control policies. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 338–347. IEEE, 2008.

[177] Alexander Pretschner, Dominik Holling, Robert Eschbach, and Matthias Gemmar. A generic fault model for quality assurance. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2013. ISBN 978-3-642-41532-6. URL `http://dblp.uni-trier.de/db/conf/models/models2013.html#PretschnerHEG13`.

[178] Alexander Pretschner, Dominik Holling, Robert Eschbach, and Matthias Gemmar. A generic fault model for quality assurance. In *Proc. ACM/IEEE 16th Intl. Conf. on Model Driven Engineering Languages and Systems*, 2013.

[179] C. R. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *IN PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON VERIFICATION, MODEL CHECKING AND ABSTRACT INTERPRETATION*, 1998.

[180] Slim Rekhis, Baha Bennour, and Noureddine Boudriga. Validation of security solutions for communication networks: A policy-based approach. In *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pages 115–122. IEEE, 2011.

[181] Dean Rosenzweig, Davor Runje, and Wolfram Schulte. Model–based testing of cryptographic protocols. In *Trustworthy Global Computing*, pages 33–60. Springer, 2005.

[182] Ayda Saidane and Nicolas Guelfi. Towards improving security testability of aadl architecture models. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 353–357. IEEE, 2011.

[183] P.A.P. Salas, P. Krishnan, and K.J. Ross. Model-based security vulnerability testing. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 284–296, April 2007. doi: 10.1109/ASWEC.2007.31.

[184] Sébastien Salva and Stassia R Zafimiharisoa. Data vulnerability detection by security testing for android applications. In *Information Security for South Africa, 2013*, pages 1–8. IEEE, 2013.

[185] Aymerick Savary, Marc Frappier, and Jean-Louis Lanet. Detecting vulnerabilities in java-card bytecode verifiers using model-based testing. In *Integrated Formal Methods*, pages 223–237. Springer, 2013.

[186] Ina Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, 2012.

[187] Ina Schieferdecker, Juergen Grossmann, and Martin Schneider. Model-based security testing. *Proceedings 7th Workshop on Model-Based Testing*, 2012.

[188] Martin Schneider, Jurgen Grossmann, Ina Schieferdecker, and Andrej Pietschker. Online model-based behavioral fuzzing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 469–475. IEEE, 2013.

[189] Martin Schneider, Jürgen Großmann, Nikolay Tcholtchev, Ina Schieferdecker, and Andrej Pietschker. *Behavioral fuzzing operators for UML sequence diagrams*. Springer, 2013.

[190] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM. ISBN 1-58113-449-5. doi: 10.1145/511446.511498. URL `http://doi.acm.org/10.1145/511446.511498`.

[191] Diana Senn, David Basin, and Germano Caronni. Firewall conformance testing. In *Testing of Communicating Systems*, pages 226–241. Springer, 2005.

[192] Hossain Shahriar and Mohammad Zulkernine. Automatic testing of program security vulnerabilities. In *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 2, pages 550–555. IEEE, 2009.

[193] Hossain Shahriar and Mohammad Zulkernine. Phishtester: automatic testing of phishing attacks. In *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, pages 198–207. IEEE, 2010.

[194] Guoqiang Shu and David Lee. Message confidentiality testing of security protocols–passive monitoring and active checking. In *Testing of Communicating Systems*, pages 357–372. Springer, 2006.

[195] Guoqiang Shu and David Lee. Testing security properties of protocol implementations-a machine learning based approach. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 25–25. IEEE, 2007.

[196] Guoqiang Shu, Yating Hsu, and David Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *Formal Techniques for Networked and Distributed Systems–FORTE 2008*, pages 299–304. Springer, 2008.

[197] Sankalp Singh, James Lyons, and David M Nicol. Fast model-based penetration testing. In *Proceedings of the 36th conference on Winter simulation*, pages 309–317. Winter Simulation Conference, 2004.

[198] SPaCIoS. Deliverable 2.1.1: Analysis of the relevant concepts used in the case studies: applicable security concepts, security goals and attack behaviours, 2011.

[199] SPaCIoS. Deliverable 2.1.2: Modeling security-relevant aspects in the IoS, 2012.

[200] SPaCIoS. Deliverable 5.5: Final Tool Assessment, 2014.

[201] Bernard Stepien, Liam Peyton, and Pulei Xiong. Using ttcn-3 as a modeling language for web penetration testing. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, pages 674–681. IEEE, 2012.

[202] L. Suto. Analyzing the effectiveness and coverage of web application security scanners, case study, Oct 2007.

[203] L. Suto. Analyzing the accuracy and time costs of web application security scanners, Feb 2010.

[204] Wen Tang, Ai-Fen Sui, and W. Schmid. A model guided security vulnerability discovery approach for network protocol implementation. In *Communication Technology (ICCT), 2011 IEEE 13th International Conference on*, pages 675–680, Sept 2011. doi: 10.1109/ICCT.2011.6157962.

[205] Gu Tian-yang, Shi Yin-sheng, and Fang You-yuan. Research on software security testing. *World Academy of Science, Engineering and Technology Issure*, 69:647–651, 2010.

[206] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing security policies: going beyond functional testing. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 93–102. IEEE, 2007.

[207] Issa Traore and Demissie B Aredo. Enhancing structured review with model-based verification. *Software Engineering, IEEE Transactions on*, 30(11):736–753, 2004.

[208] Tugkan Tuglular and Fevzi Belli. Protocol-based testing of firewalls. In *Formal Methods (SEEFM), 2009 Fourth South-East European Workshop on*, pages 53–59. IEEE, 2009.

[209] Tugkan Tuglular and Gurcan Gercek. Feedback control based test case instantiation for firewall testing. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 202–207. IEEE, 2010.

[210] Tugkan Tuglular and Gurcan Gercek. Mutation-based evaluation of weighted test case selection for firewall testing. In *Secure Software Integration and Reliability Improvement (SSIRI), 2011 Fifth International Conference on*, pages 157–164. IEEE, 2011.

[211] Tugkan Tuglular, O Kaya, Can Arda Muftuoglu, and Fevzi Belli. Directed acyclic graph modeling of security policies for firewall testing. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 393–398. IEEE, 2009.

[212] Yves Turcotte, Oded Tal, Scott Knight, and Thomas Dean. Security vulnerabilities assessment of the x. 509 protocol by syntax-based testing. In *Military Communications Conference, 2004. MILCOM 2004. 2004 IEEE*, volume 3, pages 1572–1578. IEEE, 2004.

[213] Mathieu Turuani. The cl-atse protocol analyser. In Frank Pfenning, editor, *Term Rewriting and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-36834-2. doi: 10.1007/11805618_21. URL `http://dx.doi.org/10.1007/11805618_21`.

[214] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab*, 22(2):29–312, 2012.

[215] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123725011.

[216] Alexandre Vernotte, Frédéric Dadeau, Franck Lebeau, Bruno Legeard, Fabien Peureux, and François Piat. Efficient detection of multi-step cross-site scripting vulnerabilities. In *ICISS'14, 10-th Int. Conf. on Information Systems Security*, volume 8880 of *LNCS*, pages 358–377, Hyderabad, India, December 2014. Springer. doi: 10.1007/978-3-319-13841-1_20. URL `http://dx.doi.org/10.1007/978-3-319-13841-1_20`.

[217] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, DTIC Document, 1981.

[218] David von Oheimb and Sebastian Mödersheim. ASLan++ — a formal security specification language for distributed systems. In B. Aichernig, F. de Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects, FMCO 2010, Graz, Austria*, volume 6957 of *LNCS*, pages 1–22. Springer, December 2010. ISBN 978-3-642-25270-9.

[219] Linzhang Wang, Eric Wong, and Dianxiang Xu. A threat model driven approach for security testing. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, page 10. IEEE Computer Society, 2007.

[220] Weiguang Wang, Qingkai Zeng, and Aditya P Mathur. A security assurance framework combining formal verification and security functional testing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 136–139. IEEE, 2012.

[221] Sam Weber, Amitkumar Paradkar, Suzanne McIntosh, D Toll, Paul A Karger, Matthew Kaplan, and Elaine R Palmer. The feasibility of automated feedback-directed specification-based test generation: A case study of a high-assurance operating system. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 229–238. IEEE, 2008.

[222] Tian Wei, Yang Ju-Feng, Xu Jing, and Si Guan-Nan. Attack model based penetration test for sql injection vulnerability. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 589–594. IEEE, 2012.

[223] C. Weissman. Penetration testing. In Marshall D. Abrams, Sushil G. Jajodia, and H. J. Podell, editors, *Information Security: An Integrated Collection of Essays*, pages 269–296. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[224] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, 1991.

[225] Jon Whittle, Duminda Wijesekera, and Mark Hartong. Executable misuse cases for modeling security concerns. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 121–130. IEEE, 2008.

[226] A. Wiegenstein, F. Weidemann, M. Schumacher, and S. Schinzel. Web application vulnerability scanners - a benchmark. Technical report, Virtual Forge GmbH, Oct 2006.

[227] Guido Wimmel and Jan Jürjens. Specification-based test generation for security-critical systems using mutations. In *Formal Methods and Software Engineering*, pages 471–482. Springer, 2002.

[228] Daniel Woodraska, Michael Sanford, and Dianxiang Xu. Security mutation testing of the filezilla ftp server. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1425–1430, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0113-8. doi: 10.1145/1982185.1982493. URL `http://doi.acm.org/10.1145/1982185.1982493`.

[229] Dianxiang Xu and K.E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *Software Engineering, IEEE Transactions on*, 32(4):265–278, April 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.40.

[230] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 209–218. ACM, 2012.

[231] Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. Automated security test generation with formal threat models. *Dependable and Secure Computing, IEEE Transactions on*, 9(4):526–540, 2012.

[232] Dianxiang Xu, Michael Sanford, Zhaoliang Liu, Mark Emry, Brad Brockmueller, Spencer Johnson, and Michael To. Testing access control and obligation policies. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 540–544. IEEE, 2013.

[233] Chen Yan and Wu Dan. A scenario driven approach for security policy testing based on model checking. In *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, pages 1–4. IEEE, 2009.

[234] Ma Yan, Yan Xuexiong, Zhu Yuefei, and Shao Guoliang. A method of ttcn test case generation based on tp description. In *Computer Science and Engineering, 2009. WCSE'09. Second International Workshop on*, volume 1, pages 255–258. IEEE, 2009.

[235] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1070–1076. IEEE, 2012.

[236] Yahui Yang, Yunfei Chen, Min Xia, and Juan Ma. An automated mechanism of security test on network protocols. In *Information Assurance and Security, 2009. IAS'09. Fifth International Conference on*, volume 1, pages 503–506. IEEE, 2009.

[237] Yang Yang, Huanguo Zhang, Mi Pan, Jian Yang, Fan He, and Zhide Li. A model-based fuzz framework to the security testing of tcg software stack implementations. In *Multimedia Information Networking and Security, 2009. MINES'09. International Conference on*, volume 1, pages 149–152. IEEE, 2009.

[238] Hong Yu, Huang Song, Hu Bin, and Yao Yi. Using labeled transition system model in software access control politics testing. In *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2012 Second International Conference on*, pages 680–683, Dec 2012.

[239] Hong Yu, Liu Xiao-Ming, Huang Song, and Zheng Chang-You. Data oriented software security testing. In *Proceedings of the 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 676–679. IEEE Computer Society, 2012.

[240] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. *Model-based testing for embedded systems*, volume 13. CRC Press, 2012.

[241] Philipp Zech. Risk-based security testing in cloud computing environments. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 411–414. IEEE, 2011.

[242] Philipp Zech, Michael Felderer, and Ruth Breu. Towards a model based security testing approach of cloud computing environments. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 47–56. IEEE, 2012.

[243] Philipp Zech, Michael Felderer, and Ruth Breu. Towards risk–driven security testing of service centric systems. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 140–143. IEEE, 2012.

[244] Zhao Zhang, Qiao-Yan Wen, and Wen Tang. An efficient mutation-based fuzz testing approach for detecting flaws of network protocol. In *Computer Science & Service System (CSSS), 2012 International Conference on*, pages 814–817. IEEE, 2012.

[245] Li Zhou, Xia Yin, and Zhiliang Wang. Protocol security testing with spin and ttcn-3. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 511–519. IEEE, 2011.

[246] Mohammad Zulkernine, Mohammad Feroz Raihan, and Mohammad Gias Uddin. Towards model-based automatic testing of attack scenarios. In *Computer Safety, Reliability, and Security*, pages 229–242. Springer, 2009.