JOSIP BOZIC

MODEL-BASED SECURITY TESTING OF WEB APPLICATIONS

Doctoral Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Supervisor: Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa

Graz, January 2016

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ____________________________ ____________________________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ____________________________ ____________________________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Testing of web applications for common vulnerabilities still represents a major challenge in the area of security testing. The objective here is not necessarily to find new vulnerabilities but to ensure that the web application handles well-known attack patterns in a reliable way. Previously developed methods based on model-based testing contribute to the underlying challenge. Actually, the application of this discipline to the security of web applications is the focus of this thesis. Here two approaches are introduced that rely on different methods, namely model-based security testing, combinatorial testing and planning. The corresponding implementations combine these elements into testing frameworks for testing of web applications for vulnerabilities.

Acknowledgements

The research that lead to this thesis has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice), ITEA-2 (DIAMONDS), the Austrian COMET Program (FFG) and in part by the project 3CCar.

List of Publications

[1] J. Bozic and F. Wotawa. Model-based testing - from safety to security. In *Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12)*, pages 9-16, October 2012.

[2] J. Bozic and F. Wotawa. Xss pattern for attack modeling in testing. In *Proceedings of the 8th International Workshop on Automation of Software Test (AST'13)*, 2013.

[3] J. Bozic and F. Wotawa. Security testing based on attack patterns. In *Proceedings of the 5th International Workshop on Security Testing (SECTEST'14)*, 2014.

[4] J. Bozic, D. E. Simos, and F. Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test (AST'14)*, 2014.

[5] A. Bernauer, J. Bozic, D. E. Simos, S. Winkler, and F. Wotawa. Retaining consistency for knowledge-based security testing. In *Proceedings of the 27th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE'14)*, pages 88-97, 2014.

[6] J. Bozic and F. Wotawa. Plan it! automated security testing based on planning. In *Proceedings of the 26th IFIP WG 6.1 International Conference (ICTSS'14)*, pages 48-62, 2014.

[7] J. Bozic, B. Garn, D. E. Simos, and F. Wotawa. Evaluation of the ipo-family algorithms for test case generation in web security testing. In *Proceedings of the Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15)*, pages 1-5, 2015.

[8] J. Bozic, B. Garn, I. Kapsalis, D. E. Simos, S. Winkler, and F. Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'15)*, 2015.

[9] J. Bozic and F. Wotawa. Purity: a planning-based security testing tool. In *Workshop on Trustworthy Computing*, 2015.

Contents

Abstract v

Acknowledgements vi

List of Publications ix

1 Introduction 1

2 Related Research 7

3 Integral Components 17
- 3.1 Definitions . . . 17
- 3.2 Resources . . . 18
- 3.3 SQLI & XSS . . . 19
- 3.4 Test Oracles . . . 23
- 3.5 Systems Under Test . . . 24

4 Model-Based Security Testing 27
- 4.1 Approach overview . . . 28
  - 4.1.1 Model-Based Fuzzing . . . 28
  - 4.1.2 Model Inference Assisted Evolutionary Fuzzing . . . 34
  - 4.1.3 Model-Based Mutation Testing . . . 36
- 4.2 Summary . . . 39

5 Attack Pattern-Based Combinatorial Testing 41
- 5.1 Attack Patterns . . . 42
  - 5.1.1 Modeling Attack Patters . . . 44
- 5.2 Component Interaction . . . 48

5.3 Attack Pattern-Based Testing . . . 50
5.3.1 Case Study . . . 51
5.3.2 Evaluation . . . 55
5.4 Combinatorial Testing . . . 58
5.5 Combining the Approaches . . . 59
5.6 Combinatorial Grammar for XSS Attack Vectors . . . 59
5.7 Attack Pattern-Based Combinatorial Testing: Initial Evaluation 61
5.8 Evaluation of the IPO-Family Algorithms for Test Case Generation . . . 68
5.8.1 Combinatorial Grammar for XSS Attack Vectors and Constraints . . . 69
5.8.2 Combinatorial Metrics for Security Testing . . . 72
5.8.3 Evaluation . . . 73
Exploitation Rate of SUTs and their Relation to the IPO Family . . . 74
Combinatorial Coverage Measurement for Web Security Testing . . . 77
5.9 Automated vs. Manual Testing . . . 82
5.9.1 Penetration Testing Execution Methods . . . 84
5.9.2 Evaluation . . . 85
Exploitation Rate of SUTs . . . 85
Evaluation of Combinatorial Grammars . . . 86
Comparison of Fuzzers and Combinatorial Testing . . 90
Comparison of Automated vs. Manual Test Execution Methods . . . 92
5.10 Summary . . . 92

6 Testing as a Planning Problem 97
6.1 The Security Testing via Planning Approach . . . 98
6.1.1 PLAN4SEC . . . 100
6.1.2 Running Example . . . 105
6.1.3 Evaluation . . . 109
6.2 PURITY: a Planning-based secURITY testing tool . . . 112
6.2.1 Key Components . . . 115
6.2.2 Structure of Inputs in PURITY . . . 118
6.2.3 Modes of Use . . . 119
6.2.4 Case Study . . . 123

6.3 Summary 125

7 Conclusion **129**

Bibliography **135**

List of Figures

4.1 Simple authentication model 29
4.2 Grammar for SQL statements 30
4.3 Parse tree for a generated SQL injection 31
4.4 Statement mutation-1 35
4.5 Statement mutation-2 35
4.6 Statement crossover 36
4.7 SQL injection attack model 37
4.8 Mutated attack model 38

5.1 UML based attack model in Yakindu 46
5.2 Label of a transition 47
5.3 Test creation and execution framework 49
5.4 Result of successful SQLI 53
5.5 Successful triggering of XSS in Mutillidae 54
5.6 System View in ACTS 62
5.7 Comparison of combination coverage measurement for passing tests in DVWA (inp_id 1, DL 0) when their respective test suites are generated in IPOG (above) and IPOG-F (below) with interaction strength $t = 2$ 81
5.8 Comparison of combination coverage measurement for passing tests in Gruyere (inp_id 1, DL 0) when their respective test suites are generated in IPOG (above) and IPOG-F (below) with interaction strength $t = 3$ 82
5.9 Comparison of combination coverage measurement for passing tests in Webgoat (inp_id 2, DL 0) when their respective test suites are generated in IPOG (above) and IPOG-F (below) with interaction strength $t = 4$ 83
5.10 Coverage vs. interaction strength with Attack Pattern-based Combinatorial Testing for BodgeIt (left) and DVWA (right) . . . 91

5.11 Coverage vs. interaction strength in Burp Suite for WebGoat (left) and DVWA (right) 91

6.1 Plan generation and concretization 109
6.2 PURITY in context with its environment 114
6.3 PURITY's internal software architecture 117
6.4 PURITY's GUI 120
6.5 Section for partly automated testing 122
6.6 Menu for manual testing 123
6.7 Output table 124

List of Tables

5.1 Initial evaluation results 56
5.2 A sample of XSS attack vectors 63
5.3 Initial evaluation results - Combinatorial Testing 65
5.4 Evaluation results per SUT for given difficulty level and input field with increasing strength 75
5.5 Evaluation results for measuring combination coverage per SUT with increasing strength - IPOG 78
5.6 Evaluation results for measuring combination coverage per SUT with increasing strength - IPOF 79
5.7 Combinatorial evaluation results for automated and manual tools testing 87
5.8 Evaluation results for fuzzers and combinatorial testing per SUT for given difficulty level and input field - APBT 88
5.9 Evaluation results for fuzzers and combinatorial testing per SUT for given difficulty level and input field - Burp Suite . . 89

6.1 Evaluation results for planning-based testing 111

1 Introduction

With the ever growing interconnectivity between systems in the world there is an even stronger growing need for ensuring systems' security in order to prevent unauthorized access or other malicious acts. The number of potential security threats rises with the increasing number of web applications as well. Vulnerable programs do not only cause costs, they also negatively impact trust in applications and consequently in the companies providing the applications. Consequently preventing vulnerabilities should be a top priority of any provider of systems and services, especially when considering the consequences of unintended software behavior. Such services might handle sensitive data about several thousands or millions of users. It is worth noting that from 2010 and 2013 some vulnerabilities like SQL injection (SQLI) and cross-site scripting (XSS) have belonged to the top three web application security flaws [16]. More interestingly, those two flaws have been under the top 6 from 2004 on, leaving the impression that there has not been enough effort spent in finding well known flaws.

In 2015, several successful hacking attacks made it into the news. For example, due to a SQLI security leak, the Ashley Madison website for extramarital affairs was hacked, thereby confiscating information about 37 million of its users[1]. The list included, among others, government and bank officials as well as military personnel. The obtained information, which included private profile data as well as credit card transactions, was made public by the responsible hacking group. In the aftermath, lawsuits, resignations and at least three commited suicides were reported until now. Later it was also revealed that the website was vulnerable to XSS as well[2].

[1] http://www.businessinsider.com/cheating-affair-website-ashley-madison-hacked-user-data-leaked-2015-7. Accessed: 2015-12-28.

[2] http://www.dailydot.com/politics/ashley-madison-leaked-emails-security/. Accessed: 2015-12-28.

Additionally, the United Arab Emirates Invest Bank was breached and blackmailed in the aftermath by the corresponding hacker, who demanded $3 million in Bitcoin. After the bank refused to submit to the request, the attacker published confiscated information about 50.000 clients[3].

The hack of the White House by Russian hackers caught some public attention as well[4], whereas its official website was vulnerable to XSS all until 2011[5]. Several other exposed XSS vulnerabilities were also reported by eBay[6] and Typo3[7].

Another reason why testing software and systems is by far the most important activity is to ensure high quality in terms of a reduced number of post-release faults occurring after the software deployment. Thus the subsequent damage resulting in debugging and time intensive upgrading may be minimized. Hence, for software developers of such applications one of the major tasks in providing security is to embed testing methodologies into the software development cycle. In this way already known vulnerabilities may be covered so that a program is suited against typical attack vectors. Unfortunately, testing with the aim of finding weaknesses that can be exploited for an attack is usually a manual labor and thus expensive. Therefore, automation of software testing in the domain of software security is highly demanded.

There is a wide range of ideas, proposals, and implementations for testing systems that are based on several white- and black-box testing techniques. Many of these methods are considered part of model-based testing, a technique that relies on generating test cases from a given model of the implementation, where the created test cases are tested against the given SUT. The main function of this procedure is to invoke unintended behavior (negative testing) of the program, thus reporting a potential error that might

[3]http://gulfnews.com/xpress/dubai/courts-crime/hacker-holds-uae-bank-to-ransom-demands-3m-1.1626394. Accessed: 28-12-2015.

[4]http://edition.cnn.com/2015/04/07/politics/how-russians-hacked-the-wh/. Accessed: 28-12-2015.

[5]http://www.mynetsafety.com/2011/11/persistent-xss-vulnerability-in-white.html. Accessed: 28-12-2015.

[6]http://www.infosecurity-magazine.com/news/ebay-under-fire-after-cross-site/. Accessed: 28-12-2015.

[7]http://www.livehacking.com/tag/typo3/. Accessed: 28-12-2015.

correspond to a vulnerability. So the whole process represents an offensive thinking approach on testing, which is also known as ethical hacking (white hat), realized in order to find out how a non-ethical hacker (black hat) may proceed in his intention to exploit a system.

In this research the focus lies on the topic of providing methods and techniques for testing web applications with respect to known vulnerabilities. In particular there is interest in automated methods for finding vulnerabilities without or at least with little user interactions.

The contribution of this work can be summarized as follows:

- Implementation of two testing approaches.
- The depiction and use of attack patterns for automatized testing of web applications (primarily from a black-box perspective) for specific vulnerabilities.
- Specification of an input grammar for XSS attack vectors.
- The combination of attack patterns with combinatorial testing: The result represents the first proposed approach and is called Attack Pattern-based Combinatorial Testing.
- Introduction of the planning problem into security testing of web applications for vulnerabilities. This represents the second proposed approach.
- Implementation of the planning-based tool PURITY for automated testing.

From the discussed issues, the following research objectives of this thesis are defined:

RO1: Identify and represent blueprints for attacks.

Since it is assumed that attackers usually follow a certain pattern when undertaking a specific attack, the goal would be to identify these steps. Actually, such a blueprint for an attack, that is an algorithm to trigger a vulnerability, is to be formalized. This would consider a set of generally true preconditions for an attack with the corresponding set of actions that are necessary in order to carry out a specific type of attack against any web application. Since the work is performed from an offensive point of view, that is, from the view of the attacker, the goal is to depict the necessary attack information in various ways. The approach should not be restricted but be adaptable on every web application as well. In the author's research, two ways of attack representations are provided, namely through attack patterns and as planning specifications.

RO2: Introduce a testing approach for web applications and implement a testing framework.

Various testing methodologies and tools address the problem of security testing of web applications (see Chapter 2). They implement some of the concepts, like attack graphs and use planning.

However, the goal of this work is to provide testing methodologies that enables the testing of web applications for different types of vulnerabilities. The first approach combines model-based security testing with combinatorial testing for a specific purpose, namely the detection of software vulnerability. On the other hand, in the second one the testing problem is specified as a planning problem. Additionally, the planning domain definition language is integrated into a testing tool. Both novel approaches contribute to the problem of automation of software testing.

RO3: The new testing framework serves as an alternative to already established testing approaches.

Already existing testing methods, for example fuzzing and various manual and other automated and semi-automated approaches are used in practice. In order to prove that the introduced approaches are comparable to these approaches for the sake of vulnerability triggering in web applications, a comparison between these approaches is drawn and the results are discussed.

RO4: Adapt planning for testing of web applications and implement a planning-based testing tool for this case.

Planning was initially used in artificial intelligence, for example by intelligent agents, however, its adaptation for vulnerability detection still represents an open problem. While planning for testing has been used before (see Chapter 2 for planning related works), the author's approach adapts it for the detection of vulnerabilities in web applications. First, the planning problem and definition have to be implemented in a way so that a planner generates a plan. Then, a program is tested and a verdict is generated.

According to the objectives, the following research questions are inferred:

RQ1: How can all necessary attack information be formalized?

RQ2: What testing methodologies are developed for vulnerability detection in web applications?

RQ3: How does Attack Pattern-based Combinatorial Testing perform in comparison to fuzzing and manual testing approaches?

RQ4: How does planning-based testing perform for vulnerability detection?

The answers to these questions will be given through the chapters.

This thesis is organized as follows. In the next chapter an overview is given about all related research that was taken into consideration during the research. Chapter 3 explains all common integral elements throughout the work. These include features like definitions and information about the used testing methodology. In Chapter 4 model-based security testing is discussed in more detail. Some of the method's concepts are explained according to given examples. Afterwards, Chapter 5 discusses the first of the proposed two approaches. It describes attack patterns and their adaptation to testing of web applications. The second half of that chapter goes through combinatorial testing and its incorporation into the overall testing framework. The second approach is explained in Chapter 6, namely the implementation of testing as a planning problem. The planning domain definition language is discussed as well as the functionality of the tool PURITY. Finally, the thesis is concluded with Chapter 7.

Another work from [27] discusses a testing-related topic, namely the underlying problem of consistency handling in knowledge systems. An approach is explained for resolving consistency issues in bases for test oracles. However, since the work differs from the bulk of the author's specific research, the main expertise as well as contribution was on the side of the other authors of that paper. Although the explained approach could be extended in the future, it will not be elaborated because it didn't relate further to the author's main research.

2 Related Research

The related research encompasses a variety of works that covers topics of interest to the author's work. Since the proposed research was initially motivated by some of the mentioned works, the differences and similarities to them have to be pointed out.

In general, they comprise four main topics that are related to the presented research, namely:

- Model-based testing
- Attack modeling
- Combinatorial testing
- Planning

The first part focuses on model-based testing. Several works are presented that elaborate the model concepts, testing techniques and vulnerability detection in applications.

The authors of [93] discuss general topics of model-based security testing (MBST) and propose three different model categories in order to cover several security issues in that area. The goal is the defense of a system that provides the inputs for the generation of test cases. The proposed model has to provide different aspects of how to secure a system under test (SUT) against potential security leaks. By doing so, a tester might get an insight about how to proceed when testing an application. Additionally, such modeling provides help in prioritizing and management of test cases. First they suggested the usage of architectural and functional models, which describe the structure and properties of a specific application. The goal here is to identify potential entry points for malicious activity so that appropriate measures can be taken. The authors also suggest threat, fault and risk models. These models try to depict all possible vulnerabilities that might occur as well as the output of their mutual interaction. The last types of models were weakness and vulnerabilities models. The goal of these depictions is to describe the vulnerabilities. These are categorized so that testers and developers get aware of potential malicious behavior. On the other hand, the notions of risk-based security testing and model-based fuzzing are explained as well. Its goals are test optimization and triggering of vulnerabilities with atypical inputs, respectively.

In [92] the author explained the use of model-based fuzz testing for security testing. In this black-box testing technique, the SUT is tested with an infinite number of randomized, invalid or unexpected inputs. The input values are generated using fuzz operators. Model-based fuzz testing is used for randomizing sequential or concurrent behavior of systems.

[41] elaborates a tool chain that supports the mapping of the elements of MDA-based testing, where platform-independent tests (PIT) are connected with platform-independent system models (PIM) and platform-specific tests (PST) are connected with platform-specific system models (PSM). They use a subset of UML 2.0 called eUML (essential UML), which itself is extended into the essential modeling language (eTML). Currently, the user creates the system models and test models manually. Transformation rules are applied in order to derive PIT from PIM and PST from PSM.

Rawat and Mounier [88] introduced an evolutionary algorithm that combines genetic algorithm and evolutionary strategies. They use a fitness function, which determines the fitness values of input data using program slicing. Then the malicious program input is generated and executed against the SUT until the program crashes. Afterwards, data about the internal structure of the individuals are collected. With this information, recombining and mutating individuals with the greatest fitness generate better malicious input. The same authors describe another fuzzing approach in [89]. They expand the method of smart fuzzing by using an offset-aware mutation, which mutates string inputs at specific location in order to enhance the used mutation operator using several methods (tainted path generation, source-code instrumentation, mutation and constraint learning and a exploitability metric). The goal is to detect buffer-overflow vulnerabilities.

Duchene et al. [52] use a combination of model inference and evolutionary fuzzing in order to detect non-persistent XSS vulnerabilities of web applications. First, they obtain a model of the system under test by using inference techniques. Afterwards, smart fuzzing is applied by using an evolutionary algorithm.

In the second part, there are several papers dealing with attack modeling, automatic test generation and execution in the context of UML-based and security testing.

Kim and colleagues [72] discuss the usage of UML state machines for class testing. In their approach, a set of coverage criteria is proposed according to the control and data flow in the model and the test cases are generated from the diagram by satisfying these criteria. The authors describe a method on how to transform the state machine into an extended finite state machine (EFSM) that are in turn transformed into a flow graph so common data flow analysis methods and a coverage criteria can be applied in order to generate test cases as a set of paths by using all data from the model.

Kiezun et al. [71] propose a technique for automatic test case generation for XSS and SQLI for testing of PHP applications by using concrete and symbolic execution. They take into consideration persistent cross-site scripting. For this task to be realized, they implemented the tool ARDILLA, which is able to use any input generator. The tool generates inputs and executes these inputs against the SUT. Hence, path constraints are generated and the test case generator can negate one of the constraints, thus eventually producing a solution for the new constraint system. ARDILLA tracks all user inputs through the application in order to find out whether it can lead to a security breach.

Sheyner et al. [96] introduce a technique for automatically generation and analysis of attack graphs for network security testing. First a security analysis of the network is made with the help of scanning tools and already known information so they can model the network in form of a finite state machine and specify desired security properties, which an attacker wants to exploit. The transitions in the model represent potential attacks. The whole network and its attributes may be described in XML. Then, an attack graph is generated using a modified version of the model checker NuSMV so that every path depicts some exploit that leads to an undesired state, which is a state that could be activated by the attacker. After the graph is specified and visualized, additional analysis is done in order to determine likelihood for an attack to occur so that prevention measures can be undertaken. The goal of the attack graph is to cover all possible attacks in the network and also to depict all starting point states from which the malicious user can start her or his activities.

Similar to modeling of attack patterns, Phillips and Swiler [86] use attack templates for network security testing. These templates consist of steps of known attacks but also their preconditions that must occur in order for the attack to be triggered. First they detect attack graphs with a high possibility of being exploitative. They give information about further malicious activity possibilities once a former attack was carried out successfully and also identify consequences of possible linked attacks instead of scanning just for individual ones. Like the above mentioned approach, this method also uses attack graphs but puts a greater emphasis on the view of the attack itself. The authors also analyze defense possibility and simulate attacks along a path from the graph.

Tian et al. [101] concentrate on penetration test cases for web applications by modeling SQLI attacks using security goal models (SGM's) from [42]. First they describe a general view for the attack, then they branch this model further by concentrating on the type of goals the attacker has in mind, e.g. to bypass user authentication. So a bottom-up view is implied by describing the attack behavior. Further the authors propose coverage criteria in order for test cases to cover various attack patterns.

In [103] the author compares several commercial and open source penetration testing tools against several web applications, including Mutillidae, which is also tested in this thesis. The tests were carried out on medium security level by using the application's default settings. The result was that SQLI results do not give any results at all or their number could not be fully determined. The results for XSS were also very poor but better than the ones for SQLI although there were a number of false negatives returned by the tests.

A comparison of vulnerability scanners for SQLI and XSS attacks is given in [57].

Takanen et al. [100] are dealing with a negative testing technique called fuzz testing. A fuzzer is a program or framework which generates random or semi-random and unexpected inputs in order to cause unintended software behavior. In general, the fuzz generator can create complete random and unexpected data, alter already known inputs (black-box) or generate complete systematic test cases according to a known model of the SUT (white-box). One generation-based fuzzer is *Codenomicon* from Codenomicon Ltd.[1], which can be used to fuzz servers, clients and file-processing programs. One of its features is the possibility to send valid input data and make a comparison with all other received data from the past, thus detecting some critical faults.

The authors in [28] use attacker models in form of extended finite state machines and also present the tool VERA, which uses instantiation libraries, configuration files and a XML file for the model. In order to carry out an attack, first an attack is chosen from the library. Then, specific parameters of the SUT are loaded from the configuration file and finally the attacks are executed. Although their paper follows a similar idea to our work, the difference lies in the technical realization, which affects the functionality of the approach. Also, our approach puts a greater emphasis on the XSS attacking process as well as detection mechanisms and make use of the combinatorial testing tool ACTS for test case generation for both SQLI and XSS.

In [104] the Pattern-driven and Model-based Vulnerability Testing approach (PMVT) for detecting of XSS is proposed. The method generates tests for this purpose by using generic vulnerability test patterns and behavioral models of a specific web application. On the contrary, our work does not rely on such models and test purposes but uses different test case generation techniques.

Works that apply threat representations are [105] and [79]. In the first paper the authors adapt Petri nets in order to depict formal threat models. These are represented as Predicate/Transition nets and depict a way how to breach a security leak inside a SUT. Here test cases are created automatically from these models by creating code by relying on the Model-Implementation specification. The second work deals with test case generation from Threat Trees. In this approach, valid as well as invalid inputs are created from threat modeling.

[1] `http://www.codenomicon.com/`. Accessed: 2015-12-28.

Other works that put more emphasis on the aspect of modeling by using activity diagrams include [43] and [65]. Furthermore, [25] deals with the application of state machines.

A detailed overview about SQL injections and cross-site scripting can be found respectively in [45] and [56].

In the next part a flavor is given about the many different application areas of combinatorial testing and the corresponding specialized methodology for test case generation. Its application to the domain of software testing is of current and growing interest.

For a general treatment of the field of combinatorial testing, the interested reader is referred to the recent surveys of [40], [85] and [83].

For example, three different case studies of combinatorial testing methods in software testing have been given in [81]. Applying combinatorial testing to the Siemens Suite and testing with ACTS have been presented in [62] and [31], respectively. A case study for pairwise testing through 6-way interactions of the Traffic Collision Avoidance System (TCAS) has been presented in [90], while in [48] a study was conducted to replicate the Lockheed Martin F-16 combinatorial test study in a simplified manner. Moreover, a proof-of-concept experiment using a partial t-wise coverage framework to analyze integration and test data from three different NASA spacecrafts has been presented in [80]. Combinatorial testing on ID3v2 tags of MP3 files was given in [109] while a case study for evaluating the applicability of combinatorial testing in an industrial environment was presented in [87].

Finally, combinatorial testing has recently been employed as a method to concretize the test input of XSS attack vectors in [34, 61]. In detail, in [34] a novel combinatorial testing technique for the generation of XSS attack vectors was first defined while in [61] the applicability of the previous technique has been further demonstrated by relaxing constraints and modeling white spaces in the attack grammar. Moreover, in [32] fuzz testing approaches for generating XSS attack vectors were compared to combinatorial testing techniques with promising initial results.

The final part comprises research that is based on planning. Automated planning and scheduling is a topic from artificial intelligence that deals with defining ways of problem specification and strategy implementation. Its usual adaptations are in robotics and it is generally used in intelligent agents and only recently for testing. However, its implementation in testing of software is the primary interest of our research.

Planning, that is, finding actions that lead from an initial state to a goal state can be considered an old challenge of artificial intelligence. Fikes and Nilsson [55] introduced STRIPS as a planning methodology for solving this challenge, where planning is performed under certain assumptions and where plan generation is separated from its execution. Later Nilsson [84] introduced a planner for solving dynamic real-world problems where plan generation and execution collapses under one framework. There the author introduces the notion of teleo-reactive (T-R) programs, which are artifacts that direct the execution towards a goal by responding to a changing environment according to the perceived sensor data. Teleo-reactive programs might provide a good base for implementing the behavior of an attacker in the context of security testing. However, in our approach Fikes and Nilssons original work is followed and attacks are defined as planning problems based on STRIPS planning. On the other hand, here lies the main difference in comparison to our approach as well, namely the area of adaptation, that is, web applications instead of robotics. Additionally, our approach also relies on concrete values while the authors' method remains only at the abstract level. Further, our plan generation technique depends upon already existing initial values that are submitted as input files to a planner. This is a more static approach, that is, the plan generation itself does not rely on dynamically encountered data from the outside world, for example from the SUT. However, the teleo-reactive idea could be eventually adapted into our approach as well by incorporating a more dynamically plan generation. The challenge here would be the fact that the initial PDDLs have to be enriched by more corresponding data.

Howe et al. [69] have been one of the first dealing with the use of planning for test suite generation. Besides the test case generator the authors compared their test case generator with another technique using a concrete example, that is, the StorageTek robot tape library command language. The tool Sleuth made use of the UCPOP 2.0 planner for plan generation. The authors divided the approach in three parts: problem description generation, plan generation, and transforming the plan into suitable test cases. If a plan was not able to be generated, another initial and goal states were used in order to find a plan fulfilling the final preconditions. The difference to our work lies in the data from which test cases are generated by the planner. Our program changes the initial values of the problem specification and generates test cases dynamically. Additionally, concrete commands and values are used for the generated plan and execute these against the SUT. In contrast to their work, our approach does not capture the behavior of the SUT but only checks its reactions after submitting the attack vectors.

Scheetz et al. [91] introduced a very much similar work a plan is derived taking into account initial variable conditions and concrete parameter values. Their approach relies on test objective generation from a class diagram and then generates test cases by using a UML model of the SUT. Since our work is primarily a black-box testing approach, it does not rely on a model of the program. Hence, no test objectives or test cases are derived from a graphical representation. Our work encompasses attack inputs, which are generated as part of a combinatorial testing approach. Although their work also uses a concretization phase for plan generation by applying test objectives, our approach relies on already specified methods inside the testing framework. The advantage of the authors' approach is that, by relying on a model of the SUT, testing can be adapted on the concrete application while our method implements a more general idea. Further, it should be applicable on every SUT of a specific type.

Froehlich and Link [59] introduced a method to obtain test suites from mapping of UML state charts to a STRIPS planning problem from which plans can be derived using planning tools. The transformation considers the preconditions of transitions. Despite the fact that the whole test suite generation process is automated, the generation of concrete test cases (considering the specification of the system under test) has still to be performed manually. In contrast to the authors' research, our work does not rely on use cases or UML representations. It relies on a program for test case generation. However, the used model serves as an input for the generation tool and comprises a set of parameters and corresponding values. In such way attack vectors are generated. Then, the resulting test cases are incorporated into the testing framework and afterwards the planner is being called. In contrast to our work, in this approach STRIPS is used for test suite derivation, i.e. before the test cases are being generated. Additionally, the authors still need some manual intervention in order to get test cases, while our work encompasses an automated mechanism for this purpose.

In [78, 77] the authors proposed an automatic contract-based testing strategy that combines planning and learning for test case generation. The work is based on the programming language Eiffel where pre- and postconditions for methods can be easily specified. Those pre- and postconditions can be directly mapped into a planning problem from which abstract test cases can be extracted. In contrast to their method, our plan generation technique differs in many details. First, it doesn't rely on a pre-specified contract or a similar concept but on already specified problem and domain specification. Our obtained plan undergoes a concretization phase inside an already mentioned testing framework. The test oracles also differ from the author's work by incorporating already set up oracles for the detection of SQLI and XSS.

Galler et al. [60] presented similar work. In their paper, they discussed an approach called AIana, that is able to transform Design by Contract^{TM} specification for Java classes and methods into a PDDL representation, which then can be used by a planner for generating plans. The generated plan has to be transformed into Java method calls. Random values are generated for primitive type parameters and recursive calls for non-primitive parameters. The authors also provide two case studies for evaluation purposes. In our approach, the necessary files for the planner are already specified by the tester but will be also generated during program execution. However, in their work an already existing specification first has to be transformed into PDDL. One similarity is the fact that the authors also incorporated an concretization phase by relying on Java method calls. Another difference is that they submit random values for testing purposes while our approach uses CT for that case.

Very similar work includes [94], which elaborates a method for using planning for the generation of test cases from visual contracts (VC). The latter are put into a PDDL representation so that the planning tool LAMA is able to produce a plan. While their work uses the VC before translating it into PDDL and thus generating a plan, our approach does not rely on VCs at all. Since VCs use formalisms with pre- and postconditions, the transition process into a planning problem is more straightforward. However, their approach is similar to ours because of the PDDL representation and the usage of a planner.

Armando et al. [24] analyzed security issues in security protocols using SAT-based model-checking. The authors proposed a method for attacking these protocols using planning. A protocol insecurity problem specifies all execution paths of a protocol, including the possibilities to exploit security leaks, where the entire protocol is depicted by means of a state transition system. The security properties of the protocol are specified using the tool AVISS. The security properties are transformed into an Intermediate Format (IF) and finally, they are read by the model-checker SATMC so that a planning problem can be generated. The problem itself is represented in SAT using Graphplan-based encoding, which is mapped back into a SAT representation in order to produce a solution. In contrast to their work, our approach targets primarily web applications as testing subjects. However, the planning problem, which is demonstrated by the authors, can be adapted on other types of SUTs as well. Additionally, the security specification of the protocol can be inferred only in case that a white-box testing approach is used. This fact represents a problem in our case because our approach relies on a black-box approach, where no information about the SUT is known at all. In this case, the proposal cannot be applied for plan generation. Although their work incorporates more intermediate steps, they are able to generate more SUT-specific test cases, which might improve the vulnerability detection.

The work that preceded our tool PURITY (see Section 6.2) is based either on model-based based testing or planning. However, PURITY puts a greater focus on the technique proposed in [37]. A planner generates a sequence of abstract actions for security testing. On the contrary, for every action there is a corresponding method with concrete values. In such way, concrete test cases are executed accordingly to the abstract plan.

3 Integral Components

Several elements are used in this thesis, for example programs, definitions and overview about tested programs. These will be explained in greater detail in the sections below.

3.1 Definitions

Through the text some basic definitions are referenced to in order to define the functionality of some of the background technologies used in the work. They come either from the field of combinatorial testing (see Section 5.4) or specify the planning problem (see Section 6.1).

Definition 1 *A mixed-level covering array which will be denoted as* $MCA(t, k, (g_1, \ldots, g_k))$ *is an* $k \times N$ *array in which the entries of the i-th row arise from an alphabet of size* g_i*. Let* $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$ *and consider the subarray of size* $t \times N$ *by selecting rows of the MCA. There are* $\prod_{i=1}^{t} g_i$ *possible t-tuples that could appear as columns, and an MCA requires that each appears at least once. The parameter t is also called the strength of the MCA.*

Definition 2 *A tuple* (I, G, A) *is a planning problem, where* I *is the initial state,* G *is the goal state, and* A *is a set of actions, each of them comprising a precondition and an effect (or postcondition). For simplicity it is assumed that each state is given as a set of (first order logic) predicates that are valid within this state. It is also assumed that the preconditions and the effects of an action* $a \in A$ *can be accessed via a function* $pre(a)$ *and* $eff(a)$ *respectively.*

Definition 3 *A solution to the planning problem* (I, G, A) *is a sequence of actions* $\langle a_1, \ldots, a_n \rangle$ *such that* $I \rightarrow_{a_1} S_1 \rightarrow_{a_2} \ldots \rightarrow_{a_{n-1}} S_{n-1} \rightarrow_{a_n} G$.

3.2 Resources

The presented approaches encompass various software resources that make up the functionality of the whole system. Programs that are used in addition to the approaches or as external systems include the following ones:

- **WebScarab** [21]: A Java based framework that reads the communication between programs over HTTP and HTTPS. It can intercept and modify incoming messages and gives information about their content. Similar programs (for example [5]) could be used for this purpose as well.
- **Yakindu** [23]: An open source tool-kit with a modeling environment for the specification and execution of state machines. The tool and its adaptation are elaborated in more detail in Section 5.1.
- **ACTS** [2]: A tool for the generation of combinatorial multi-way test sets according to specified parameters, which in this case is adapted to HTML so valid attack vectors are constructed. It is used in combinatorial testing and is also considered in the area of automated software testing. The adaptation of the tool is demonstrated in Chapter 5. ACTS is developed jointly by the US National Institute Standards and Technology and the University of Texas at Arlington and currently has more than 1400 individual and corporate users.

Java libraries that represent the foundations for the communication between the testing program and the tested applications include:

- **HttpClient** [9]: Implements the client side of the HTTP standard. Used by the program in order to communicate over HTTP with the SUT. Attack vectors are submitted as parts of the request after which the response is read from the tested system so the content from HTTP header and body may be extracted. Using HttpClient offers the advantage to bypass communication via browser and its intern input filters. Additionally, it can also handle incorrect HTML element syntax.
- **jsoup** [11]: A Java based HTML parser. It is used in order to parse the HTTP response in search for critical data after the attack. This is critical for the test oracle so the final test verdict is given according to that information.

Additional software used in the planning-based methodology includes the following programs:

- **Metric-FF** [12]: A planning system from [66] that is based on a forward chaining planner. It handles PDDL files, thus producing a plan.
- **JavaFF** [10]: A Java implementation of FF [67] that parses PDDL files and incorporates a planner. However, PURITY makes only use of the parser in order to extract components like objects, initial values, actions etc. from both the problem and domain definitions.

3.3 SQLI & XSS

As mentioned in Chapter 1, SQLI and XSS can still be found among the most common software exploitation methods despite the fact that several protection mechanisms are already discussed and implemented. For this case, these two vulnerabilities have been chosen for implementation in the framework.

In the following both types will be analyzed in more detail.

SQLI is a malicious method that targets user input elements of a web application that is connected to a SQL database. A SQL statement is injected into these elements in order to retrieve stored data from the back-end database. Usually an injected command is sanitized by the application, however, if the attack vector isn't filtered, the code from the statement is executed. As is mentioned in [45], SQL injections are probably as old as SQL databases are interconnected with web applications. The article [18] from Jeff Forristal is considered to be the first publicly available report about this sort of vulnerability. One attribute of SQL statements is their wide structure that allows attackers to construct injection strings in a very broad way. The usual target data from the database is confidential data about associated people or clients of an organization.

In order to demonstrate the functionality of SQLI, an example from [45] is taken. The application is a retail store where a user can select products by a certain criteria. For example, if only products are selected that cost less than $100, the URL is generated and submitted via the GET mehtod:

```
http://www.victim.com/products.php?val=100
```

The code below shows an excerpt how such an input is passed to a new SQL statement:

```
$query = "SELECT * FROM Products WHERE Price
< '$_GET["val"]' ";
$result = mysql_query($query);
```

The following SQL statement will return all products from the database that are cheaper than $100:

```
SELECT * FROM Products WHERE Price < '100.00';
```

However, an user could alter the URL or submit an SQL statement into an input field as well. For example, he or she could append the code ' OR '1'= '1 to the URL or directly to the SQL statement:

```
http://www.victim.com/products.php?val=100' OR `1'='1
```

The SQL statement will return all products from the database because of the OR and the fact that the query will always return a true:

```
SELECT * FROM ProductsTbl WHERE Price < '100.00'
OR '1'='1';
```

Although the executed code from this example will not result in any harm, it still demonstrates how the user can cause the application to behave in an unintended way, eventually gaining access to secured data.

XSS is defined as a method to force a website to execute injected malicious code. Usually, it is the browser where the attack takes place and it is the user of the browser the attacker tries to target. The code itself is usually written in HTML or JavaScript. Whatever the attacker chooses to do, first he has to detect a XSS vulnerability of the SUT. In order to do this, all client-side inputs on a specific URL address, which values will be sent to the server must be detected. If a leak is found, the hacker has the possibility to inject further malicious code. XSS attacks are commonly categorised as follows, accordingly to [56]:

- *Non-persistent (or Reflected; "Type-1")*: All sent user input data is processed immediately by the server, returning possible results for the request. The attacker might send an e-mail to the victim with a link to a script on the attackers' server and lure him to click on it. By doing this, the client executes the remote script unintentionally, thus activating whatever is coded inside.
- *Persistent (or Stored; "Type-2")*: User input data is stored in a database behind the SUT, so the attacker can inject JavaScript code which becomes stored permanently. The malicious code will be executed every time when some user accesses the application. The attack may cause much more harm than the non-persistent counterpart.
- *DOM-based (sometimes "Type-0")*: Similar to the non-persistent version. The difference lays in the fact that user input data is not sent to the server but remains within the DOM. So this vulnerability comes not from the server but is caused by improper handling of user inputs in the client-side JavaScript. The response from the server does not include the injected script [97].

Let's briefly discuss the basic ideas using a variant of XSS on an example used by Hoglund and McGraw [68]. The example makes use of a web application comprising an HTML page, where a user name can be entered, and a server side script handling a request from the HTML page. Communication between a web browser interpreting the HTML page and the server is performed using standard HTTP where sending information to the server is done using the GET message method. Let's now consider the following (partial) HTML page interpreted using the client side browser:

```
<form action=test.cgi method=GET>
User: <input maxlength=10 type=input name=username>
...
</form>
```

From the HTML code an attacker immediately identifies the existence of a script named `test.cgi` with a parameter `username` that should be allowed to be of length 10. In order to test whether there is a server side limitation of the parameter the attacker might submit the following request to the server (ignoring the web browser):

```
http://to_server/test.cgi?username=TOO_LONG
```

If this request does not lead to an appropriate error message coming from the server indicating a far too long user name, the attacker knows at least that it is possible to submit longer strings. This can be used in the next step of the attack, where the attacker tries to trigger XSS on side of the server by sending the following request:

```
http://to_server/test.cgi?username=../etc/passwd
```

If this request returns the content of the password file, the attacker succeeded. If not, the attacker might try other requests like:

```
http://to_server/test.cgi?username=Mary.log; rm rf /;
cat blah
```

where the whole directory structure is going to be removed in case of success, maximizing the potential damage. There are of course many different strings to be used by an attacker. There are also many ways for reactions coming from the server as well. However, the basic principles are always the same. Every request of an attacker can be seen as potential action having a precondition and an effect. For example, the call `test.cgi?username=TOO_LONG` can be seen as action for testing string boundaries of parameters `bad_bound` with a HTTP address including a server script, the parameter name and a parameter length as preconditions, and the effect that no corresponding error message is returned. The implementation of this action would compose the request string, send it to the server and parse the return value. The effect of this action can be used by another action, for example, the action for testing whether access to a password file is possible and so on.

3.4 Test Oracles

Another important issue of the author's approaches is the identification of security leaks and because of their nature, different methods for both SQLI and XSS have been implemented. Note that specifying the success indicator of an attack for a particular SUT has to be performed only once.

SQLI: This type of attacks target all user inputs. The attack pattern makes use of the precondition that submitted user information (for example via a web form) generates a SQL call to the database, which is not validated. In this case, a malicious user may want to get access to the database by entering SQL queries inside the input fields, thereby asking to read all stored user credentials or information about the system. If the application rejects the SQL string, the attacker may want to slightly change its syntax and submits the input again. The tester repeats this process all until she or he succeeds to bypass all application-intern input filters, thus detecting a security leak inside the software. For the first case it is assumed that the attack is successful whenever the SUT returns a response with retrieved data from a database inside the HTTP body. Since the form of confirmation may be hard to predict, the tester is asked to define an expected value for that case, which usually should be information about a database entry like one of the usernames or passwords. In case the program comes along the same value when parsing the response body, it can be affirmed that the test case was successful.

XSS: In general, XSS in web applications is found upon unsanitized user inputs, that is, if a submitted script is processed unfiltered by an application, then its potentially harmful content is sent to a client and executed, eventually causing damage to her or him. This happens if the malicious code is sent as part of a HTTP response to the victim so a method is needed for the detection of these specific inputs. After reading the response, the obtained HTML or XML code is parsed by using a parser. By using these methods, the whole document is segmented into elements, attributes and values so the entire response message can be extracted by searching for key characters or words. In this way an encountered element, for example *<script>*, *<img>* etc., can be detected and so it becomes obvious that the application is vulnerable to script injections. But if the inserted script is returned in a filtered form, the client might read only some non-executable code. In this case the XSS attack was in vain.

Because a web application might have several scripts as well as other HMTL elements by default, the program first counts the number of already existing elements and then submits the malicious injection. In case of reflected XSS, the response is obtained immediately but in the stored version every time the program submits a new string, all already stored ones are sent back to the user. However, because every time before another input is submitted, a recalculation of occurrence of all elements from the website is done. Because of this, a new malicious input will be always recognized as an additional element in case it triggers an action, thus indicating a vulnerability.

If all goes well, the responded HTML code is parsed with the help of jsoup in search for additional elements and if one is found, a successful XSS leak is confirmed. Because of this fact, the goals of the proposed approaches are to inject such a string that will go unsanitized through the application and to detect whether it was manipulated when parsing the response. In general, both types of XSS can be detected the same way.

3.5 Systems Under Test

The tested SUTs in this thesis comprised a set of web applications that are included in the Open Web Application Security Project (OWASP) Broken Application Project[1] and in the Exploit Database Project[2]. Many of these programs are primarily meant for testers to practice their exploitation skills by implementing several security levels, which become even harder to breach.

All of them have been locally deployed and tested. For the case of efficiency measurement of the testing methods, some of the SUTs were tested several times. In such way a conclusion could be drawn about the efficiency of the corresponding testing technique and technology.

[1] https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project. Accessed: 2015-12-28.

[2] http://www.exploit-db.com/. Accessed: 2015-12-28.

These testing applications are namely:

- NOWASP (Mutillidae) [13]
- Damn Vulnerable Web Application (DVWA) [6]
- BodgeIt [4]
- WebGoat [17]
- Gruyere [8]
- Bitweaver [3]

The other applications are blog software:

- Wordpress [22]
- Anchor CMS [1]

DVWA and Mutillidae include more security levels that can be selected by the tester. Testing on each of these levels will differ with regards to built-in filtering mechanisms, which become even more restrictive with each higher level so these applications had to be tested against all of them. The other applications, on the other side, do not have any additional security features.

Blogs are very popular attack fields for stored XSS as well. For this case, the known Wordpress blog application was tested as well as Anchor CMS.

All of the SUTs were tested for SQLI and XSS, with every one of them being checked manually before the actual testing. The fact that DVWA and Mutillidae also offer the possibility to be checked against additional types of attacks was ignored. In the other applications only the second type of XSS is tested because these are blog software, where posts are usually stored and kept inside a database. All applications have been deployed on an Apache Server and comprise a MySQL database.

4 Model-Based Security Testing

Model-based testing is an active testing methodology with the objective to generate test suites from models of the SUT. When using models for test suite generation it can be guaranteed that the test suite is complete with respect to the given model. Several approaches of model-based testing to security testing are discussed and it is argued that this methodology is very beneficial for this purpose in order to ensure quality constraints.

The current research from this area offers several methods and solutions in order of how to formalize and implement testing techniques that are able to detect potential security leaks in programs. These methods differ accordingly to the initial problem statement. For example, if the source code of the SUT is unknown, black-box techniques like fuzz testing are the first choice. Fuzz testing is an optimizing random test case generation method that also makes use of underlying models like communication process models. On the other hand, if a developer wants to test her or his own implementation having complete insights of the source code, white-box testing methods may be applied.

The main function of this approach is to invoke unintended behavior of the program, thus reporting a potential error, which might correspond to a vulnerability. So the whole process is actually an offensive thinking approach on testing, which is also known as ethical hacking (white hat), realized in order to find out how a non-ethical hacker (black hat) may proceed in his intention to cause damage to the client.

Even if the focus is not in detecting new security issues, testing applications regarding known security holes is important in order to prevent applications to be obviously vulnerable. The introduced approach utilizes models like state machines or activity diagrams and also makes use of mutation testing. The purpose is to give an overview of the area of model-based security testing and to compare different approaches with respect to their ability of fault detection in the context of security properties. The introduced approaches vary from the use of models, for example models of the SUT versus models of attack behaviour, and their fault detection abilities, for example is it possible to detect new vulnerabilities or not. Here it is worth mentioning that vulnerability per se might not lead to a successful attack. Some of the faults might still not be detected either because of shortcomings regarding the system's model or combinations of test inputs that make use of unintended interactions between the SUT and its environment. The latter is of special interest in security testing because in many cases security holes are due to unrealistic assumptions regarding interactions and the involved data. In order to prevent from exploitation of security holes in software, several techniques have been proposed, including better educating programmers to attract more attention towards security, and program analysis [68].

Three approaches for the problem of testing SQLI vulnerabilities in a given web application are explained. The task here is to provide a method for reading stored data from a database behind the given application using SQLI attacks in order to fetch stored passwords.

The content of this chapter is taken from the author's work in [35].

4.1 Approach overview

In this example, it is assumed that the traffic between the tester and the SUT is already recorded and variable names are extracted. The focus lies on variables for the username and password and the discussion starts with the model-based fuzzing approach. Afterwards evolutionary fuzzing is explained and finally an approach that makes use of mutations.

4.1.1 Model-Based Fuzzing

Fuzzing is a testing technique that generates random or semi-random inputs for a SUT in order to test for vulnerabilities. This is done through a fuzz generator [100]. Although fuzzing is divided into several categories, it still

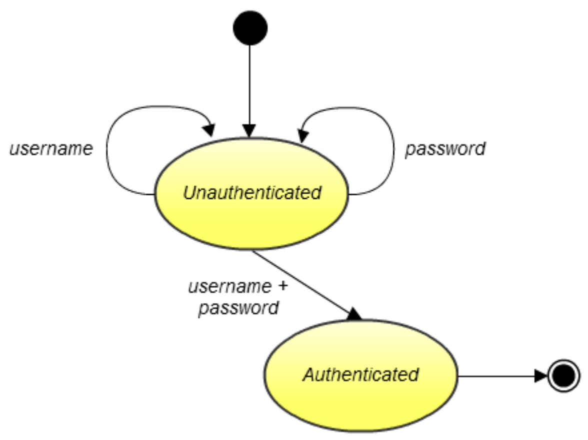
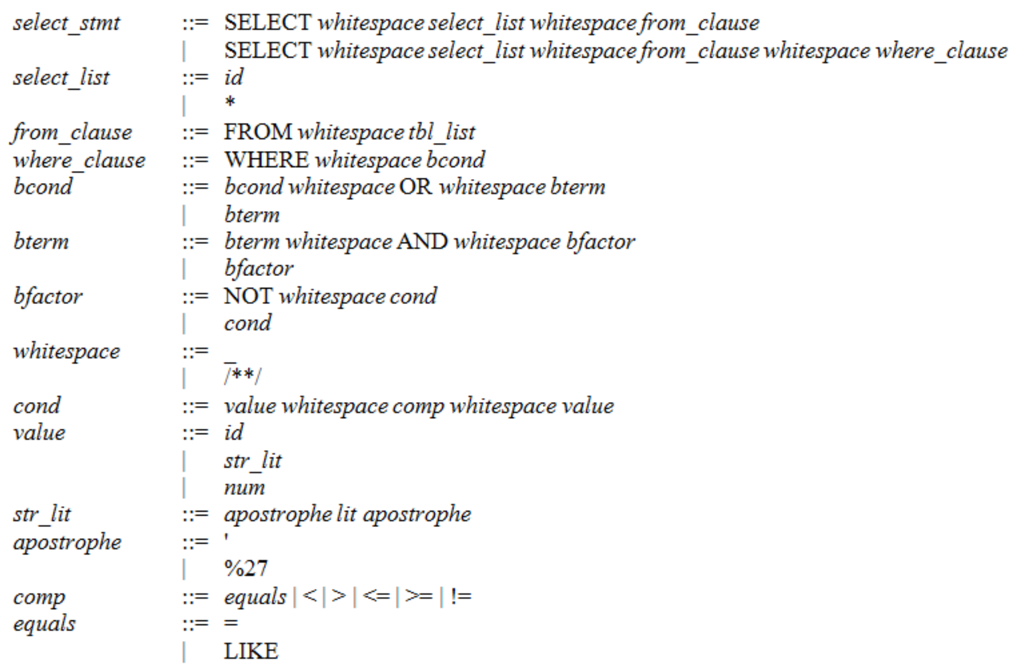


Figure 4.1: Simple authentication model

consists of just vague methods, not only because of the absence of one generally agreed definition. Fuzzing can be divided into three test case generation methods [26]: *Random fuzzing* was the initial fuzzing method. It just creates complete random input without any special guiding strategy or information about the SUT. *Mutation-based fuzzing* is a black-box testing method, which alters already known input data, for example by manipulating some parts of the input to becoming invalid. Although mutation-based fuzzing has better results than random fuzzing, it depends very much on the available information of the SUT. *Model-based fuzzing* uses a white-box testing approach where a model of the SUT must be known in order to generate input data. In this way it relies completely on the given structure and behavior of the SUT and generates systematic test cases with no randomness [99].

The model-based fuzzing approach is discussed by using a running example. First, a closer look is taken at a partial model of the SUT. Figure 4.1 depicts a model of a simple authentication mechanism. If just one of the mandatory inputs is submitted, the user remains in the initial state but by entering valid values for both variables, the next state can be entered, which gives access to the stored data. The goal of a SQLI fuzzer is to modify a part of the structure of a SQL statement as a new input, without violating its existing syntactic validity.



select_stmt	::=	SELECT *whitespace select_list whitespace from_clause*
	|	SELECT *whitespace select_list whitespace from_clause whitespace where_clause*
select_list	::=	*id*
	|	*
from_clause	::=	FROM *whitespace tbl_list*
where_clause	::=	WHERE *whitespace bcond*
bcond	::=	*bcond whitespace* OR *whitespace bterm*
	|	*bterm*
bterm	::=	*bterm whitespace* AND *whitespace bfactor*
	|	*bfactor*
bfactor	::=	NOT *whitespace cond*
	|	*cond*
whitespace	::=	_
	|	/**/
cond	::=	*value whitespace comp whitespace value*
value	::=	*id*
	|	*str_lit*
	|	*num*
str_lit	::=	*apostrophe lit apostrophe*
apostrophe	::=	'
	|	%27
comp	::=	*equals* | < | > | <= | >= | !=
equals	::=	=
	|	LIKE

Figure 4.2: Grammar for SQL statements

A grammar-based white-box fuzzing method is adapted from [63]. The fuzzing technique relies on its specific fuzz generator, which itself can use various fuzzing strategies. The generation fuzzing engine must have a template or other form of input vectors, which acts as a provider of input data for the generator. In this case, the data description is a grammar, which has to capture all standard SQL parts but also special signs or characters so that input filters can be evaded. An adapted version from [98] is shown in Figure 4.2.

In this way, the generator is fuzzing SQL based tokens. Figure 4.3 shows the generated grammar-based parse tree.

The authors of [64] give a description of their approach. First, a predefined input is symbolically executed against the SUT, thus creating constraints on inputs of the program from conditional statements during their execution. Then, the collected constraints are negated and processed in a constraint

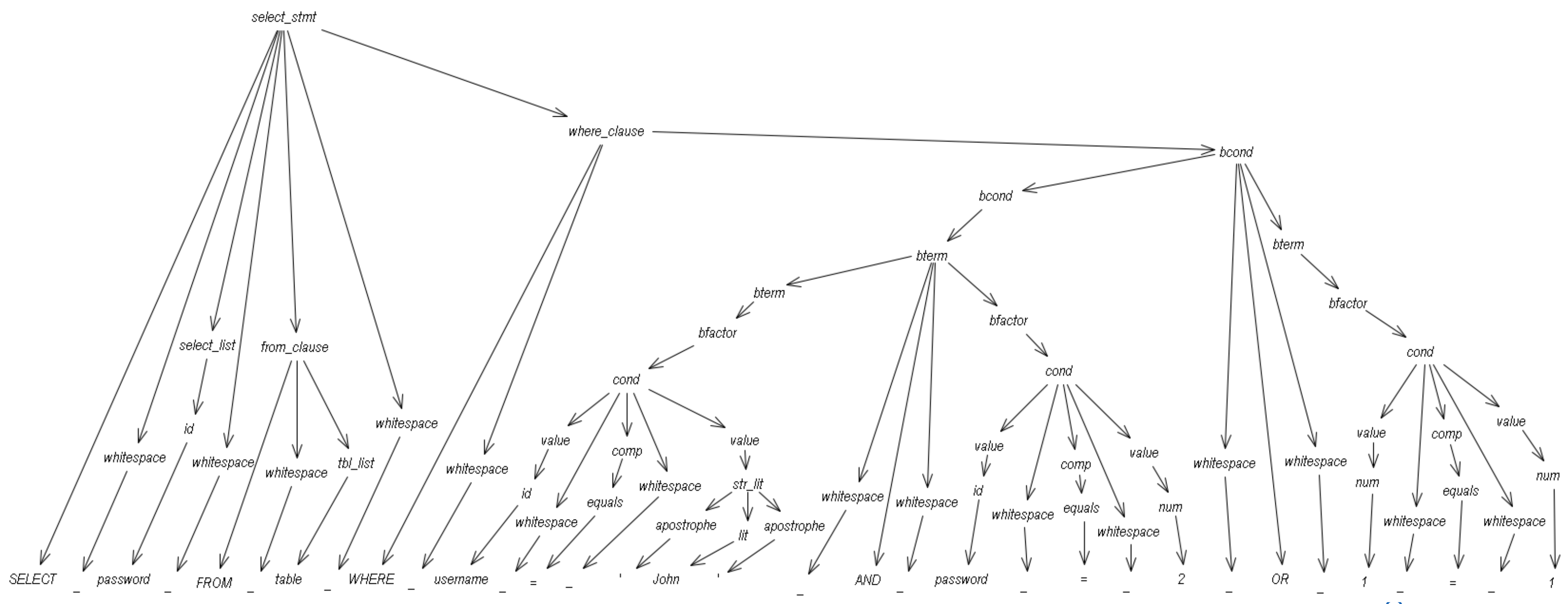


Figure 4.3: Parse tree for a generated SQL injection

solver, thus defining new inputs for the tested application. In the authors' case, symbolic values are tokens. Conditional statements define input filters. Let's consider a vulnerability from [44] where the following filter is used:

```
if (stristr($value, '=') || stristr($value, ''))
die('Insert valid input');
```

This means that all equal and white space signs are filtered from the input. Now the SUT (which contains the input filters) is run on the valid with the fixed starting input:

```
SELECT password FROM table WHERE username='John'
```

By doing so, the following sequence of symbolic token constraints and will be defined according to the grammar (just an excerpt is shown in order to keep the explanation short):

$\texttt{token}_0 = \texttt{id}$
$\texttt{token}_1 = \texttt{whitespace}$
$\texttt{token}_2 = \texttt{=}$
$\texttt{token}_3 = \texttt{whitespace}$
$\texttt{token}_4 = \texttt{apostrophe}$
$\texttt{token}_5 = \texttt{lit}$
$\texttt{token}_6 = \texttt{apostrophe}$

Now the constraint in the path is negated, which corresponds to the given input filter from the SUT:

$\texttt{token}_0 = \texttt{id}$
$\texttt{token}_1 = \texttt{whitespace}$
$\texttt{token}_2 \neq \texttt{=}$
$\texttt{token}_3 = \texttt{whitespace}$
$\texttt{token}_4 = \texttt{apostrophe}$
$\texttt{token}_5 = \texttt{lit}$
$\texttt{token}_6 = \texttt{apostrophe}$

The grammar-based constraint solver will now try to satisfy the constraint and conclude that the keyword LIKE is the only solution, which does not change the structure of the statement. So the new test case will look like:

```
SELECT password FROM table WHERE username LIKE 'John'
```

The next condition is responsible for filtering all white spaces, which can be replaced with the comment signs. So by applying the same method to the last generated input and satisfying the new constraints, the following statement is obtained:

```
SELECT/**/password/**/FROM/**/table/**/
WHERE/**/username/**/LIKE/**/'John'
```

The process will continue as long as new input grammar-related combinations can be used. If the application uses another input filters, for example does not allow SQL keywords like SELECT, FROM etc., the grammar can be adapted in such a way so that a statement like the following can be created:

```
SEL/**/ECT/**/password/**/FR/**/OM/**/table/**/
WHE/**/RE/**/username/**/LI/**/KE/**/'John'
```

Another form of filter evasion may be URL encoding, which replaces a potentially dangerous character with the hexadecimal form of their specific ASCII code. For example, in order to URL-encode the apostrophe symbol, its representation can be taken, which is %27, and generate the query:

```
SEL/**/ECT/**/password/**/FR/**/OM/**/table/**/
WHE/**/RE/**/username/**/LI/**/KE/**/%27John%27
```

Another interesting idea from the authors from [44] is to URL-encode the URL signs, for example to URL-encode the percentage sign at the beginning, which results in the alternative expression %25:

```
SEL/**/ECT/**/password/**/FR/**/OM/**/table/**/
WHE/**/RE/**/username/**/LI/**/KE/**/%2527John%2527
```

Double-, triple- etc. encodings may be also applied on other input characters, thus getting a very huge number of possible input patterns. However, because the tester already knows the model of the SUT and maybe he or she also knows all used filter mechanisms, a qualitative fuzzing approach can be defined. But there is still the problem that a big application requires formulating a huge model. Beside the required resources for writing the model there is always the question how the quality of the model reflects on the quality of model-based fuzzing in general.

4.1.2 Model Inference Assisted Evolutionary Fuzzing

Duchene et al. [52] propose a black-box testing approach by combining the usage of model inference and a genetic algorithm (GA) in order to exploit potential cross-site scripting (XSS) vulnerability of a given SUT. First the model of the application is being figured out by using techniques developed in [19]. Then, using an evolutionary algorithm (EA), various inputs are automatically generated and tested against the inferred model. They define an attack grammar G_{AI} which represents attack inputs that have been already used for testing of other SUTs and feature also random sequences.

Given an initial population of individuals, which represent various input parts, an evolutionary algorithm is used to crossover and to mutate the individual inputs. These operations can also be done on individuals from one or between several population pools like in [50]. To check the fitness of new individuals from the population, a specific fitness function $Fit(I)$ is used, which calculates the likelihood of an individual to exploit a potential vulnerability. The function itself is characteristic for the type of exploitation.

Here the same approach is used in order to detect and exploit possible SQLI vulnerabilities. First the model of the SUT is inferred by model inference. Therefore the mentioned techniques from [19] may also be applied again. By

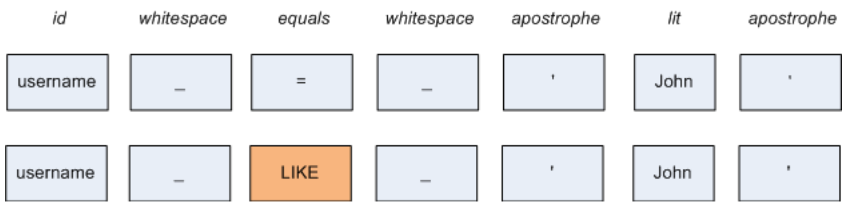


Figure 4.4: Statement mutation-1

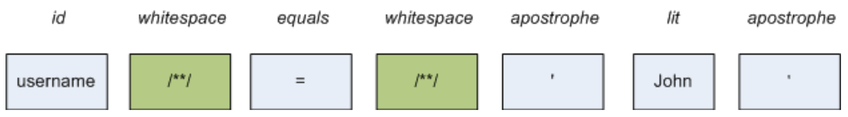


Figure 4.5: Statement mutation-2

doing so, the depicted model from Figure 4.1 is obtained, thus generating all state transitions. Then the entire system is navigated through and searched for sinks, that is dynamically constructed SQL statements or user-controlled input (username, password), on the outputs of all given states. In this case, all transitions contain user inputs but only the valid combination of both leads to the next state. So, fuzzing is started from the initial state using both inputs.

The grammar from Figure 4.2 is also used for generating test cases. Now the first SQL input statement is created by applying input sequences, which were obtained during inference. In order to successfully test the given SUT, all locations within a program should be reached, which means also that all conditionals must be satisfied.

A mutation is defined as either adding or replacing one or more parts of a statement. In order to keep the example short, the equals sign from the conditional token is mutated, which is shown in Figure 4.4.

In this way, the same SQL input is obtained as in the model-based fuzzing example. By mutating all white spaces from the initial statement, the statement from Figure 4.5 is inferred.

Now, by applying crossover, both statements are combined forming the injection from Figure 4.6, thus forming an input, which is likely to fulfill both input filters.

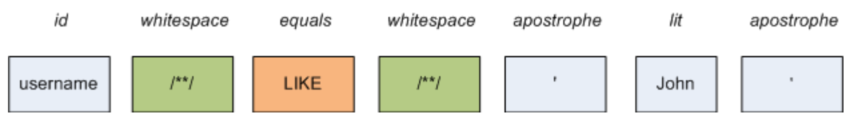


Figure 4.6: Statement crossover

The fitness function $Fit(I)$ evaluates all input sequences, thus calculating the fitness of the individuals. The ones with higher values then form the new population pool and are used to form new test cases.

In this way, several test cases can be combined and crossed, thus generating a set of different input sequences for the SUT. These processes are continued as long as input filters reject the submitted data.

4.1.3 Model-Based Mutation Testing

This testing approach shares similarities with the other two described methods. In particular, all of the approaches use models. However, the author's work uses a model of the attack directly to generate the test cases. In the proposed approach a general attack plan for exploiting application vulnerabilities using one special hacking technique is used. It relies on SQL injection as an example, which includes input queries but also alternative attacking directions and bears all known types of filter techniques. All test cases are created according to that scheme, thus minimizing the level of randomness (at least as long as all model specifications are not exhausted), that is the generation process of test sequences should be directed as much as possible.

However, because of the fact that application programmer may define custom filters, it might be difficult to model all possible attack variations. Figure 4.7 depicts a short abstract model of this sort of exploitation from the very beginning (in order to keep the model simple, it is minimized only to show a few steps and only for a MySQL database). Also, the model shows specific SQL instruction syntax as inputs for the SUT.

In this example the test cases start with the generation of injection sequences, which ask for the technology behind the application by grabbing its banner.

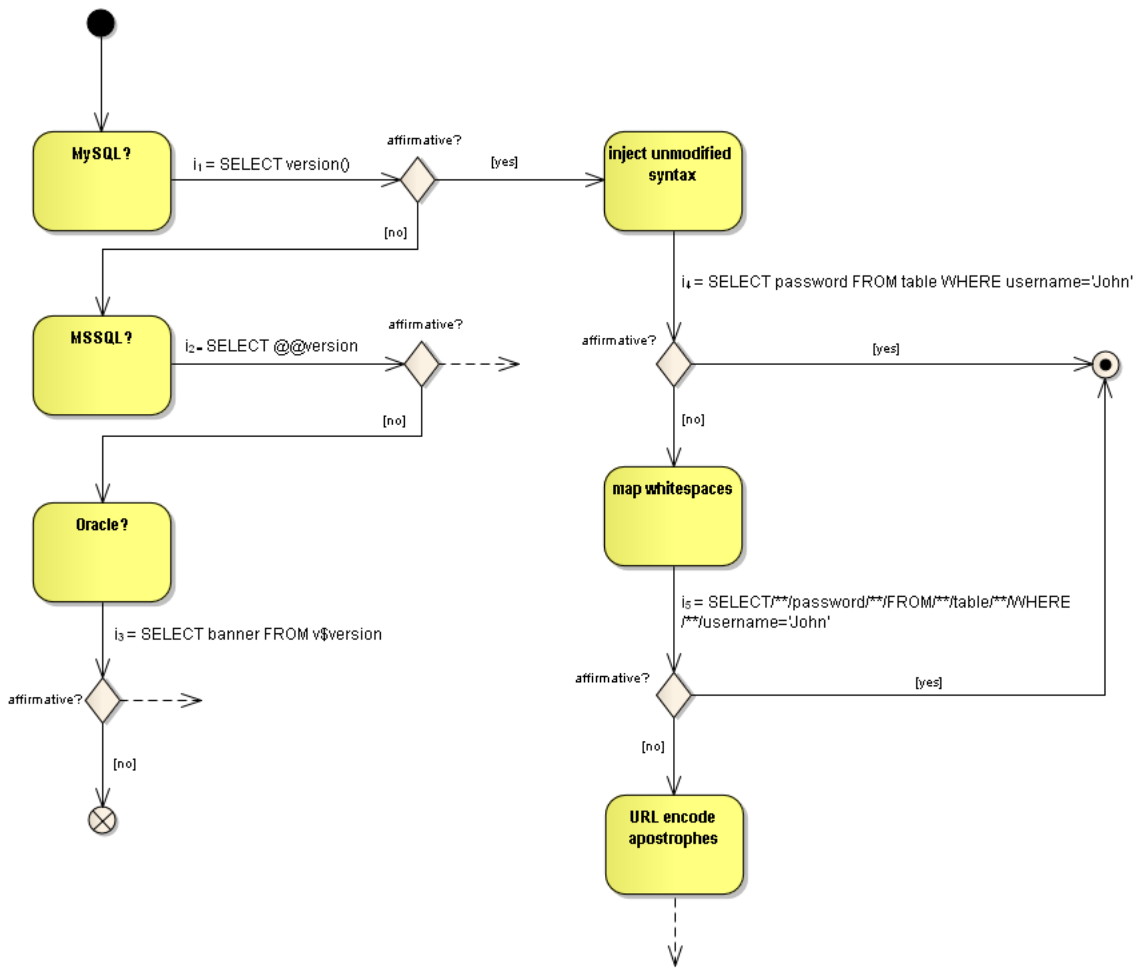


Figure 4.7: SQL injection attack model

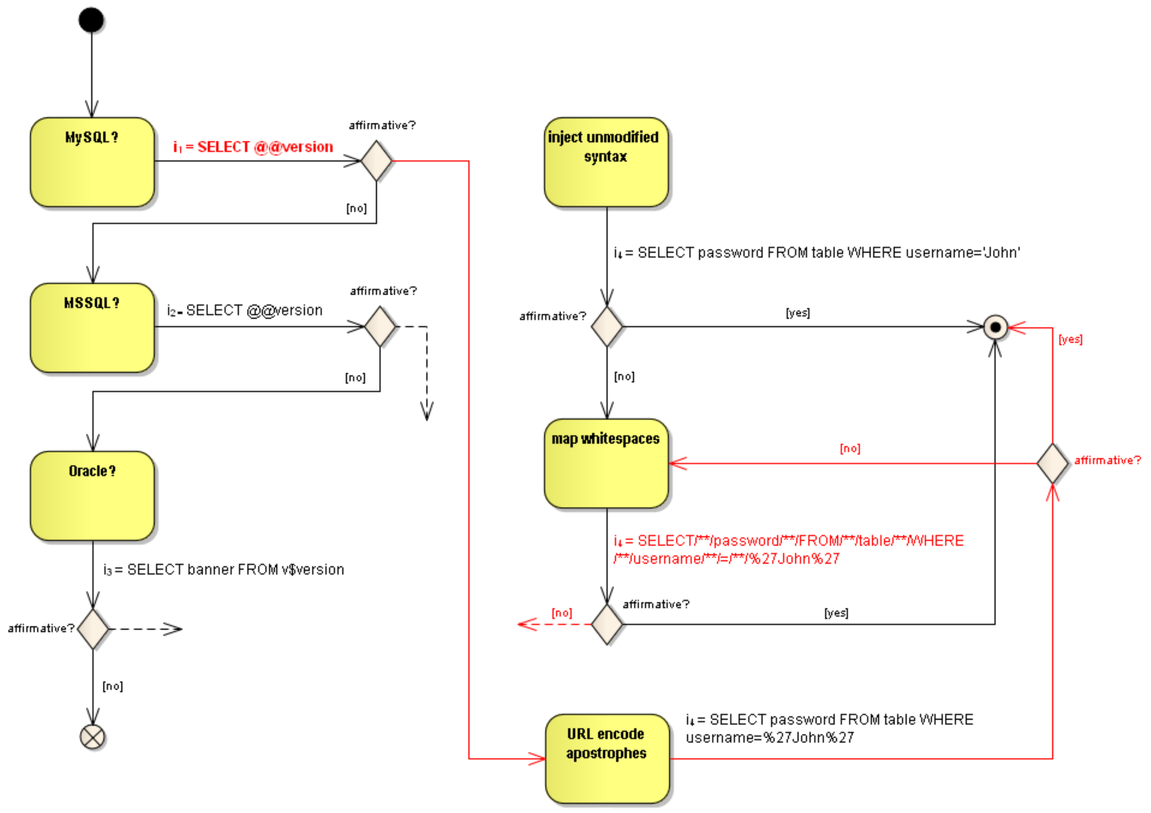


Figure 4.8: Mutated attack model

This is very important because all further used SQL statements depend partially on the used database system (MySQL, MS SQL Server, Oracle). The used database can be identified using pre-existing global variables. If no affirmative message is obtained, the next test cases will test for another database. The idea behind this method is to mutate the attack model, either by changing parts of SQLI statements or the order of the execution. A possible collection of mutations is shown in Figure 4.8. If the generator starts again to create test cases, different SQLI attacks will be tested against the SUT. As can be seen in the mutated model, the order but also the values in Figure 4.8 differ from those in Figure 4.7. By applying this scenario, a vulnerability may be triggered, which remained uncovered during the first generated set of tests. Similar to white-box testing, this approach relies entirely on the complexity and reliability of a pre-existing attack model.

4.2 Summary

Here some new ideas concerning the implementation of several model-based testing techniques have been shown in order to successfully cover malicious SQL injections in web applications. Although some other proposes exist, the adaptation of these techniques is still an area with a high demand for further scientific research. Also, considerations should be given to make this topic an interest for the industry. The intention here has been to examine the presented approaches to cover practical problems but also to examine further fundamental research. The same techniques may also be applied for other security breaking and vulnerability triggering methods.

5 Attack Pattern-Based Combinatorial Testing

In order to make test automation possible, all attack vector information must be gathered and structured in one single representation. For this sake the use of attack patterns is proposed, that is, methods that describe all pre- and postconditions, attack steps as well as the expected implications for an attack to be carried out successfully. A pattern is formalized by using the UML state machine modeling language [14] and the whole testing system is integrated into the Eclipse development environment.

In this approach an attack pattern model is executed against the application, reporting a positive or negative feedback. Further, a framework is provided for testing and detection of vulnerabilities in web applications. The framework comprises two parts: a model-based testing method that is based on attack patterns, and combinatorial testing for generating attack vectors to be submitted against the SUT. Here the idea is to use model-based testing for generating abstract test cases, for example, sequences of necessary interactions, and combinatorial testing for concretizing the input. With combinatorial testing different combinations of inputs are captured and thus the likelihood to find weaknesses in the implementation under test is increased.

Afterwards, the scope of the research is changed and focused on comparing, for the first time, the efficiency of the IPO-family algorithms when the generated test inputs for vulnerability detection in web applications. Most important, an extensive analysis is presented in terms of (total) vulnerabilities found by the generated test suites using the combinatorial coverage measurement (CCM) tool [51], where it is revealed that the indicated security leaks are caused by the interaction of a handful parameters [33].

For generating comprehensive and sophisticated testing inputs, input parameter modeling was used by adding constraints between the different parameter values. In addition, using constraints during the modeling of the attack grammars results in an increase on the number of test inputs that cause security breaches. Finally, a state-of-the-art manual testing tool has been compared to the presented automated approach.

In this chapter answers are given for several of the asked research questions from Chapter 1, namely RQ1, RQ2 and RQ3.

The preceding work, which content was taken for this chapter was published in [36, 38, 34, 33, 32]. In these works, the main focus of the thesis' author was based around model-based testing. On the other hand, the partners from SBA Research put their emphasis on combinatorial testing and were also responsible for test execution and evaluation with Burp Suite in Section 5.9.

5.1 Attack Patterns

In order to model attack patterns, the proposed work follows Moore and colleagues' [82] definition rules of attack patterns comprising the major segments: Goal, Preconditions, Actions, and Postconditions. In the following the general information is outlined, which is necessary to carry out a SQLI or XSS attack:

Goal: Exploit software vulnerability in order to access sensitive data or to cause damage.

Precondition(s):

Tester can submit data into application.

Actions:

1) Connect to the application via its URL by communicating over HTTP.

2) Generate and submit malicious input.

3) Compare output with expected values. (For XSS: Detect unexpected HTML elements.)

Postcondition(s):

Tester accesses obtained information or causes harm to the client.

It is worth mentioning that a model of the particular attack pattern has to implement the defined actions under the assumption that the precondition can be ensured.

In the author's case, the purpose of the model is not to verify the behavior of the SUT so the application instead of the attack pattern would be modeled. Hence, trying to achieve state or transition coverage of the model during testing does not cover any coverage metrics on side of the tested program. Here the emphasis is put on specifying an attack model which is executed against the SUT. Moreover, it is worth mentioning that the model itself should be as general as possible in order not to restrict its use in practice, that is, it should be able to cover a wide range of web applications and would demand only minor user inputs in order to adapt the pattern on another SUT.

For the sake of modeling the attack model, the open source tool-kit YAKINDU Statechart Tools (see Section 3.2) is used. It consists of an integrated modeling environment for the specification and development of systems based on state machines. The UML state machine is used to specify and model all necessary information in order for executing the model automatically.

The following data is defined within the YAKINDU editor:

- *interface* - define the communication points. All variables, operations and events are defined as their parts.
- *variable* - specify variables used during model execution. They represent an essential part because they are used within transitions in guards and actions.
- *operation* - are used in order to specify methods, which may be called either from within the states or as parts of transitions.

The state machine is defined as an aggregation of states, variables, methods, guards, actions and transitions. Every state represents the current situation of execution with transitions being the connections between states. Also, a state may have several transitions, which represent alternate paths of execution and they can also have entry or exit actions. A transition is defined with a guard and one or more actions where the first represents one or several conditions on variables which must occur in order for the action to take place. The action is usually an operation and is responsible for activating the next state of the model, which -when entered- activates all its entry actions. However, the definition of entry or exit operations is not mandatory. Also, guards and actions may not be specified at all, the next state will always be activated without conditions or data manipulation. Choices are also added into the model, which -when encountered- offer at least two transitions. According to which conditions is satisfied, different states will be activated.

One important fact of the proposed approach is that the model covers just the attack pattern and not the inner structure of the SUT. In contrast to other model-based testing techniques that use a correct version of the application and compare the output of the real program with the expected one derived from the model. The functionality of the SUT is not tested at all but the concentration lies entirely on the test case execution for vulnerability detection.

5.1.1 Modeling Attack Patters

In order to model attack patterns a modeling language is required, which would be suitable for visualizing attack patterns in order to depict a sequence of interactions with the SUT that could lead to a software exploitation on side of the SUT. Such a language has to be easy to learn and use. Therefore, UML state machines (UML SMs) [15] are adapted due to its heavy use in practice for this purpose. The main concept of UML 2.0 state machines are states and transitions, which in the author's case hold all the needed information used for computing attacks based on the modeled attack pattern.

In Figure 5.1 an example of the UML model of the attack pattern for SQLI and XSS attacks is depicted for one of the SUTs. As can be seen, the depiction comprehends additional elements like variables and method calls as well. In this model an attack starts by specifying a web address of the SUT from which the relevant information is going to be extracted. In the state `Parsed` different attack schemata are applied whenever this state is invoked. The calculation of the attack schemata is performed using methods that are executed after leaving the state. This attack pattern model is discussed in the case study section. Before that the basic entities of the model are introduced, which are needed for describing such an attack pattern.

For the purpose of adapting UML state machines to handle attack patterns, the entities variables V and methods M are introduced, that are used for extracting necessary information as well as for constructing concrete test cases. It is assumed that methods from M are implemented in a programming language like Java. Every method might change the value of a variable M. It is also assumed that guards and actions used in transitions are also written in the same language. The value of a guard has to be of type Boolean. Guards are not allowed to change variables. Actions itself are programs that might change variable values or cause method invocations.

Formally, state machines Σ are defined as follows:

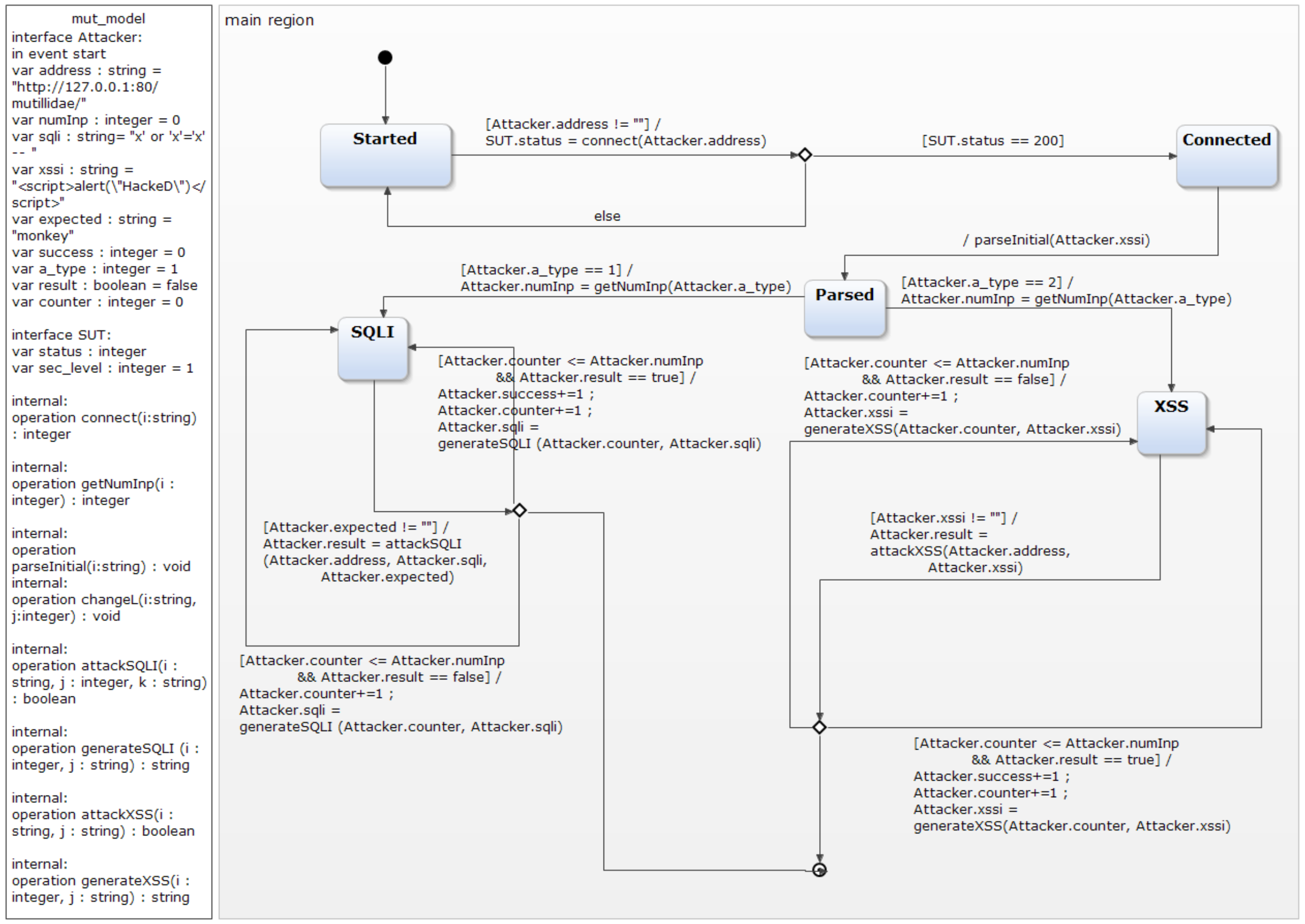
$$\Sigma = (S, V, M, G, A, T, s_0, F)$$

where

- S is a finite, non-empty set of states.
- V is a finite set of variables over a predefined domain.
- M is a finite set of methods that might change the value of variables.
- G is a set of guards, that is, program fragments that return a Boolean value. Guards are not allowed to change variable values.
- A is a set of actions, that is, program fragments that change the value of variables.
- T is a set of transitions, that is, $T \in S \times S \times G \times A$, connections between states with a corresponding guard and an action.
- $s_0 \in S$ is the initial state.
- $F \subseteq S$ are the final states.

Now the different parts of a state machine will be explained in more detail.

Note that in UML a state may be extended in order to become a composite state by adding one or more regions, that is, containers that contain states



```
mut_model
interface Attacker:
in event start
var address : string =
"http://127.0.0.1:80/
mutillidae/"
var numInp : integer = 0
var sqli : string= "x' or 'x'='x'
-- "
var xssi : string =
"<script>alert(\"HackeD\")</
script>"
var expected : string =
"monkey"
var success : integer = 0
var a_type : integer = 1
var result : boolean = false
var counter : integer = 0

interface SUT:
var status : integer
var sec_level : integer = 1

internal:
operation connect(i:string)
: integer

internal:
operation getNumInp(i :
integer) : integer

internal:
operation
parseInitial(i:string) : void
internal:
operation changeL(i:string,
j:integer) : void

internal:
operation attackSQLI(i :
string, j : integer, k : string)
: boolean

internal:
operation generateSQLI (i :
integer, j : string) : string

internal:
operation attackXSS(i :
string, j : string) : boolean

internal:
operation generateXSS(i :
integer, j : string) : string
```

main region
Started
[Attacker.address != ""] / SUT.status = connect(Attacker.address)
[SUT.status == 200]
Connected
else
/ parseInitial(Attacker.xssi)
[Attacker.a_type == 1] / Attacker.numInp = getNumInp(Attacker.a_type)
Parsed
[Attacker.a_type == 2] / Attacker.numInp = getNumInp(Attacker.a_type)
SQLI
[Attacker.counter <= Attacker.numInp && Attacker.result == true] / Attacker.success+=1 ; Attacker.counter+=1 ; Attacker.sqli = generateSQLI (Attacker.counter, Attacker.sqli)
[Attacker.counter <= Attacker.numInp && Attacker.result == false] / Attacker.counter+=1 ; Attacker.xssi = generateXSS(Attacker.counter, Attacker.xssi)
XSS
[Attacker.expected != ""] / Attacker.result = attackSQLI (Attacker.address, Attacker.sqli, Attacker.expected)
[Attacker.xssi != ""] / Attacker.result = attackXSS(Attacker.address, Attacker.xssi)
[Attacker.counter <= Attacker.numInp && Attacker.result == false] / Attacker.counter+=1 ; Attacker.sqli = generateSQLI (Attacker.counter, Attacker.sqli)
[Attacker.counter <= Attacker.numInp && Attacker.result == true] / Attacker.success+=1 ; Attacker.counter+=1 ; Attacker.xssi = generateXSS(Attacker.counter, Attacker.xssi)

Figure 5.1: UML based attack model in Yakindu

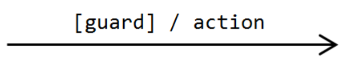


Figure 5.2: Label of a transition

or transitions. A composite state has nested states (substates) inside itself. By applying redefinition to the original state, it applies to the entire state machine [29]. However, in the proposed case composite states are not used. Hence, a formal definition is omitted here.

The set of variables V might comprise local or global variables. Local variables are only used in a particular state. Variables are used by methods M to undertake necessary data manipulation. Variables may be also involved as part of the expressions or statements used in guards or actions. As already said in case of guards variables are only referenced but never defined. In classical state machines usually there is also a set of events. Events are usually triggered from actions or external events [72] and are used in guards.

Although defining events may be useful in some cases, the author's approach implements an automatically execution, so events are left out from the case study. Note that this assumption does not really restrict modeling. In the proposed case events can be replaced by Boolean variables. Instead of triggering an event, a method or action can set its value to true and thus allowing a guard to become true afterwards.

Transitions (see Figure 5.2) are connections between states. Using a transition, control can pass from one state to another. In order to control the flow, each transition has a corresponding guard and an action.

A guard from G represents a firing condition and also takes the role of a constraint for an action to be triggered. They restrict values or demand a specific expression to be satisfied before allowing the state to be traversed under consideration of given variables, time-constraints etc. If no guards are specified, then it is always considered as satisfied.

An action from A comprises a program fragment to be executed before reaching a target state from the source state. An action can be whatever the programmer defines, for example method calls or definitions of variable values. Note that within an action, statements are executed sequentially in the pre-defined order. Moreover, it is possible to specify no actions at all.

Hence, the semantics of a state machine can be informally defined as follows. Start with the initial state s_0. If there is a transaction from the current state where its guard evaluates to `true`, move to the new state. When reaching a final state the state machine successfully terminates. Otherwise, search again for transitions to move from one state to another. In order to keep the state machine deterministic, it is assumed that whenever a state s is active, there is only one transition where its guard evaluates to `true` that goes from s to another state.

After specifying the modeling language, the testing process is discussed by the using attack patterns approach in Section 5.3.1 in a case study. As already said, the approach relies on a model of attack patterns. A test case is a path through the state machine model. In the author's case state machines contain cycles in order to ensure that different test cases are executed. A final state either refers to a case where an attack was successfully carried out, or when the number of potential test cases is exhausted. In the latter case it is clear that the attacks modeled can be handled correctly by the SUT. This does not mean that there are no security issues left.

5.2 Component Interaction

In order realize the testing method of the approach, the communication of several components has to be established in the testing framework. First of all, an overview is needed about the various key components.

The framework WebScarab was used for the interpretation of the communication between a client and a web application. It acts as an intercepting proxy and reads and analyses the message traffic between a user and the web application on a server (that can be also employed locally).

Another important part of the framework represents the already mentioned Java methods that were implemented manually in the first place. Each of them executes a specific task and eventually returns a new value. Method

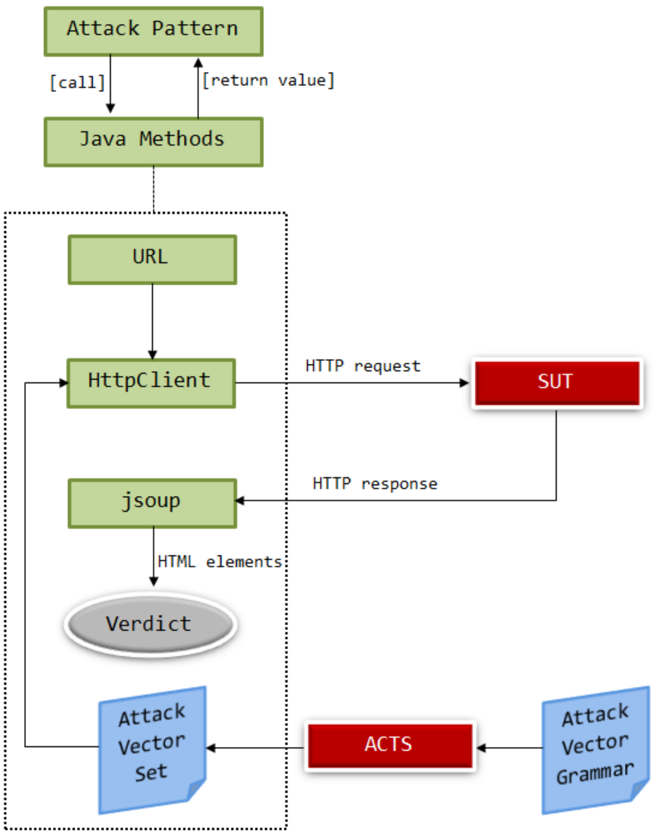


Figure 5.3: Test creation and execution framework

calls are defined as parts of the transitions in an attack pattern, usually as postconditions. During test execution, the methods are called while traversing the model, eventually returning new values for specific variables. Now, these new values may represent a new precondition in the model for further processing. A transition in the model might be traversed several times during model execution so some methods might be executed over and over again. The elaborated examples in the case study, the model but also some corresponding methods and variables could be used in all other case studies as well by only slightly adding some changes, usually some software specific property like the URL address for exploitation sites. Also, it should be independent from the used implementation technology.

In Figure 5.3 a global overview is given about the main components of the approach of the framework. The greater part of the depiction, apart from ACTS and the grammar, encompasses the part of the framework that is built for test execution, that is, the part that is found upon attack patterns. The execution begins with the attack model, which calls different methods that comprehend all other elements in the selected part of the depiction. Eventually, the tester may be asked for input as well, for example the URL address. The execution and the functionality of every of the components will be discussed in detail in the case studies and evaluation sections. It is important to mention that these elements present the backbone of all approaches throughout the work. All additional technology is built upon or incorporated into this main system. The main and only precondition is a given UML address of the SUT, the HttpClient being responsible for the HTTP traffic while jsoup parses the response and helps with generating a final verdict for every submitted attack vector.

The two remaining elements in the picture represent the combinatorial testing test generation principle. As it will be discussed through Section 5.4 and Section 5.6, it encompasses a grammar, which was created manually by the authors. It is implemented as an input model in ACTS by eventually applying constraints as well. According to these specifications, the test set is generated and saved as a TXT file that is then attached to the execution framework. Before applying ACTS, attack vectors have been specified manually.

The entire system including the statechart model and all methods, variables and external libraries and files are integrated into the Eclipse development environment.

5.3 Attack Pattern-Based Testing

The approach, called Attack Pattern-based Testing, will be discussed on an example and evaluated in the aftermath.

5.3.1 Case Study

Regarding the test cases, two input sets are used for testing, one for SQLI and the other one for both types of XSS. In both cases separate text files are used, each of them containing input strings for the attack type. The program reads the file and saves all strings in a ArrayList and later, these strings are fetched by the model by using a counter variable in order to return a specific entry from the data structure.

The SQLI file consists of injections that comprise strings like *x' OR 'x' = 'x* in several variations, for example with another keywords or symbols but also by using URL encodings etc.

However, for XSS some standard injection scripts have been added but a specific input generation strategy was applied as well. Namely, ACTS was chosen in order to generate input strings. It should be noted that the XSS inputs themselves do not contain any harmful code but are rather kept simple because the goal of the author's method is to test whether a program is vulnerable at all and not to directly cause harm to it. Usually, these strings contain opening and closing tags with the corresponding HTML elements and try to pop-up a warning or some image.

Now the execution of the Mutillidae attack model from Figure 5.1 is discussed and possible differences to other examples are mentioned. The execution of the attack begins in the initial state and resumes to *Started*. The next transition asks the tester to enter the URL address of the SUT and if submitted, the method *connect* is called where the program sends a HTTP request and returns the status code from the response.

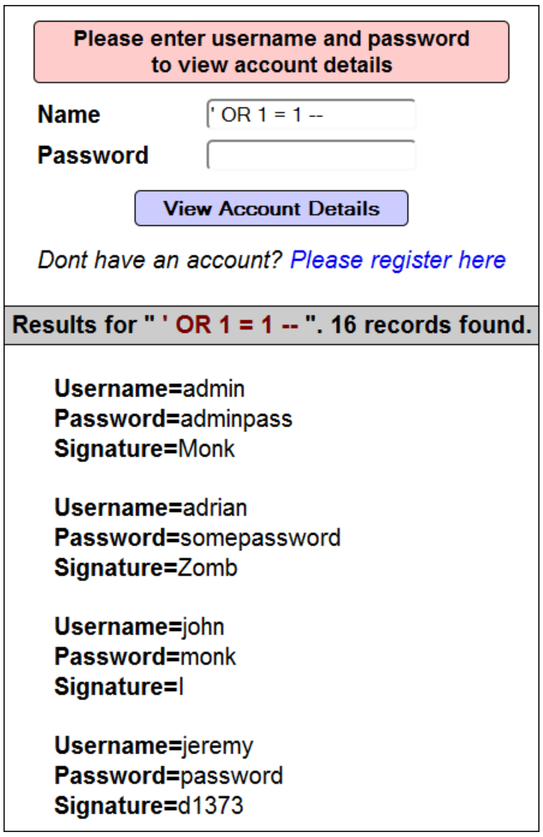
It should also be mentioned again that the background logic of transitions is realized on level of methods, which are already implemented and are part of the testing workspace. If the submitted address is valid, the model accepts a positive status code but otherwise the state *Started* will become active again, hence asking for another URL. In case of a 200, the state *Connected* is activated and because there is no precondition in the next transition, the method *parseInitial* can be called. This method plays an important role because it parses the website for user input fields, request methods etc. in order to find out how a request is to be submitted. The values are stored in global variables in the background. Next the attack type has to be chosen. Here the tester decides what type of attack will be submitted against the SUT, the first choice being SQL injection, the second one XSS. Then, accordingly to the chosen attack, the method *getNumInp* will read the corresponding input file and return the number of entries.

First it is assumed that the SUT is about to be tested against SQLI. In order to test for SQLI, the tester has to specify an expected value which will be compared to the result of the attack function. This could be a string value which would normally never occur by using the application in a predefined manner. The exploitation attempt is realized in a manner that during *attackSQLI* the first attack vector is submitted as a part of the input fields found by *parseInitial* to the SUT by using the URL and *expected* value. Also inside the method, the returned feedback is parsed and searched for this value. For example, the expected value is a stored username, so if the program is vulnerable for an input, the response may look like depicted in Figure 5.4.

If the expected value is found, a Boolean value is returned to the model. In case the result is true, the corresponding transition is taken and now the variable *success*, which indicates a positive vulnerability detection, is increased by one and the next input is read by *generateSQLI* by picking the content from the next line of the ArrayList. However, if the current SQL input returns a negative result, the procedure is the same only that there is no increasing of *success* and the other transition is taken.

In both cases, *SQLI* is entered again but this time the new string is submitted against the SUT. Now, the checking and attacking process is repeated over and over again all until no further inputs can be read. It is important to mention that only a successful attack accordingly to the user specification can be automatically detected. However, it might be the case that the attack invokes another unexpected behavior on side of the SUT, which might not be treated as failure.

The second type attack is meant to trigger XSS. In this approach, XSS exploitation is detected by parsing the response in search for unexpected HTML elements, which should normally not occur. As mentioned before, XSS is realized because of inefficient filtering of user inputs so an injected



Please enter username and password
to view account details

Name ' OR 1 = 1 --

Password

Dont have an account? Please register here

Results for " ' OR 1 = 1 -- ". 16 records found.

Username=admin
Password=adminpass
Signature=Monk

Username=adrian
Password=somepassword
Signature=Zomb

Username=john
Password=monk
Signature=I

Username=jeremy
Password=password
Signature=d1373

Figure 5.4: Result of successful SQLI

input is sent to the user inside the response. So if the attacker submits a payload inside a script, the same script is sent back to a victim.

In fact, both types of XSS can be detected in the same way. First *attackXSS* parses the website for HTML elements like scripts or images and saves their number in the according variables. Then, the first attack vector is taken from the list and is sent as part of the input fields (again from *parseInitial*) against the SUT. Now the program accepts the corresponding response and parses it again but also does a recounting of HTML elements and if an additional element is found, then there is an indicator for a security leak.

If no such suspicious data has been found, the execution proceeds further by fetching the next input from the list. For visualization purposes, Figure 5.5

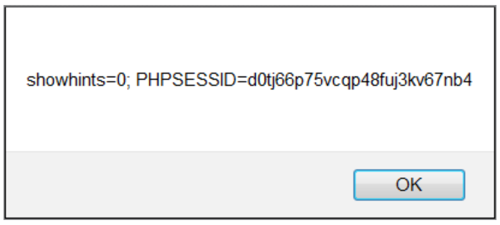

showhints=0; PHPSESSID=d0tj66p75vcqp48fuj3kv67nb4

Figure 5.5: Successful triggering of XSS in Mutillidae

depicts the case when the input *<script>alert(document.cookie)</script>* triggers an action in a browser, in this case being the display of cookie data.

However, if the input is filtered, the program does not parse any new elements but *&lt;script&gt;alert&#x28;document.cookie&#x29;&lt;& #x2f;script&gt;* inside the response body. As can be seen, special HTML characters were used in order to mask certain symbols, thus evading the security breach. The data was returned encrypted in the response so the number of parsed HTML elements after submission remains the same as before and no vulnerability is reported; hence the execution takes another transition.

The principle here is the same as with SQLI although other injection and detection types are used of course. The execution terminates when all inputs are exhausted. The models were tested against five SUT applications and the execution of the test model reveals the faulty behavior. Thus it became possible to show that these applications were vulnerable against SQLI or XSS attacks.

It is also worth mentioning that the model might comprise several different test cases, which are carried out during traversing the states. Testing can be automated using the model, thus making it applicable for a test-driven development process. By modeling attacks instead of the behavior of the model it is also ensured that a SUT has at least limited capabilities of preventing successful attacks that are well known.

5.3.2 Evaluation

As mentioned before, some manual work is still required before automating the testing process. The process of modeling attack patterns may not be very time consuming if the tester has knowledge about the internal structure of the SUT but if the application is unknown, help can be provided by tools like WebScarab, which can give information about the usual communication process between a client and the program. After that, this data can be implemented in the model and especially into necessary Java methods.

In order to evaluate the method, a collection of 33 custom SQLI and 107 XSS attack vectors was used, which syntax differs slightly with respect to the sophistication level. Some SQL inputs are the most common initial injections whereas others contain more advanced examples that use special symbols, comments, escape special symbols, key words, UNION statements and the combination of these methods. For XSS parts strings were used for both types but also symbol escapes, URL encodings, keyword escapes etc. and also combined all of the input types. For that case, manually crafted XSS inputs were used as well as the ones generated by ACTS.

In fact, only those test cases were taken into account, which lead to a positive result, that is, a breach of the security. Table 5.1 depicts all results according to the type of application and attack. Here, the SUT, type of vulnerability (ToV), security level, average execution time in seconds (AET(s)), # of successful injections (#SI) and the coverage percentage categorize the overall results.

However, both attack patterns for both blog applications have been slightly adapted. Firstly, Wordpress was tested while the testing application was authenticated so all inputs were submitted after that step. Then, a blog is picked and all malicious scripts were inserted without checking immediately for vulnerability and after the end of the list of attacks has been reached, the blog address is called again. During this step, all firing scripts are parsed by the program and counted, so the final number of successful inputs is calculated.

Anchor CMS is similar to Wordpress with the difference that all posts have to be approved by the administrator. So in this case, the execution of the

Table 5.1: Initial evaluation results

SUT	ToV	Security Level	AET(s)	#SI	% coverage
DVWA	SQLI	low	8.47	8	24.24
		medium	10.55	2	6.06
		high	-	-	-
	RXSS	low	23.00	15	14.02
		medium	-	-	-
	SXSS	low	26.60	15	14.02
		medium	-	-	-
Mutillidae	SQLI	low	15.69	5	15.15
		medium	17.94	5	15.15
		high	-	-	-
	RXSS	low	42.20	40	37.38
		medium	52.60	40	37.38
		high	-	-	-
	SXSS	low	53.30	17	15.89
		medium	78.10	17	15.89
		high	-	-	-
BodgeIt	SQLI	-	8.50	3	9.09
	RXSS	-	27.20	13	12.15
	SXSS	-	26.30	26	24.30
Wordpress	SXSS	-	33.5	7	6.54
Anchor	SXSS	-	30	8	7.48

model stops after the inputs were submitted and after the tester approves the posts, the execution may continue by parsing all potentially incoming scripts by requesting the blog page.

Concerning DVWA and Mutillidae, the measured difference in the results lies in the fact that every tested program contains different security levels and thus, filtering mechanisms. Also, it was impossible to detect vulnerability on the hardest security level, which means that a more sophisticated test case generation strategy has to be adapted for this purpose, which should implement evasion techniques for more advanced filtering mechanisms that use special HTML characters etc.

However, it should be mentioned that the execution time of the model is slower than the usual execution of Java code. On the other side, when adding choices and more method calls, the execution time remains relatively unaffected so it can be concluded that the number of states has the biggest influence on the overall execution time. In order to reduce this, the number of states should be kept smaller if possible. Because of this and to keep an overview of the model, formerly active states are allowed to become activated again so the number of states remains independent from the number of generated test cases.

It is worth mentioning that when testing stored XSS in DVWA on medium level a vulnerability may be detected only if a successfully string was already injected in the database at the lowest security level. Surprisingly, there seems not to be any security measurements against already stored data so that a former successful input was also successful at this level. If inputs are submitted on the same level by having a clear database, the process was unable to breach the security at all with the new string. The highest difficulty level also remains unbroken.

In Mutillidae if working on medium or high level in a browser, input data won't even be submitted if it contains prohibited input but it is possible to evade this mechanism with HTTPClient, so this part of software could be exploited despite this fact but only on medium level.

Another interesting fact is that it was possible to enter data with a greater length than the actual allowed one by the application without having any error feedback.

5.4 Combinatorial Testing

Testing a SUT requires the existence of test cases and in particular a method capable of generating such test cases. For developing the testing framework methods can be used that arise from the field of combinatorial testing, which is an effective testing technique to reveal errors in a given SUT, based on input combinations coverage. Combinatorial testing of strength t (where $t \geq 2$) requires that each t-wise tuple of values of the different system parameters is covered by at least one test case.

To design a test case, the tester identifies possible output values from each of the actions of the SUT. It is important to find a test case that is not too large, but yet tests for most of the interactions among the possible outputs in the action of the SUT. Recently, some researchers [46, 106, 107] suggested that some faults or errors in SUTs are a combination of a few actions when compared to the total number of parameters of the system under investigation.

Combinatorial testing is motivated by the selection of a few test cases in such a manner that good coverage is still achievable. The combinatorial test design process can be briefly described as follows:

1. Model the input space. The model is expressed in terms of parameter and parameter values.
2. The model is input into a combinatorial design procedure to generate a combinatorial object that is simply an array of symbols.
3. Every row of the generated array is used to output a test case for a SUT.

One of the benefits of this approach is that steps 2. and 3. can be completely automated. In particular, ACTS was used and subsequently the attack pattern-based methodology given in Section 5.3 for these steps.

5.5 Combining the Approaches

A framework is provided for testing and detection of both reflected and stored XSS in web applications. It two parts: a model-based testing method that is based on attack patterns, and combinatorial testing for generating the input data to be submitted to the SUT. Here the idea is to use model-based testing for generating abstract test cases, for example sequences of necessary interactions, and combinatorial testing for concretizing the input.

In the implementation ACTS was used for generation of input strings by specifying parameters and constraints in order to structure the inputs for the particular application domain. Once specified, these inputs are used by the attack pattern model in order to submit them to a web application.

The goal of the proposed approach is to cover standard XSS exploitation attempts by checking whether certain parts of the SUT are vulnerable to potentially malicious scripts. Also, the tester can detect what parts of the SUT are vulnerable and gets the impression how an injection is structured so further measures can be taken. The main differences to other techniques and tools lie in the generation, structure and execution of test cases.

5.6 Combinatorial Grammar for XSS Attack Vectors

In this section, a general structure for XSS attack vectors is considered where each one of them is comprised of 12 discrete parameters where 6 of them are single-valued or have to satisfy certain constraints. The used combinatorial grammar is presented below in BNF form so as to be able to generate possible parameter values through ACTS. It should be noted that although attack grammars for XSS attack vectors have been given in [53], [102], this is the first time that such a grammar is used in terms of combinatorial modeling and the authors' approach for combinatorial security testing is novel in that sense.

```
FOBRACKET ::= <
TAG ::= img | frame | src | script | body | HEAD |
BODY | iframe | IFRAME | SCRIPT
FCBRACKET ::= >
QUOTE1 ::= ‘‘ | ‘ | null
SPACE ::= \n | \t | \r | \r\n | \a | \b | \c | _ | null
EVENT ::= onclick | onmouseover | onerror | onfire |
 onbeforeload | onafterload | onafterlunch |
 onload | onchange | null
SPACE2 ::= \n | \t | \r | \r\n | \a | \b | \c | _ |
 null
QUOTE2 ::= ‘‘ | ‘ | null
PAYLOAD ::= alert(1) | alert(0) |
 alert(document.cookie) | alert("hacked") |
 alert('hacked') |
 src="http://www.cnn.com"&gt;
LOBRACKET ::= </
CLOSINGTAG ::= img | frame | src | script | body |
HEAD | BODY | iframe | IFRAME | SCRIPT
LCBRACKET ::= >
```

BNF Grammar for XSS Attack Vectors

Based on the previously presented attack grammar a XSS attack vector is considered to be of the following form:

$$\text{AV} :=< \text{FOBRACKET}, \text{TAG}, \text{FCBRACKET}, \text{QUOTE1}, \text{SPACE}, \text{EVENT}, \text{SPACE2},$$
$$\text{QUOTE2}, \text{PAYLOAD}, \text{LOBRACKET}, \text{CLOSINGTAG}, \text{LCBRACKET} >$$

The given parameter values are just a fragment of possible options. If the designer would like to generate a vast number of inputs, it is sufficient to just add parameter values in the given BNF for XSS attack vectors. Moreover, the grammar also satisfies some constraints using the constraint tool from

ACTS. For example, the TAG parameter is matched with the CLOSINGTAG parameter to always be able to produce valid inputs.

For the second step of the combinatorial test design process the notion of mixed-level covering arrays (a specific class of combinatorial designs) is used. For the sake of complicity the definition of mixed-level covering arrays is provided in Section 3.1 as Definition 1, which is taken from [47] since this is the underlying generated structure in the ACTS tool.

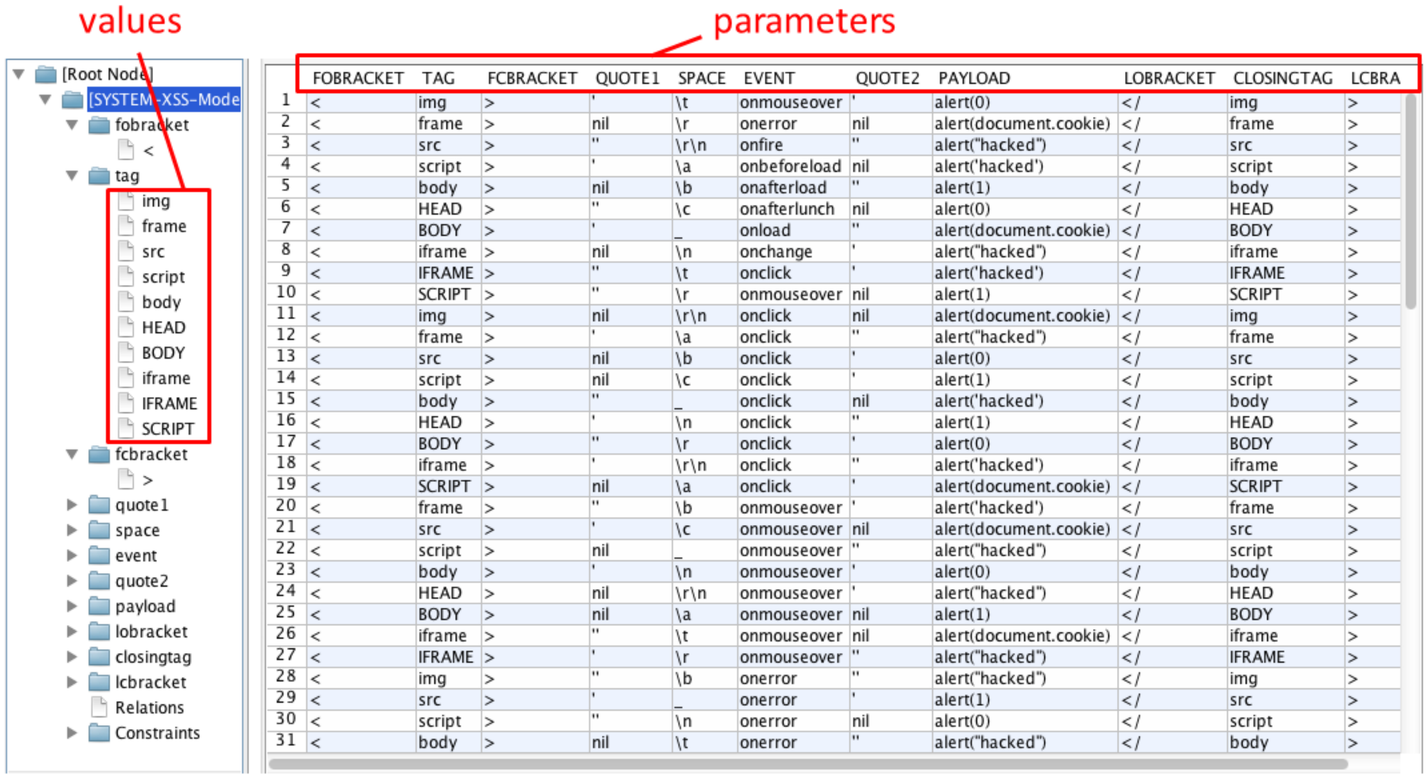
It should be remarked that this technique of discretizing the parameter values is referred to as input parameter modeling for combinatorial testing [73] and essentially enables the designer to choose the possible parameter values for the SUT. Thus, it is natural to define the attack grammar as a combinatorial one when the first one is used for input parameter modeling. Figure 5.6 depicts the System View of ACTS with already specified grammar, its parameters, values as well as the generated output.

Essentially, this means that given the t-wise interaction of the covering array ACTS generates all possible t-tuples of parameter values for a number of t total parameters in the SUT. This is another explanation for the strength t of the covering array where for any selection of t-rows each t-tuple appears at least once.

For all cases, the parameters of the MCA are derived from the BNF combinatorial grammar according to the following formulation. The number of rows of the MCA equals to the number of types in the presented combinatorial grammar while the size of alphabets g_i of the MCA equals to the number of derivation rules per type. For example, all XSS attack vectors from $MCA(2, 12, (1, 1, 1, 1, 3, 3, 6, 9, 9, 10, 10))$ for the *src* tag when there is need to test for pairwise interactions ($t = 2$) can be found in Table 5.2.

5.7 Attack Pattern-Based Combinatorial Testing: Initial Evaluation

For combinatorial interaction strength 2, ACTS threw out 114 inputs and respectively 1031 and 8332 strings for strength 3 and strength 4. The ob-



- [Root Node]
  - [SYSTEM-XSS-Mode
    - fobracket
      - <
    - tag
      - img
      - frame
      - src
      - script
      - body
      - HEAD
      - BODY
      - iframe
      - IFRAME
      - SCRIPT
    - fcbracket
      - >
    - quote1
    - space
    - event
    - quote2
    - payload
    - lobracket
    - closingtag
    - lcbracket
    - Relations
    - Constraints

	FOBRACKET	TAG	FCBRACKET	QUOTE1	SPACE	EVENT	QUOTE2	PAYLOAD	LOBRACKET	CLOSINGTAG	LCBRA
1	<	img	>	'	\t	onmouseover	'	alert(0)	</	img	>
2	<	frame	>	nil	\r	onerror	nil	alert(document.cookie)	</	frame	>
3	<	src	>	"	\r\n	onfire	"	alert("hacked")	</	src	>
4	<	script	>	'	\a	onbeforeload	nil	alert('hacked')	</	script	>
5	<	body	>	nil	\b	onafterload	"	alert(1)	</	body	>
6	<	HEAD	>	"	\c	onafterlunch	nil	alert(0)	</	HEAD	>
7	<	BODY	>	'	_	onload	"	alert(document.cookie)	</	BODY	>
8	<	iframe	>	nil	\n	onchange	'	alert("hacked")	</	iframe	>
9	<	IFRAME	>	"	\t	onclick	'	alert('hacked')	</	IFRAME	>
10	<	SCRIPT	>	"	\r	onmouseover	nil	alert(1)	</	SCRIPT	>
11	<	img	>	nil	\r\n	onclick	nil	alert(document.cookie)	</	img	>
12	<	frame	>	'	\a	onclick	"	alert("hacked")	</	frame	>
13	<	src	>	nil	\b	onclick	'	alert(0)	</	src	>
14	<	script	>	nil	\c	onclick	'	alert(1)	</	script	>
15	<	body	>	"	_	onclick	nil	alert('hacked')	</	body	>
16	<	HEAD	>	'	\n	onclick	"	alert(1)	</	HEAD	>
17	<	BODY	>	"	\r	onclick	'	alert(0)	</	BODY	>
18	<	iframe	>	'	\r\n	onclick	"	alert('hacked')	</	iframe	>
19	<	SCRIPT	>	nil	\a	onclick	'	alert(document.cookie)	</	SCRIPT	>
20	<	frame	>	"	\b	onmouseover	'	alert('hacked')	</	frame	>
21	<	src	>	'	\c	onmouseover	nil	alert(document.cookie)	</	src	>
22	<	script	>	nil	_	onmouseover	"	alert("hacked")	</	script	>
23	<	body	>	'	\n	onmouseover	'	alert(0)	</	body	>
24	<	HEAD	>	nil	\r\n	onmouseover	'	alert("hacked")	</	HEAD	>
25	<	BODY	>	nil	\a	onmouseover	nil	alert(1)	</	BODY	>
26	<	iframe	>	"	\t	onmouseover	nil	alert(document.cookie)	</	iframe	>
27	<	IFRAME	>	'	\r	onmouseover	"	alert("hacked")	</	IFRAME	>
28	<	img	>	"	\b	onerror	"	alert("hacked")	</	img	>
29	<	src	>	'	_	onerror	'	alert(1)	</	src	>
30	<	script	>	"	\n	onerror	nil	alert(0)	</	script	>
31	<	body	>	nil	\t	onerror	"	alert("hacked")	</	body	>

Figure 5.6: System View in ACTS

Table 5.2: A sample of XSS attack vectors

FOBRACKET	TAG	FCBRACKET	QUOTE1	SPACE	EVENT	SPACE2	QUOTE2	PAYLOAD	LOBRACKET	CLOSINGTAG	LCBRACKET
<	src	>	’	\r\n	onclick	\b	’	alert(document.cookie)	</	src	>
<	src	>	null	\a	onmouseover	\c	’’	alert(\hacked\")"	</	src	>
<	src	>	’’	\b	onerror	_	null	alert(’hacked’)	</	src	>
<	src	>	null	\c	onfire	null	’’	src=\http://www.cnn.com\">"	</	src	>
<	src	>	’	_	onbeforeload	\n	’’	alert(1)	</	src	>
<	src	>	’	nil	onafterload	\t	’’	alert(0)	</	src	>
<	src	>	’’	\n	onafterlunch	\r\n	’’	alert(document.cookie)	</	src	>
<	src	>	’	\t	onload	\a	’	alert(\hacked\")"	</	src	>
<	src	>	’’	\r	onchange	\b	’’	alert(’hacked’)	</	src	>
<	src	>	’	\r\n	nil	\r	’	alert(1)	</	src	>

tained results of the evaluation are depicted in Table 5.3. It should be also mentioned that input fields were tested with textual or password values as well as textarea tags.

In the table, information is given about the strength (S), SUT, type of vulnerability (ToV), difficulty level (DL), execution time in seconds, # of successful injections (#SI) as well as the percentage of coverage.

In contrast to manual testing, several issues were encountered during the testing process about the structure and meaning of successful vulnerability detection. Before doing the automated tests, many of the generated inputs from ACTS were used by the testers in order to see how the SUT reacts by executing them in the browser. During that process the conclusion was drawn that Google Chrome is more resistant to malicious inputs than Mozilla Firefox regarding the filtering of inputs on browser side so many scripts which were executed in Firefox, could not be triggered in Chrome. For this sake, another interesting fact is that the program reported more encounters of critical HTML elements than when executed by a browser.

In the table it can be observed that in DVWA not a single breakthrough was reported on the medium and hard levels. On the lowest security level the got the same results for both persistent and stored XSS, so it is assumed that the software uses identical defense mechanisms for these cases although this seems not to be the case with the higher level, where type-2 XSS could not be triggered at all.

On the hand, Mutillidae and BodgeIt have been breached on both levels for both types of XSS.

The authors' observations led to the conclusion that only certain tags could go unfiltered, especially $script$, src and $iframe$ while others like $body$, $head$ and $frame$ were not able to cause a security break. This leads to the obvious conclusion that such applications need more protection against these critical tags and keywords.

Also, the specified attack grammar in ACTS is so far successful for script tags in case no filtering is applied, so it actually may be possible to put a more malicious code inside a script by using the same structure but by expanding the grammatical content between the tags. On the other hand, a redefinition of unsuccessful elements is eventually needed. All this does

Table 5.3: Initial evaluation results - Combinatorial Testing

S	SUT	ToV	DL	Execution time (s)	#SI	% coverage
2	BodgeIt	RXSS	-	30.00	69	60.53
		SXSS	-	31.80	56	49.12
3		RXSS	-	244.60	619	60.04
		SXSS	-	242.80	512	49.66
4		RXSS	-	1788.70	4991	59.90
		SXSS	-	1950.80	4157	49.89
2	DVWA	RXSS	1	27.40	69	60.53
		SXSS	1	28.50	69	60.53
		RXSS	2	30.70	56	49.12
		SXSS	2	30.60	0	0.00
3		RXSS	1	281.80	619	60.04
		SXSS	1	284.90	619	60.04
		RXSS	2	269.30	512	49.66
		SXSS	2	289.20	0	0.00
4		RXSS	1	2284.30	4991	59.90
		SXSS	1	3200.60	4991	59.90
		RXSS	2	2684.19	4157.00	49.89
		SXSS	2	3010.10	0	0.00
2	Mutillidae	RXSS	1	56.60	69	60.53
		SXSS	1	61.10	41	35.96
		RXSS	2	74.20	69	60.53
		SXSS	2	80.80	41	35.96
3		RXSS	1	514.00	619	60.04
		SXSS	1	536.30	302	29.29
		RXSS	2	625.90	619	60.04
		SXSS	2	700.30	302	29.29
4		RXSS	1	3990.90	4991	59.90
		SXSS	1	4451.70	2398	28.78
		RXSS	2	5246.00	4991	59.90
		SXSS	2	5586.50	2405	28.86

not downgrade the using of combinatorial testing but eventually reveals something about the mechanism of designing grammars and payloads for XSS, which might be expanded further.

The reason for the overall results may be the fact that the structure of the generated inputs by ACTS is relatively manageable according to the specified parameter segments for the generation process. For example, it was specified that all of the inputs begin with the opening tag and conclude with its closing counterpart and also use the usual HTML element names. The detection mechanisms in the proposed program have been implemented as well in order to be effective against the same symbols and HTML elements. In all the cases where the SUT didn't have any intern filtering mechanisms specified against these symbols, the submitted element is returned and parsed successfully without being filtered by the application. However, if the system already has some prevention mechanisms like the evasion of specific tags or keywords from the input, then all incoming injections with these symbols were rejected. For that reason, no testing quality could be achieved just by increasing the number of inputs as long as some different combinatorial syntax isn't defined. In fact, high coverage rates are obtained in all cases where this filtering hasn't been applied.

Another very interesting fact is that for every combinatorial interaction strength, the same coverage of positive results was achieved, that is, vulnerability detections, no matter how many inputs were submitted. Also, BodgeIt and Mutillidae were tested with inputs from strength 5 with 48755 inputs but then only to reaffirm the same conclusion.

From a purely combinatorial point of view some additional conclusions are drawn for the same coverage of positive results. Firstly, it is deduced that the interaction of the involved parameters is independent of the SUTs that were evaluated. In other words, if a close look is taken on the evaluation results it can be seen that an interaction of two modeled parameters ($t = 2$) is sufficient for a tester to penetrate the specific web applications (at least on the first security levels where applicable). As already discussed, increasing the strength of the covering arrays in ACTS for $t = 3$ and $t = 4$ implies an increase on the number of interactions of the parameters of the generated inputs. Having this in mind, obtaining the same coverage percentage leads to the conclusion that every possible positive input has been generated already for strength $t = 2$ (and reproduced for higher strengths).

As usually is the case when considering the mathematical modeling of complex systems (and such are the SUTs that were evaluated) there are some certain advantages and disadvantages of Attack Pattern-based Combinatorial Testing. One positive aspect of the presented methodology is that the (general) combinatorial testing oracle from [74] can also be used for security testing of web applications. Recall that this oracle is based on first testing all two pairwise interactions ($t = 2$). Then the tester continues by increasing the interaction strength t until no further errors (injections in this case) are detected by the t-way tests (inputs), and finally optionally tries $t + 1$ and ensures that no additional errors are detected. The main motivation for combinatorial testing is the reduction of the search space while still being able to test for t-way interactions. The latest is also a sufficient condition for the successful application of combinatorial testing. This milestone is considered to be achieved as for all tested SUTs; a significant reduction on the number of possible inputs was accomplished. For example, with the grammar all 2-way interactions were tested with 114 inputs out of all 158799 possible inputs when considering exhaustive search of its parameters. This implies a reduction of 99.93% of the total search space produced by the combinatorial grammar while still being able to penetrate the aforementioned SUT.

This efficiency on test execution however does not come with some drawbacks. It was mentioned that a more complex combinatorial grammar is needed to break through the higher security levels of the SUTs. Unfortunately, adding more values in the parameters of the grammar would contribute to an exponential growth of the computational time required to generate the required covering arrays by ACTS. One possible way to overcome this shortcoming would be to consider adding more parameters in the grammar instead of parameter values as this would increase the complexity only by a logarithmic scale. Here the evaluation on the combinatorial part of the authors' methodology is concluded and the last remark is considered as a starting point for further research on the cross-fertilization of the fields of combinatorial testing and web application security.

5.8 Evaluation of the IPO-Family Algorithms for Test Case Generation

An examination is undertaken in order to show how two of the most popular algorithms for combinatorial test case generation, namely the IPOG and IPOG-F algorithms, perform in web security testing. For generating comprehensive and sophisticated testing inputs input parameter modeling has been used by adding constraints between the different parameter values. The evaluation indicates that both algorithms generate test inputs that succeed in revealing security leaks in web applications with IPOG-F giving overall slightly better results w.r.t. the test quality of the generated inputs. In addition, using constraints during the modeling of the attack grammars results in an increase on the number of test inputs that cause security breaches.

Last but not least, a detailed analysis of the evaluation results confirms that combinatorial testing is an efficient test case generation method for web security testing as the security leaks are mainly due to the interaction of a few parameters. This statement is further supported by some combinatorial coverage measurement experiments on the successful test inputs. It should be noted that, in all of the previous works the focus was to compare manual to automated penetration testing tools and the underlying combinatorial test generation algorithm was IPOG. The main difference to this work is the inclusion of the IPOG-F algorithm for generating the test suites and a comparison between these two algorithms of the IPO-family applicable in web security testing, for the first time.

In this work the IPOG-based test generation algorithm [75] was used and its refinement IPOG-F [58] to generate the test suites, as these are implemented in ACTS. Both algorithms are a generalization of the in-parameter-order (IPO) strategy first introduced by Lei and Tai [76]. A detailed description of the IPO-family can be further found in [73] and [108].

5.8.1 Combinatorial Grammar for XSS Attack Vectors and Constraints

Here the general structure for XSS attack vectors is reviewed where each one of them is comprised of 11 discrete parameters (types) and discussed in detail below. This new structure builds upon a combinatorial grammar given in Section 5.6 by modeling white spaces and executable JavaScript that can appear in an XSS attack vector but also extends the one given in [61, 32] by adding constraints between the different parameters values.

For the sake of completeness, a fragment of the combinatorial grammar is presented below, denoted by `G`, in BNF form so as to be able to generate possible parameter values through ACTS, where inside the parentheses in the parameters the full range of values is listed, which has been taken into account in the implementation.

```
JSO(15)::=␣<script>␣|␣<img␣|␣’’><script>␣|␣...
WS1(3)::=␣tab␣|␣space␣|␣empty
INT(14)::=␣\";␣|␣’>␣|␣">>␣|␣...
WS2(3)::=␣tab␣|␣space␣|␣empty
EVH(3)::=␣onLoad(␣|␣onMouseOver(␣|␣onError(
WS3(3)::=␣tab␣|␣space␣|␣empty
PAY(23)::=␣alert(’XSS’)␣|␣SRC="javascript:alert(’1’);">␣|␣
   HREF="http://ha.ckers.org/xss.js">␣|␣...
WS4(3)::=␣tab␣|␣space␣|␣empty
PAS(11)::=␣’)␣|␣;//␣|␣’>␣|␣...
WS5(3)::=␣tab␣|␣space␣|␣empty
JSE(9)::=␣</script>␣|␣\>␣|␣’\>␣|␣...
```

Reviewed BNF Grammar for XSS Attack Vectors

The expert knowledge was essential in the design of the AV, where the goal is to produce valid JavaScript code when this is injected into SUT parameters. The description of the types in the previous AV has briefly mentioned in [61], however here it is also included, in more detail, for the sake of completeness:

- The **JSO** (JavaScript Opening Tags) type represent tags that open a JavaScript code block. They also contain values that use common techniques to bypass certain XSS filters, like `<script>` or `<img`.
- The **WS** (white space) type family represents the white space but also variations of it like the tab character in order to circumvent certain filters.
- The **INT** (input termination) type represents values that terminate the original valid tags (HTML or others) in order to be able to insert and execute the payload, like `"'>` or `">`.
- The **EVH** (event handler) type contains values for JavaScript event handlers. The usage of JavaScript event handlers, like `onLoad(` or `onError(`, is a common approach to bypass XSS filters that filter out the typical JavaScript opening tag like `<script>` or filters that remove brackets (especially < and >).
- The **PAY** (payload) type contains executable JavaScript like `alert("XSS")` or `ONLOAD=alert('XSS')>`. This type contains different types of executable JavaScript in order to bypass certain XSS filters.
- The **PAS** (payload suffix) type contains different values that should terminate the executable JavaScript payload (PAY state value). The PAS is necessary to produce valid JavaScript code that is interpreted by a browser like `')` or `'>`.
- The **JSE** (JavaScript end tag) type contains different forms of JavaScript end tags in order to produce valid JavaScript code like `</script>` or `>`.

In addition, constraints were added to the XSS attack grammar. The motivation for this reason rises from the fact that in real-world systems adding constraints may produce test suites with better quality and also considerably reduce the search space. This approach for combinatorial testing has been followed for example in [85, 30, 95].

It should be noted that although attack grammars for XSS attack vectors have been given in [34, 61, 32, 102, 53, 54], this is the first time that constraints are imposed on such attack grammars in terms of combinatorial modeling when the model is input to the IPOG-F algorithm and the approach for revisiting the notion of *combinatorial security testing* is novel in that sense.

Below the full set of constraints for the grammar is presented by using the constraint tool from ACTS where the symbol => means an implication and the symbol `||` an OR statement. This grammar will be denoted with `G_C` to distinguish it from `G` when constraints are enforced.

```
(JSO=1)  =>  (JSE=1)
(JSO=2)  =>  (JSE=2)
(JSO=3)  =>  (JSE=3)
(JSO=4)  =>  (JSE=2  ||  JSE=4)
(JSO=5)  =>  (JSE=5  ||  JSE=6  ||  JSE=7  ||  JSE=8  ||  JSE=9)
(JSO=6)  =>  (JSE=5  ||  JSE=6  ||  JSE=7  ||  JSE=8  ||  JSE=9)
(JSO=7)  =>  (JSE=5  ||  JSE=6  ||  JSE=7  ||  JSE=8  ||  JSE=9)
(JSO=9)  =>  (JSE=5  ||  JSE=6  ||  JSE=7  ||  JSE=8  ||  JSE=9)
(JSO=10)  =>  (JSE=9)
(JSO=11)  =>  (JSE=2  ||  JSE=3  ||  JSE=4)
(JSO=12)  =>  (JSE=5  ||  JSE=6  ||  JSE=7  ||  JSE=8  ||  JSE=9)
(JSO=13)  =>  (JSE=5  ||  JSE=6  ||  JSE=7  ||  JSE=8  ||  JSE=9)
(JSO=14)  =>  (JSE=2  ||  JSE=3  ||  JSE=4)
(JSO=15)  =>  (JSE=2  ||  JSE=3  ||  JSE=4)
(EVH=1)  =>  (PAY=12  ||  PAY=14  ||  PAY=17  ||  PAY=18  ||  PAY=19)
(EVH=2)  =>  (PAY=13  ||  PAY=14  ||  PAY=17  ||  PAY=18  ||  PAY=19)
(EVH=3)  =>  (PAY=12  ||  PAY=13  ||  PAY=17  ||  PAY=18  ||  PAY=19)
(INT=2)  =>  (PAS=10  ||  PAS=11)
(INT=3)  =>  (PAS=10  ||  PAS=11)
(INT=4)  =>  (PAS=10  ||  PAS=11)
(INT=5)  =>  (PAS=6  ||  PAS=7  ||  PAS=8  ||  PAS=9)
(INT=6)  =>  (PAS=6  ||  PAS=7  ||  PAS=8  ||  PAS=9)
(INT=7)  =>  (PAS=6  ||  PAS=7  ||  PAS=8  ||  PAS=9)
(INT=8)  =>  (PAS=6  ||  PAS=7  ||  PAS=8  ||  PAS=9)
(INT=9)  =>  (PAS=6  ||  PAS=7  ||  PAS=8  ||  PAS=9)
(INT=10)  =>  (PAS=6  ||  PAS=7  ||  PAS=8  ||  PAS=9)
(INT=11)  =>  (PAS=10  ||  PAS=11)
(WS1=WS2  &&  WS2=WS3  &&  WS3=WS4  &&  WS4=WS5)
```

Constraints for XSS Attack Grammar

In the implementation, a translation layer was used from the integer values presented previously to actual values per derivation type, in a similar manner to the ones presented for `G`. The rationale for the used constraints by incorporating the expert knowledge is as follows:

- Constraints of the form `(JSO=x) => (JSE=y)` for appropriate values x and y ensure coupling of the outermost JavaScript opening and closing tags. An alignment between these two types is necessary to produce valid JavaScript code.
- Constraints of the form `(EVH=x) => (PAY=y)` for appropriate values x and y offer the possibility to use only those payload values, which are, derived from expert knowledge, most likely to succeed when coupled with the respective event handler.
- Constraints of the form `(INT=x) => (PAS=y)` for appropriate values x and y further express necessary correlations/conditions between the two stages to be able to ultimately execute the payload type, which an experienced tester would intuitively use when performing manual penetration testing.
- Constraints between the whitespace types mainly serve to reduce the overall size of the test suites.

This grammar will be denoted with `G_c` to distinguish it from `G` when constraints are enforced.

5.8.2 Combinatorial Metrics for Security Testing

There has been a great need for metrics, for example how to measure the efficiency of testing experiments, in software security the latest years. Many different notions of coverage criteria used in traditional software testing such as branch coverage and statement coverage were also adopted by security researchers [70] and some new have been proposed [49].

In the context of automated security testing for web applications [49], it was defined as the *exploitation rate* of a SUT, denoted by ER, the proportion of XSS attack vectors that were successful, for example the ones that exploit an XSS vulnerability, per given test suite and SUT:

$$\mathrm{ER} = \frac{\text{\# Attack vectors that exploit an XSS vulnerability}}{\text{Total number of attack vectors per test suite and SUT}} \tag{5.1}$$

In this work, combinatorial coverage measurement metrics (which differ to the ER) have been used as well to determine the quality of the successful attack vectors. In particular, the interest is put on the number of t-way combinations covered in passing tests. This notion of combinatorial coverage [73], which can be computed by the CCM tool [51], is defined as the proportion of covered t-way (valid) tuples of given k parameters in a mixed-level covering array. Such tuples in combinatorial testing literature are also met as variable-value configurations.

5.8.3 Evaluation

It should be noted that until now in all of the previous case studies the test suites have been generated with the IPOG algorithm. However, in this case study the test suites for IPOG have been regenerated and also new ones were created with the IPOG-F algorithm.

This was needed as one of the goals of the case study is to investigate which of the two algorithms of the IPO family, namely IPOG and IPOG-F, generates better quality test suites w.r.t. to XSS vulnerability detection. In particular, the intention is to compare the exploitation rate of the test suites focused on triggering XSS exploits. Secondly, on the same level of importance, the intention is to investigate the combinatorial coverage of passing tests to determine whether security leaks are caused by the interaction of a few parameters.

The SUTs were tested with different sets of test suites, produced by the ACTS tool and based on the combinatorial grammar given in Section 5.8.1. The difference between the four sets of test suites rely on imposing various constraints on the different types of the attack grammar and the underlying combinatorial test case generation algorithm used.

As before, an XSS grammar was used for the input model in ACTS and generated inputs for combinatorial interaction strengths $t \in \{2, 3, 4\}$ twice for each algorithm, where the first dataset consists of attack vectors that were created without setting any constraints on the data structure, while the second input file comprises attacks which were generated according to constraints mentioned in Section 5.8.1. All mentioned applications have been tested against all attack vectors and the responsive results are displayed.

The cardinality of the total space is $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 9 \times 11 \times 14 \times 15 \times 23 = 348585930$ tests. When generating the test suites using the authors' model as input to a $MCA(t, 11, (3, 3, 3, 3, 3, 3, 9, 11, 14, 15, 23))$, for $t \in \{2, 3, 4\}$, a reduction of $\approx 99.99\%$ of the total search space can be observed. In addition, from Table 5.4. it is evident that the test suites generated by the IPOG-F are smaller than the ones produced by the IPOG algorithm, for the model without constraints and the situation is reversed when these are enforced. Regarding the usage of constraints, enforcing them in test case generation makes the test suites even smaller in both cases.

Finally, it should be noted that the test generation in ACTS for all test suites was quite fast in a normal workstation, ranging from some seconds to a couple of minutes for increasing strength.

All test suites were executed using the attack pattern-based testing framework. In order to draw a meaningful comparison, the same elements of a SUT were tested with all test suites, which were input fields with textual and password values or textarea tags but attack vectors were as well attached to the URL paths without any variable binding.

Exploitation Rate of SUTs and their Relation to the IPO Family

In this section, it is investigated how the exploitation rate of the different SUTs, which were considered in the case study, scales when the combinatorial interaction strength increases, for given difficulty level and input field in each one of the SUTs for the IPOG and IPOG-F algorithms. The evaluation results are depicted in Table 5.4 for the XSS combinatorial grammar that was used (with and without constraints on the parameter values) when the generated vectors are tested against the SUTs.

In particular, in Table 5.4 information is given about the combinatorial interaction strength (Str.), the SUT, the input field ID (inp_ID), type of vulnerability (VT), eventually the difficulty level (DL), the exploitation rate (the number of positive inputs divided by the total number of tested vectors) and its respective percentage for both IPOG and IPO-F algorithms. The abbreviation for the SUTs are given as well, that is, Mutillidae (M), BodgeIt

Table 5.4: Evaluation results per SUT for given difficulty level and input field with increasing strength

SUT parameters				Str.	G				G_c			
					IPOG		IPOG-F		IPOG		IPOG-F	
App	DL	VT	inp_id		ER	% ER	ER	% ER	ER	% ER	ER	% ER
M	0	RXSS	1	2	111/345	32.17	102/345	29.57	116/250	46.40	121/252	48.02
M	0	RXSS	1	3	1580/4875	32.41	1561/4830	32.32	836/1794	46.59	950/2012	47.22
M	0	RXSS	1	4	17344/53706	32.29	17127/53130	32.24	3974/8761	45.36	4449/9760	45.58
M	0	RXSS	2	2	129/345	37.39	136/345	39.42	116/250	46.40	123/252	48.81
M	0	RXSS	2	3	1849/4875	37.93	1822/4830	37.72	839/1794	46.77	942/2012	46.82
M	0	RXSS	2	4	20135/53706	37.49	19957/53130	37.56	4108/8761	46.89	4667/9760	47.82
M	0	RXSS	3	2	0/345	0.00	0/345	0.00	0/250	0.00	0/252	0.00
M	0	RXSS	3	3	0/4875	0.00	0/4830	0.00	0/1794	0.00	0/2012	0.00
M	0	RXSS	3	4	0/53706	0.00	0/53130	0.00	0/8761	0.00	0/9760	0.00
M	1	RXSS	1	2	111/345	32.17	102/345	29.57	116/250	46.40	121/252	48.02
M	1	RXSS	1	3	1580/4875	32.41	1561/4830	32.32	836/1794	46.59	950/2012	47.22
M	1	RXSS	1	4	17344/53706	32.29	17127/53130	32.24	3974/8761	45.36	4449/9760	45.58
M	1	RXSS	2	2	129/345	37.39	136/345	39.42	116/250	46.40	123/252	48.81
M	1	RXSS	2	3	1849/4875	37.93	1822/4830	37.72	839/1794	46.77	942/2012	46.82
M	1	RXSS	2	4	20135/53706	37.49	19957/53130	37.56	4108/8761	46.89	4667/9760	47.82
M	1	RXSS	3	2	0/345	0.00	0/345	0.00	0/250	0.00	0/252	0.00
M	1	RXSS	3	3	0/4875	0.00	0/4830	0.00	0/1794	0.00	0/2012	0.00
M	1	RXSS	3	4	0/53706	0.00	0/53130	0.00	0/8761	0.00	0/9760	0.00
B	0	RXSS	1	2	198/345	57.39	201/345	58.26	145/250	58.00	153/252	60.71
B	0	RXSS	1	3	2842/4875	58.30	2842/4830	58.84	1073/1794	59.81	1207/2012	59.99
B	0	RXSS	1	4	31441/53706	58.54	31120/53130	58.57	5366/8761	61.25	6084/9760	62.34
B	0	RXSS	2	2	131/345	37.97	135/345	39.13	97/250	38.80	106/252	42.06
B	0	RXSS	2	3	1890/4875	38.77	1888/4830	39.09	737/1794	41.08	823/2012	40.90
B	0	RXSS	2	4	20927/53706	38.97	20648/53130	38.86	3918/8761	44.72	4551/9760	46.63
B	0	SXSS	3	2	32/345	9.28	32/345	9.28	37/250	14.80	38/252	15.08
B	0	SXSS	3	3	561/4875	11.51	551/4830	11.41	257/1794	14.33	273/2012	13.57
B	0	SXSS	3	4	6052/53706	11.27	6024/53130	11.34	1504/8761	17.17	1590/9760	16.29
B	0	SXSS	4	2	308/345	89.28	305/345	88.41	217/250	86.80	215/252	85.32
B	0	SXSS	4	3	4434/4875	90.95	4407/4830	91.24	1558/1794	86.85	1745/2012	86.73
B	0	SXSS	4	4	42898/53706	79.88	42378/53130	79.76	7676/8761	87.62	8618/9760	88.30
G	0	RXSS	1	2	122/345	35.36	122/345	35.36	89/250	35.60	92/252	36.51
G	0	RXSS	1	3	1744/4875	35.77	1755/4830	36.33	671/1794	37.40	758/2012	37.67
G	0	RXSS	1	4	19381/53706	36.09	19223/53130	36.18	3303/8761	37.70	3566/9760	36.54
G	0	SXSS	2	2	23/345	6.67	23/345	6.67	17/250	6.80	18/252	7.14
G	0	SXSS	2	3	327/4875	6.71	322/4830	6.67	118/1794	6.58	136/2012	6.76
G	0	SXSS	2	4	3587/53706	6.68	3542/53130	6.67	610/8761	6.96	749/9760	7.67
W	0	RXSS	2	2	198/345	57.39	201/345	58.26	145/250	58.00	153/252	60.71
W	0	RXSS	2	3	2842/4875	58.30	2842/4830	58.84	1073/1794	59.81	1207/2012	59.99
W	0	RXSS	2	4	31441/53706	58.54	31120/53130	58.57	5366/8761	61.25	6084/9760	62.34
D	0	RXSS	1	2	175/345	50.72	178/345	51.59	128/250	51.2	134/252	53.17
D	0	RXSS	1	3	2517/4875	51.63	2520/4830	52.17	954/1794	53.18	1081/2012	53.73
D	0	RXSS	1	4	27864/53706	51.88	27578/53130	51.91	4755/8761	54.27	5345/9760	54.76
D	0	SXSS	2	2	91/345	26.38	92/345	26.67	88/250	35.20	84/252	33.33
D	0	SXSS	2	3	1303/4875	26.73	1302/4830	26.96	537/1794	29.93	616/2012	30.62
D	0	SXSS	2	4	14068/53706	26.19	13928/53130	26.21	2276/8761	25.98	2825/9760	28.96
D	1	RXSS	1	2	106/345	30.72	109/345	31.59	80/250	32.00	86/252	34.13
D	1	RXSS	1	3	1548/4875	31.75	1554/4830	32.17	613/1794	34.17	692/2012	34.39
D	1	RXSS	1	4	17173/53706	31.98	16952/53130	31.91	3285/8761	37.50	3787/9760	38.80
D	1	SXSS	2	2	0/345	0.00	0/345	0.00	0/250	0.00	0/252	0.00
D	1	SXSS	2	3	0/4875	0.00	0/4830	0.00	0/1794	0.00	0/2012	0.00
D	1	SXSS	2	4	0/53706	0.00	0/53130	0.00	0/8761	0.00	0/9760	0.00

(B), Gruyere (G), Webgoat (W) and DVWA (D). Further, the table gives the corresponding results for constrained values in each case.

In DVWA, higher interaction strengths caused higher exploitation rates when using the constrained test set. This was the case with test sets from both algorithms. When the same element in the SUT was tested against reflected XSS, the program obtained 32% for $t = 2$ and 37.50% for the $t = 4$ for IPOG. The other algorithm gave for the same elements results around 34.13% and 38.80%. A slight increase for constrained vectors was also reported in Webgoat with IPOG, where for $t = 2$ the output has been 58% and 61.25% for the highest strength. A slight increase of 1.63% was also reported when using IPOG-F for the same SUT. When Mutillidae was tested with unconstrained vectors, 29.57% was achieved for $t = 2$ and corresponding 32.24% for $t = 4$.

On the other hand, when comparing the testing outcome from both algorithms regarding the test sets, the most evident difference is obtained in Mutillidae. In this SUT the exploitation rate differs most between constrained values and their counterpart. While the overall results for the first tested element result around 32% for all interaction strengths in IPOG, for the same algorithm we achieved much higher results with constrained vectors. In this case, the program reported respectively 46.60% and 48.02% in favor for IPOG-F. Similar results were observed after testing another element of this SUT. However, the program was not able to trigger any vulnerability for another element. The assumption is that this part uses defense mechanisms that reject parts of the vectors. Also, equal results were obtained when the SUT was tested against the same test sets but on a higher difficulty level. In this case, additional attack prevention mechanisms were activated inside the SUT. However, vectors that were before able to succeed in triggering XSS, also caused the same effect in the upgraded SUT.

In DVWA the program calculated 31.75% for IPOG for $t = 3$, while achieving 34.17% for the same algorithm when using constrained values. When tested with $t = 4$, results varied even more with respectively 31.91% and 38.80% for IPOG-F. In BodgeIt the results were 79.88% for unconstrained vectors in IPOG but 87.62% for the other test set when tested against stored XSS. Similar results were obtained while testing for reflected XSS.

Now the test results for both algorithms for Webgoat will be discussed. This SUT was tested just for reflected XSS against one element. In this case, an exploitation rate of about 57.39% was observed for the lowest interaction strength in IPOG. However, the biggest difference is calculated when using constrained vectors. Here a slight increase was obtained in the exploitation rate with IPOG-F when tested against $t = 2$. Moreover, a result of about 60.71% was observed compared to the same test set from the other algorithm, where only 58% was achieved. A slightly smaller difference was achieved for the unconstrained counterpart. However, here the exploitation result was higher for IPOG-F for every interaction strength. A better result is also confirmed for $t = 4$ where IPOG got 61.25% but IPOG-F succeeded with additional 1.09%.

To summarize the evaluation results of this section, in the majority of the input fields of the SUTs, an increase in the exploitation rate could be witnessed when changing `G` with `G_c` and it is clear that using constraints results in better quality attack vectors. In addition, IPOG-F gives overall better results than IPOG and the usage of the first algorithm is used by the authors' side when generating tests for web security testing in terms of exploitation rate. Last but not least, the fundamental rule of combinatorial testing is confirmed; testing with higher interaction strength makes likely to reveal more errors. In this context, this rule is interpreted and confirmed in terms of exploitation rate, that is increasing the interaction strength implies higher exploitation rates when these are tested for XSS vulnerabilities.

Combinatorial Coverage Measurement for Web Security Testing

In this section, the simple t-way combination coverage of passing tests derived from the experiments for Gruyere, DVWA and Webgoat in the CCM tool has been computed. In particular, test suites were taken into account that were generated only via `G` as some problems were encountered while parsing the constraints of `G_c` in CCM. The CCM tool offers a user-friendly GUI interface to perform the experiments and has the functionality to print combination coverage charts. It should be noted that in contrast to prior applications of CCM for evaluating combination coverage in large test suites that are not necessarily designed using covering arrays (for example [80]),

Table 5.5: Evaluation results for measuring combination coverage per SUT with increasing strength - IPOG

			IPOG					
SUT		`C(11,t)`	t=2		t=3		t=4	
DVWA	2-way	55	2477/2922	84.77%	2922/2922	100.00%	2922/2922	100.00%
	3-way	165	16676/57812	28.85%	54066/57812	93.52%	57803/57812	99.98%
	4-way	330	46388/716350	6.48%	340821/716350	47.58%	700147/716350	97.74%
Gruyere	2-way	55	2076/2772	74.89%	2771/2772	99.96%	2771/2772	99.96%
	3-way	165	12469/53168	23.45%	46270/53168	87.03%	53086/53168	99.85%
	4-way	330	32522/637878	5.10%	262873/637878	41.21%	601575/637878	94.31%
Webgoat	2-way	55	2610/2997	87.09%	2997/2997	100.00%	2997/2997	100.00%
	3-way	165	18368/60134	30.55%	56710/60134	94.31%	60125/60134	99.99%
	4-way	330	52355/755586	6.93%	368455/755586	48.76%	742208/755586	98.23%

Table 5.6: Evaluation results for measuring combination coverage per SUT with increasing strength - IPOF

			IPOF					
SUT		`C(11,t)`	t=2		t=3		t=4	
DVWA	2-way	55	2507/2922	85.80%	2922/2922	100.00%	2922/2922	100.00%
	3-way	165	18256/57812	31.58%	54223/57812	93.79%	57803/57812	99.98%
	4-way	330	51826/716350	7.23%	349377/716350	48.77%	700739/716350	97.82%
Gruyere	2-way	55	2106/2772	75.97%	2771/2772	99.96%	2771/2772	99.96%
	3-way	165	13637/53168	25.65%	46842/53168	88.10%	53086/53168	99.85%
	4-way	330	36445/637878	5.71%	271808/637878	42.61%	602139/637878	94.40%
Webgoat	2-way	55	2661/2997	88.79%	2997/2997	100.00%	2997/2997	100.00%
	3-way	165	19998/60134	33.26%	57200/60134	95.12%	60125/60134	99.99%
	4-way	330	57843/755586	7.66%	379774/755586	50.26%	743397/755586	98.39%

in the authors' case the passing tests (successful XSS exploits) come from test suites that are produced through combinatorial testing. This feature plays an important role in the evaluation of the passing tests.

In Table 5.5 and Table 5.6 information is given about the combinatorial interaction strength (Str.) of the test suites that the passing tests were originated, the SUT (App), the simple t-way combination coverage for $t \in \{2,3,4\}$ that is to be measured and its respective percentage for the passing tests that their original test suites were generated with the IPOG and IPOG-F algorithms. In addition, the number of t-way combinations is listed in each case denoted by $C(11,t)$. Recall that, combination coverage is measured as the proportion of fully covered t-way combinations of given k parameters in a test suite.

Moreover, in Figures 5.7, 5.8, 5.9 a visualization is given of the combination coverage (in Y axis) with the percentage of combinations reaching a particular coverage (X axis) for DVWA, Gruyere and Webgoat, respectively.

From the evaluation results of this table, for DVWA and Gruyere applications when the simple 2-way combination coverage is measured in the passing tests of the test suites that were generated with interaction strength $t = 3$ and $t = 4$ in both IPOG and IPOG-F algorithms, this selection of passing tests forms a covering array of strength 2 as all 2-way combinations of the 11 variables of the attack grammar are fully covered. Usually, large test suites naturally cover a high percentage of t-way combinations but a case study for web security testing where the passing tests are a covering array is not known to the authors of this work. However, after some post-processing of these results it should be noted that the cardinality of each one of the 11 variables is in some cases smaller than the ones that were presented in Section 5.8.1. This implies that some variable values in the attack grammar do not contribute in revealing XSS vulnerabilities. This could lead to a method to reverse engineer the structure of successful vectors in order to achieve better results in terms of exploitation rate and will be explored further in future research.

For example, when testing Webgoat with the test suites generated via IPOG-F algorithm, for interaction strength $t \in \{2,3,4\}$ it can be seen that **JSO**(1) = `<scr<script>ipt>` does not appear in any of the variable-value configurations of the t-way combinations of the parameters of the attack

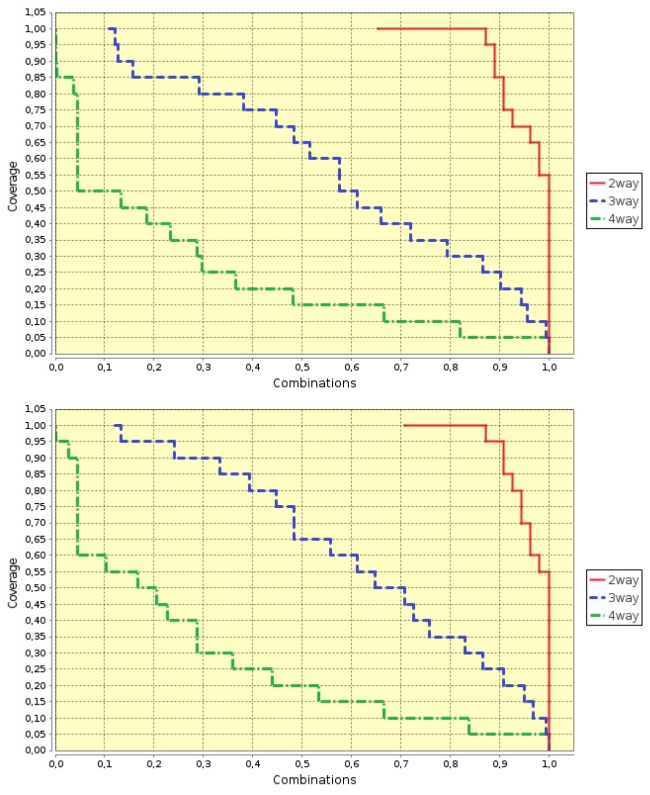


Figure 5.7: Comparison of combination coverage measurement for passing tests in DVWA (inp_id 1, DL 0) when their respective test suites are generated in IPOG (above) and IPOG-F (below) with interaction strength $t = 2$

grammar in passing tests. The first step for attack grammar refinement would be to remove this value from the parameter JSO. At this point, it should be mentioned that is also related to the filter mechanisms of the Webgoat application but since the focus is put on successful attack vectors as a discrete structure will not be elaborated further on this security testing perspective.

Moreover, from the evaluation results in Table 5.5 and Table 5.6 it can be observed that IPOG-F again gives better results than IPOG algorithm. Another point that should be mentioned to conclude this evaluation is that when taking into account the measurement results for simple 2-way combination coverage in the passing tests of the test suites generated with

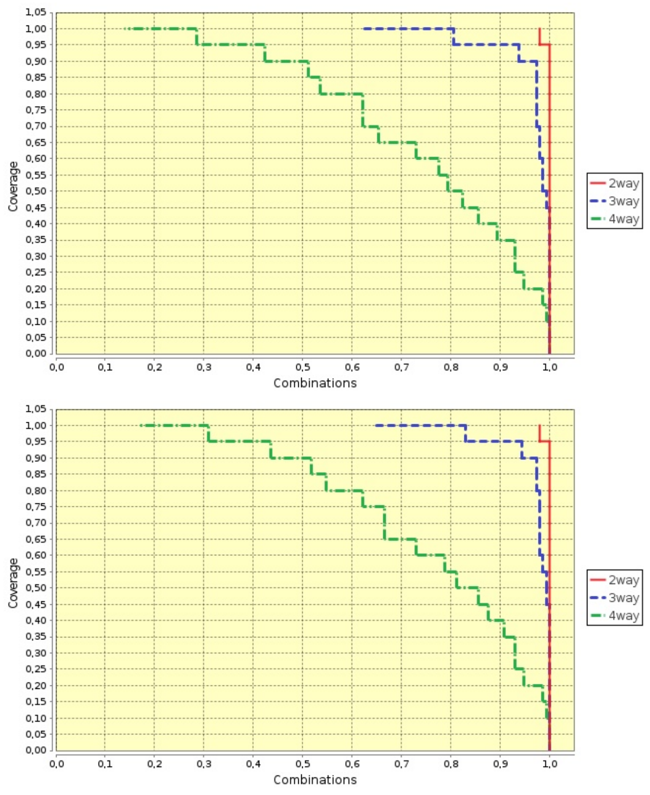


Figure 5.8: Comparison of combination coverage measurement for passing tests in Gruyere (inp_id 1, DL 0) when their respective test suites are generated in IPOG (above) and IPOG-F (below) with interaction strength $t = 3$

interaction strength $t = 2$, some "hidden" variable-value configurations of 3-way and 4-way combinations can be seen appearing. This implies that such combinations will produce successful attack vectors w.r.t. XSS exploits also for higher interaction strengths.

5.9 Automated vs. Manual Testing

In order to reveal vulnerabilities, manual and automatic testing approaches use different strategies for detection of certain kinds of inputs that might

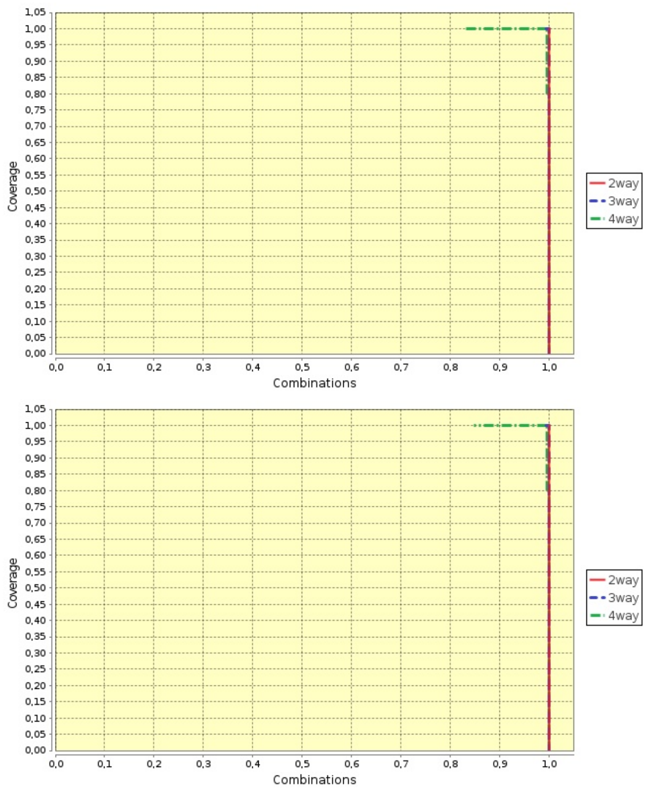


Figure 5.9: Comparison of combination coverage measurement for passing tests in Webgoat (inp_id 2, DL 0) when their respective test suites are generated in IPOG (above) and IPOG-F (below) with interaction strength $t = 4$

lead to a security breach. Here a state-of-the-art manual testing tool is compared to an automated one that is based on model-based testing. The first tool requires user input from the tester whereas the second type reduces the necessary amount of manual manipulation. Both approaches depend on the corresponding test case generation technique and its produced inputs are executed against the system under test.

Manual testing tools provide mechanisms for a user to adapt the testing framework on the SUT by choosing tested elements. An automation of this process is certainly desirable. However, complete automation as of today is still under constant development, even though there are some partly automated tools [20, 102] that support professional penetration testers.

The evaluated results indicate that both techniques succeed in detecting security leaks in web applications with different results, depending on the background logic of the testing approach. Last but not least, the claim of the study is that Attack Pattern-based Combinatorial Testing with constraints can be an alternative method for web application security testing, especially when the authors' method is compared to other test generation techniques like fuzz testing.

Here the main contributions are:

- Detailed evaluation of a case study for web security testing, including automated and manual test execution methods;
- Extensive comparison between the test generation component of the attack pattern-based combinatorial testing technique with various fuzzers.

5.9.1 Penetration Testing Execution Methods

In this section details about the two penetration testing execution methods are provided, automated and manual ones, that have been used in the case study. A description of their test oracles is given as their functionality has been described already in past works. Both methods can be applied to security testing, however here the focus is put explicitly on penetration testing, for example exploiting XSS vulnerabilities, where the main difference (to security testing) relies on the fact that the testing procedure is initiated once the web applications are installed in an operational environment.

- **Attack Pattern-based Testing**: This approach is elaborated in detail in the previous sections.
- **Manual Penetration Testing Tools**: The Burp Suite [5] is an integrated platform for performing security testing of web applications. It is widely used by security professionals since it allows them to perform many penetration testing tasks. The oracle used within Burp Suite was enabled by using the "Search responses for payload strings" configuration option within the intruder. This option flags all results where the payloads were exactly reflected to the response. The rationale behind this decision is that if the vector was not blocked or potential

dangerous characters were not stripped out, the assumption is that a XSS vulnerability was triggered. Additionally, for the cases where stored XSS has been tested for, the option ”Follow redirections:On-site only” is enabled in order to catch the redirections triggered from the injected vectors that were manage to be stored on the server side.

5.9.2 Evaluation

As described in Section 5.8 a more complex XSS grammar has been formulated for the input model in ACTS and inputs were generated for combinatorial interaction strengths $t = \{2, 3, 4\}$ twice. The first dataset consists of inputs that were created without setting any constraints on the model and comprises different test suites depending on the strength while the second one comprises of analogue test suites, which were generated according to constraints introduced in Section 5.8.1. All mentioned applications were tested against both attack inputs and the responsive results have been displayed. In the first case, for interaction strength $t = 2$ the combinatorial tool generated 345 inputs and respectively 4875 and 53706 attack strings for $t = 3$ and $t = 4$. Because constraints put a limitation on the data structure, a remarkably smaller amount of strings were created for the second dataset, namely 250, 1794 and 8761 inputs. Both sets were used in the attack pattern-based approach and in Burp Suite so a comparison could be made according to the results from both cases. In order to draw a meaningful comparison, the same parts of a SUT were tested, which were input fields with textual and password values or textarea tags but attack strings were also attached to the URL paths without any variable binding.

Exploitation Rate of SUTs

In this section it is investigated how the exploitation rate of the different SUTs, which were considered in the case study scales when the combinatorial interaction strength increases, for given difficulty level and input field in each one of the SUTs per penetration testing tool. The evaluation results are depicted in Table 5.7 for the modeled XSS combinatorial grammar (with and without constraints on the derivation types) and when the generated vectors where executed using the attack pattern-based testing method and also with manual penetration testing frameworks (Burp Suite). In particular, it gives information about the combinatorial interaction strength (Str.), the SUT, the input parameter ID (ID), type of vulnerability (VT), eventually the difficulty level (DL), the exploitation rate (the number of positive inputs divided by the total number of tested vectors) and its respective percentage. Again, the names of the SUTs are abbreviated by their corresponding initial letter plus Bitweaver (BW). Further, the table gives the corresponding results for constrained values, where in only a few cases in Gruyere some test runs were not completed due to its unexpected behavior (denoted by "N/A" in the table).

In the majority of the input fields of the SUTs the fundamental rule of combinatorial testing is re-confirmed; testing with higher interaction strengths is likely to reveal more errors (see for example results for WebGoat and BodgeIt). In this context, this rule is interpreted and confirmed in terms of exploitation rate, for example increasing the interaction strength implies higher exploitation rates of web applications when these are tested for XSS vulnerabilities.

Evaluation of Combinatorial Grammars

In almost all of the experiments, a better exploitation rate was achieved by applying constraints upon input generation, which leads to the conclusion that even better results might be achieved by setting even more constraints but also testing with greater combinatorial interaction strengths. For both input sets a somehow higher rate was obtained with increasing t. In all test runs when testing with attack vectors generated with constraints, either a significant increase of the exploitation rate was achieved or exactly the same. In detail, in some test runs the improvement in the exploitation rate is up to 7 times greater than the exploitation rate achieved with the vectors generated by the combinatorial grammar without constraints. The authors' opinion is that the reason the obtained results when constraints were applied are better (from the ones without constraints) because a test suite with better quality was generated, that is by excluding many low quality attack vectors.

Table 5.7: Combinatorial evaluation results for automated and manual tools testing

SUT parameters				Str.	Attack Pattern-based Testing				Manual Testing (Burp Suite)			
					G		G_C		G		G_C	
SUT	DL	VT	ID		ER	% ER	ER	% ER	ER	% ER	ER	% ER
W	0	RXSS	2	2	198/345	57.39	145/250	58.00	177/345	51.30	156/250	62.40
W	0	RXSS	2	3	2842/4875	58.3	1073/1794	59.81	4240/4875	86.97	1672/1794	93.20
W	0	RXSS	2	4	31441/53706	58.54	5366/8761	61.26	47249/53706	87.98	8586/8761	98.00
B	0	RXSS	1	2	175/345	50.72	145/250	58.00	315/345	91.30	250/250	100.00
B	0	RXSS	1	3	2518/4875	51.64	1073/1794	59.81	4445/4875	91.18	1794/1794	100.00
B	0	RXSS	1	4	31441/53706	58.54	5366/8761	61.25	49012/53706	91.26	8761/8761	100.00
B	0	RXSS	2	2	107/345	31.01	97/250	38.80	9/345	2.61	40/250	16.00
B	0	RXSS	2	3	1564/4875	32.08	737/1794	41.08	117/4875	2.40	264/1794	14.72
B	0	RXSS	2	4	20926/53706	38.96	3918/8761	44.72	1279/53706	2.38	1379/8761	15.74
B	0	SXSS	3	2	31/345	8.99	42/250	16.80	57/345	16.52	75/250	30.00
B	0	SXSS	3	3	561/4875	11.51	294/1794	16.39	831/4875	17.05	524/1794	29.21
B	0	SXSS	3	4	6052/53706	11.27	1481/8761	16.90	8996/53706	16.75	2531/8761	28.89
B	0	SXSS	4	2	308/345	89.28	175/250	70.00	0/345	0.00	0/250	0.00
B	0	SXSS	4	3	4434/4875	90.95	1264/1794	70.46	0/4875	0.00	0/1794	0.00
B	0	SXSS	4	4	42899/53706	79.88	6172/8761	70.45	0/53706	0.00	1/8761	0.01
D	0	RXSS	1	2	175/345	50.72	128/250	51.20	315/345	91.30	250/250	100.00
D	0	RXSS	1	3	2517/4875	51.63	954/1794	53.18	4445/4875	91.18	1794/1794	100.00
D	0	RXSS	1	4	27864/53706	51.88	4755/8761	54.28	49012/53706	91.26	8761/8761	100.00
D	0	SXSS	2	2	104/345	30.14	98/250	39.20	150/345	43.48	138/250	55.20
D	0	SXSS	2	3	1364/4875	27.98	565/1794	31.50	2149/4875	44.08	996/1794	55.52
D	0	SXSS	2	4	13862/53706	25.81	2581/8761	29.46	23402/53706	43.57	4739/8761	54.09
D	1	RXSS	1	2	106/345	30.72	80/250	32.00	210/345	60.87	170/250	68.00
D	1	RXSS	1	3	1547/4875	31.73	613/1794	34.17	2966/4875	60.84	1219/1794	67.95
D	1	RXSS	1	4	17172/53706	31.97	3285/8761	37.50	32724/53706	60.93	6223/8761	71.03
D	1	SXSS	2	2	0/345	0	0/250	0.00	0/345	0.00	0/250	0.00
D	1	SXSS	2	3	0/4875	0	0/1794	0.00	0/4875	0.00	0/1794	0.00
D	1	SXSS	2	4	0/53706	0	0/8761	0.00	2/53706	0.00	6/8761	0.07
G	0	RXSS	1	2	122/345	35.36	89/250	35.60	315/345	91.30	250/250	100.00
G	0	RXSS	1	3	1744/4875	35.77	671/1794	37.40	4445/4875	91.18	1794/1794	100.00
G	0	RXSS	1	4	19382/53706	36.09	3303/8761	37.70	N/A	N/A	N/A	N/A
G	0	SXSS	2	2	23/345	6.67	17/250	6.80	50/345	14.49	42/250	16.80
G	0	SXSS	2	3	326/4875	6.69	118/1794	6.58	629/4875	12.90	256/1794	14.27
G	0	SXSS	2	4	3576/53706	6.66	456/8761	5.20	N/A	N/A	N/A	N/A
M	0	RXSS	1	2	111/345	32.17	116/250	46.40	345/345	100.00	250/250	100.00
M	0	RXSS	1	3	1580/4875	32.41	836/1794	46.60	4875/4875	100.00	1794/1794	100.00
M	0	RXSS	1	4	17344/53706	32.29	1833/8761	20.92	53706/53706	100.00	8761/8761	100.00
M	0	RXSS	2	2	158/345	45.8	161/250	64.40	63/345	18.26	83/250	33.20
M	0	RXSS	2	3	2304/4875	47.26	1153/1794	64.27	921/4875	18.89	581/1794	32.39
M	0	RXSS	2	4	25199/53706	46.92	5521/8761	63.02	9803/53706	18.25	2812/8761	32.10
M	0	RXSS	3	2	0/345	0	0/250	0.00	345/345	100.00	250/250	100.00
M	0	RXSS	3	3	0/4875	0	0/1794	0.00	4875/4875	100.00	1794/1794	100.00
M	0	RXSS	3	4	0/53706	0	0/8761	0.00	53706/53706	100.00	8761/8761	100.00
M	1	RXSS	1	2	111/345	32.17	116/250	46.40	345/345	100.00	250/250	100.00
M	1	RXSS	1	3	1580/4875	32.41	836/1794	46.60	4875/4875	100.00	1794/1794	100.00
M	1	RXSS	1	4	17344/53706	32.29	1833/8761	20.92	53706/53706	100.00	8761/8761	100.00
M	1	RXSS	2	2	158/345	45.8	161/250	64.40	63/345	18.26	83/250	33.20
M	1	RXSS	2	3	2304/4875	47.26	1154/1794	64.33	921/4875	18.89	581/1794	32.39
M	1	RXSS	2	4	25199/53706	46.92	5521/8761	63.02	9803/53706	18.25	2812/8761	32.10
M	1	RXSS	3	2	0/345	0	0/250	0.00	345/345	100.00	250/250	100.00
M	1	RXSS	3	3	0/4875	0	0/1794	0.00	4875/4875	100.00	1794/1794	100.00
M	1	RXSS	3	4	0/53706	0	0/8761	0.00	53706/53706	100.00	8761/8761	100.00
BW	0	RXSS	1	2	0/345	0	0/250	0.00	72/345	20.87	84/250	33.60
BW	0	RXSS	1	3	0/4875	0	0/1794	0.00	1269/4875	26.03	588/1794	32.78
BW	0	RXSS	1	4	0/53706	0	0/8761	0.00	14225/53706	26.49	2904/8761	33.15
BW	0	RXSS	2	2	0/345	0	0/250	0.00	72/345	20.87	84/250	33.60
BW	0	RXSS	2	3	0/4875	0	0/1794	0.00	1269/4875	26.03	588/1794	32.78
BW	0	RXSS	2	4	0/53706	0	0/8761	0.00	14225/53706	26.49	2904/8761	33.15
BW	0	RXSS	3	2	198/345	57.39	145/250	58.00	345/345	100.00	250/250	100.00
BW	0	RXSS	3	3	2842/4875	58.69	1073/1794	59.81	4875/4875	100.00	1794/1794	100.00
BW	0	RXSS	3	4	31441/53706	58.54	5366/8761	61.25	53706/53706	100.00	8761/8761	100.00
BW	0	RXSS	4	2	0/345	0	0/250	0.00	345/345	100.00	250/250	100.00
BW	0	RXSS	4	3	0/4875	0	0/1794	0.00	4875/4875	100.00	1794/1794	100.00
BW	0	RXSS	4	4	0/53706	0	0/8761	0.00	53706/53706	100.00	8761/8761	100.00

Table 5.8: Evaluation results for fuzzers and combinatorial testing per SUT for given difficulty level and input field - APBT

SUT parameters				Attack Pattern-based Testing				
				Fuzzers				Best CT
SUT	DL	VT	ID	OWASP	Rsnake	HTML 5SEC	Xenotix	% ER
M	0	RXSS	1	41.59	43.42	32.94	38.91	46.60
M	0	RXSS	2	71.68	60.53	92.94	89.86	64.40
M	0	RXSS	3	2.65	0.00	16.47	9.35	0.00
M	1	RXSS	1	41.59	43.42	32.94	38.91	46.60
M	1	RXSS	2	71.68	60.53	92.94	89.86	64.40
M	1	RXSS	3	2.65	0.00	16.47	9.35	0.00
B	0	RXSS	1	43.36	46.05	32.94	40.88	61.25
B	0	RXSS	2	43.36	46.05	22.35	28.52	44.72
B	0	SXSS	3	12.39	23.68	23.53	28.12	16.90
B	0	SXSS	4	71.68	85.53	40.59	51.60	90.95
G	0	RXSS	1	6.19	3.95	14.12	19.23	37.70
G	0	SXSS	2	0.00	0.00	7.06	11.25	6.80
W	0	RXSS	2	43.36	46.05	33.53	41.39	61.26
D	0	RXSS	1	41.59	43.42	31.76	40.88	54.28
D	0	SXSS	2	39.82	31.58	30.59	27.73	39.20
D	1	RXSS	1	41.59	43.42	20.59	27.99	37.50
D	1	SXSS	2	0.88	0.00	0.59	2.48	0.07
BW	0	RXSS	1	0.00	0.00	0.00	0.00	0.00
BW	0	RXSS	2	0.00	0.00	0.00	0.00	0.00
BW	0	RXSS	3	43.36	46.05	32.94	41.14	59.81
BW	0	RXSS	4	0.00	0.00	0.00	0.00	0.00

Table 5.9: Evaluation results for fuzzers and combinatorial testing per SUT for given difficulty level and input field - Burp Suite

SUT parameters				Manual Testing (Burp Suite)				
				Fuzzers				Best CT
SUT	DL	VT	ID	OWASP	Rsnake	HTML 5SEC	Xenotix	% ER
M	0	RXSS	1	99.12	97.37	93.53	88.82	100.00
M	0	RXSS	2	39.82	30.26	61.76	63.86	33.20
M	0	RXSS	3	83.19	82.89	74.12	78.37	100.00
M	1	RXSS	1	99.12	96.05	93.53	88.82	100.00
M	1	RXSS	2	39.82	30.26	61.76	63.86	33.20
M	1	RXSS	3	83.19	82.89	74.12	78.37	100.00
B	0	RXSS	1	98.23	97.37	90.59	85.69	100.00
B	0	RXSS	2	7.96	0.00	5.29	1.11	16.00
B	0	SXSS	3	46.02	15.79	0.00	0.00	30.00
B	0	SXSS	4	0.00	0.00	0.00	0.00	0.01
G	0	RXSS	1	83.19	82.89	74.12	78.37	100.00
G	0	SXSS	2	6.19	3.95	6.47	26.41	16.80
W	0	RXSS	2	39.82	57.89	42.35	76.67	98.00
D	0	RXSS	1	83.19	82.89	74.12	78.37	100.00
D	0	SXSS	2	92.04	93.42	84.12	68.56	55.52
D	1	RXSS	1	83.19	82.89	60.00	66.93	71.03
D	1	SXSS	2	0.00	0.00	0.00	0.00	0.00
BW	0	RXSS	1	26.55	40.79	8.24	26.73	33.60
BW	0	RXSS	2	26.55	40.79	8.24	26.73	33.60
BW	0	RXSS	3	99.12	96.05	93.53	88.82	100.00
BW	0	RXSS	4	99.12	96.05	93.53	88.82	100.00

Comparison of Fuzzers and Combinatorial Testing

In this section, the focus of the evaluation is changed and now an investigation is aimed how the authors' test suites generated by combinatorial testing compare to fuzzers. A number of such test suites (produced by fuzzers) have been collected, which are publicly available and executed them against the same SUTs using both automated and manual test case execution methods. The exploitation rate is compared for the vectors produced with combinatorial and fuzz testing within the same test execution method (when these are tested against the same web application), in order to draw more accurate conclusions. In the evaluation results presented in Table 5.8 and Table 5.9 the best exploitation rate achieved with combinatorial testing for the same test run from Table 5.7 is taken into account, denoted by best CT % ER. When comparing the exploitation rate of fuzzers and combinatorial testing in Table 5.8 and Table 5.9, as before, information for the SUT (App) is given, the input parameter ID (inp_ID), type of vulnerability (VT) for the attack pattern-based testing method and also with the manual testing tool, Burp Suite. In particular, the following resources have been considered:

1. OWASP XSS Filter Evasion Cheat Sheet[1] with 113 vectors.
2. Attack and Discovery Pattern Database for Application Fuzz Testing[2] (rsnake) with 76 vectors.
3. HTML5 Security Cheat Sheet[3] with 170 vectors.
4. OWASP Xenotix XSS Exploit Framework[4], where its 1530 vectors were extracted.

Comparing the exploitation rate that the different test inputs achieve against the SUTs in both tools, it can be argued that the diversity of the vectors generated with combinatorial testing has achieved better results than fuzzers in some cases. In other words, the fact that test suites with different sizes (attributed to the interaction strength) can be generated for given SUTs

[1]https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Accessed: 2015-12-28.

[2]https://code.google.com/p/fuzzdb. Accessed: 2015-12-28.

[3]https://html5sec.org/. Accessed: 2015-12-28.

[4]https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework. Accessed: 2015-12-28.

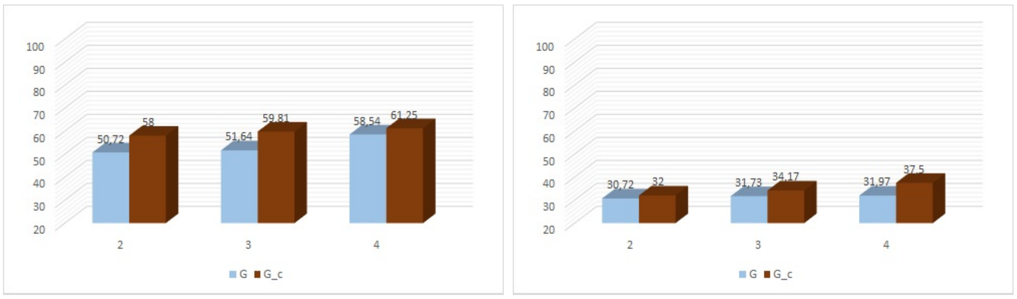


Figure 5.10: Coverage vs. interaction strength with Attack Pattern-based Combinatorial Testing for BodgeIt (left) and DVWA (right)

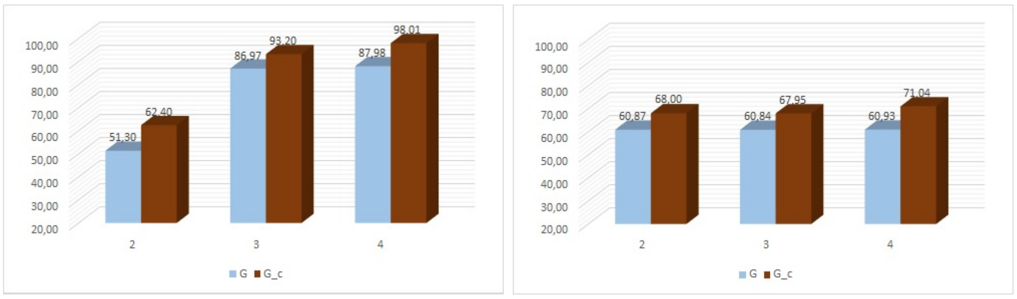


Figure 5.11: Coverage vs. interaction strength in Burp Suite for WebGoat (left) and DVWA (right)

offers a larger attack surface when compared to the one achieved with fuzz testing. Moreover, the use of constraints in the attack grammar can filter out some low quality attack vectors and this is another reason why combinatorial testing outperforms fuzz testing in some test runs. Clearly, these two features of combinatorial testing cannot be achieved with fuzz testing. It should be noted as well that even in the case where combinatorial testing and fuzz testing achieve the same exploitation rate, in practice the number of actual positive inputs differs since the size of test suites generated with combinatorial test suites is quite larger. To give an example, for $t = 4$ there are 8761 attack vectors when G_c is considered. To conclude with, it is evident from Table 5.8 and Table 5.9 that in half of the test runs in both automated and manual test execution methods, inputs generated with combinatorial testing achieve better exploitation rate.

Comparison of Automated vs. Manual Test Execution Methods

The obtained results rely heavily on the test oracles from the tools. While testing with Burp, there have been several cases where a detection rate of 100% is noticed. This would indicate that every submitted vector was able to trigger a vulnerability. In other words, both testing procedures have produced a certain amount of false positives, which influenced the final results. The used tools were not able to detect such potential outcomes in an automatic manner. The authors' assumption is that the obtained results depend more on the test execution method rather than the quality of the inputs due to different mechanisms that take place on the test oracles of the testing tools. This argument is further supported by the fact that differences were noticed on the results not only when comparing the combinatorial grammars themselves, but also when comparing the results that were obtained when testing the vectors that have been produced by fuzzers.

It is intriguing to investigate whether in the authors' experiments the findings of [85] are confirmed, which states that imposing constraints on real-world applications make higher strength combinatorial interaction testing feasible (in the sense that they reveal more subtle XSS attacks).

This motivating question led the authors to revisit the notion of attack pattern-based combinatorial testing by including constraints in the combinatorial grammar in the first place. Figure 5.10 and Figure 5.11 demonstrate that an increase of the interaction strength implies an increase of the coverage when testing for XSS vulnerabilities, for both automated testing and manual testing approaches. In these figures the coverage results obtained by the XSS grammars are denoted with `G` and `G_C`, without and with constraints, respectively.

5.10 Summary

An approach based on attack patterns has been introduced in this chapter. In contrast to other work in security testing based on UML models, the attack pattern-based approach does not rely on the behavior of the system but instead attack patterns are modeled. By executing the model against the applications, the approach proved that testing a program against known SQLI and XSS vulnerabilities is possible. The whole process can be automated using tools for executing UML models. The most important fact is that the model, once specified, executes fully automatically and reports its status.

Further, the mentioned approach is combined with a test case generation method from the area of combinatorial testing. The main task for expanding this technique into a more general approach for automatically XSS detection is to put more emphasis on the extension of an attack grammar for XSS although this presents a greater challenge because of the variety of possible inputs and an undefined structure of such strings.

Additionally, an input grammar was revised for combinatorial generation of test inputs and constraints were adapted for another test suite in more detail. These two sets of input data were used by both an automatically and a manually testing approach in order to test several programs for XSS. The conclusion is highlighted that testing with combinatorial attack grammars with increasing interaction strength results in higher exploitation rates. Setting constraints upon the input model also results in significantly improved attack vectors. However, in order to further improve the testing results in the attack pattern-based technique, a revisited test oracle might be taken into consideration since some discrepancies were witnessed with manual penetration testing methods.

According to the elaborated approach, several of the asked research questions are being answered.

RQ1: How can all necessary attack information be formalized?

Once necessary attack information is gathered, this can be formalized in form of a pattern. A formal definition of such an attack pattern is given in form of a UML state machine. All necessary attack information is implemented in the model as parts of transitions with corresponding variables and method calls. The relevant aspects of modeling are discussed and it is shown how SQL injection and reflected as well as stored XSS attacks can be modeled. Because of the flexibility of the model, future adaptations are easily possible, for example by incorporating mutation functions, encoding types or evasion of special symbols.

In addition to this attack depiction, Section 6.3 will explain another representation.

RQ2: What testing methodologies are developed for vulnerability detection in web applications?

The first approach, called Attack Pattern-based Combinatorial Testing, comprises the area of combinatorial testing with the emphasis on test case generation for attack vectors and the attack pattern-based testing technique for test case execution against web applications. The generated vectors were automatically tested against the SUTs with different combinatorial interaction strength with highly promising initial results. In addition, new elements may be applied into the attack pattern model, for example mutation functions, encoding types or evasion of special symbols.

The second approach will be explained as an addition to this research question in Section 6.3.

RQ3: How does Attack Pattern-based Combinatorial Testing perform in comparison to fuzzing and manual testing approaches?

Three main benefits of the proposed approach are identified and confirmed when compared to fuzz testing, especially with regard to combinatorial test generation.

- The quality of the XSS attack vectors generated from the combinatorial testing procedure is achieved through the covered combinations of different types in a test suite while with fuzzing a designer is not able to target a specific combination of type values.
- The diversity of the XSS attack vectors generated from the proposed approach is achieved through the different number of attack vectors per test suite, a direct implication of the interaction strength parameter.
- Last but not least, combinatorial testing is motivated by a selection of a few tests such that good coverage is still achievable while still being able to reduce the search space. In this regard for $t = 2$, as was already shown in Section 5.8.1, a reduction of 99.99% of the total search space has been achieved.

On the other hand, when comparing the results from the automated and manual approaches, following observations are made:

- On the performance side, the evaluated manual tool is faster in executing each test run when executing automatically. However, Burp required some manual set-up in order to execute the huge amount of attack vectors in an automated fashion. On the other hand the tool offers the possibility to set the number of threads that will run concurrently when executing a test run, so the execution time was shorter than its automated counterpart.
- Slighty better test results are obtained when testing with Burp Suite. However, as mentioned in Section 5.9.2, both testing methods possibly generated a certain number of false positives that remained undetected by both approaches.

Although the Attack Pattern-based Combinatorial Testing approach may have some drawbacks, its obtained testing results confirm the authors' opinion that the approach can be used and in some cases even outperform the already established testing practices. From the comparison of the combinatorial grammars against ones used for fuzz testing the conclusion is drawn that the proposed approach can be seen as an alternative method for revealing XSS vulnerabilities in web security testing. A redefinition of some features, for example the test oracle and the input model as well as setting additional constraints on the input grammar could lead to even better results.

6 Testing as a Planning Problem

The adaptation of the attack models is not easy and requires substantial effort. In order to make modeling easier, it is suggested to represent attacks as a sequence of known actions that have to be carried out in order to be successful. Each action has some preconditions and some effects. Hence, it is possible to represent testing in this context as a planning problem where the goal is to break the application under test.

Here a method is introduced that is based on planning for computing test cases where a test case is a sequence of interactions with the web application under test. The underlying idea of using planning for test case generation originates from two sources. First, there is already publication available describing testing as a planning problem. Second, and even more important, when having a look of how to break a system, it becomes obvious that providing and executing an attack is nothing else than finding an interaction sequence that finally leads to a situation where a vulnerability can be exploited.

The contributions of this chapter are the following: A presentation is given for an approach for test case generation and execution in the security domain that is based on planning. An algorithm is presented that makes use of a planner for generating test cases, which are executed after generation. The approach is illustrated by using a small example and the empirical evaluation is discussed, indicating that the approach has similar capabilities for detecting vulnerabilities of web applications than previous approaches whereas the new approach is easier to adapt and extend.

Additionally, a contribution is made by presenting the tool PURITY (Planning-based secURITY testing tool) for testing web applications. PURITY executes test cases against a given website while the test execution proceeds automatically. In contrast to other penetration testing tools, PURITY relies on planning. In addition, it also allows a tester to configure input parameters and also tests a website in a manual manner.

Finally, the chapter contributes to the answers for the research questions RQ1 and RQ2 and gives an answer for RQ4 (see Chapter 1).

The original works for this chapter are [37, 39].

6.1 The Security Testing via Planning Approach

Planners are commonly used for intelligent agents and autonomous systems in order to generate action sequences that lead a system from the initial state into a defined goal state. Once specified, these plans instruct the system what to do in each step as long as all actions can be undertaken and typically considering that the environment does not change during plan execution. In the proposed security testing approach the generated plan for testing web applications is used with respect to the well-known vulnerabilities: SQLI and reflected as well as stored XSS. For this purpose the test case generation problem is specified as a planning problem. First, the planning problem is defined in the classical way in Section 3.1 as Definition 2, following [55].

An action a can be executed in a state S if and only if its precondition $pre(a)$ is fulfilled in S. If an action a can be executed, then the execution moves to a new state S' comprising all predicates that are in S and do not contradict $eff(a)$ and all predicates of $eff(a)$. In this case it is given as $S \rightarrow_a S'$.

Definition 3 in Section 3.1 defines the solution of a planning problem.

In classical planning it is assumed that there is atomic time, that is, the execution of an action can be done in finite time and no interruption is possible, there are no exogenous events, the action effects are deterministic and there is omniscience on the part of the agent. In the context of the proposed application all these assumptions are (more or less) fulfilled, when assuming stateless applications.

In order to state security testing as a planning problem, the following representation is suggested:

- Each action that can be performed by an attacker has to be modeled as a planning action, considering the preconditions and the potential effects.
- The initial state considers the currently available information of a web application, that is, the web address, the script to be executed, and the parameter to be used, etc.
- On the other hand, the goal state specifies what to expect from an application in case of a detected vulnerability.

When specifying the security information as a planning problem, the problem of generating tests immediately becomes a planning problem. Every plan is a test case comprising the actions necessary to be carried out in order to detect a vulnerability. This approach is very flexible because it allows for easy adaptation. Every time new information about other attack actions is available, they can be integrated into the set of actions. The plans can be generated once more taking care of the new actions. Moreover, if designed in a good way each action can be used for testing different applications. For this purpose each action definition has to be as general as possible. In this way, also reuse is supported.

In order to implement the proposed approach, it relies on ordinary planner and planning languages. In particular it is assumed to use the Planning Domain Definition Language (PDDL) in order to specify the corresponding domain. That is, the actions those are problem independent, and the problem file, that is, application specific values, the initial state, and the goal state, which are specific to a certain application.

Every action definition in the domain consists of a list of parameters and preconditions as well as the resulting effects. In case the initial values satisfy a specific precondition from some action, this action is put on top of the planner. Because of the execution of the action, its effects might change some values, which may lead to the satisfaction of preconditions from some other action. The action generation continues as long the specified goal is not reached, thus generating a plan. Otherwise the problem is considered improvable.

In order to adapt the planning-problem to security testing, a specific domain and problem description have to be defined. Furthermore, a generated plan is considered as one abstract test case. An abstract test case is a test case that cannot directly be executed by the SUT. This is due to the fact that concrete values are missing or that the abstract actions do not provide any information on how to execute them in the current environment. In order to solve this issue and to come to a concrete test case, for each action a corresponding method is specified in Java. This method implements the interaction with the SUT and makes use of concrete values. As described in the previous sections, the message traffic between tester and application is handled by using HttpClient and jsoup for parsing of responses, thereby relying on detection mechanisms for SQLI and XSS.

6.1.1 PLAN4SEC

In the following the algorithm **PLAN4SEC** is discussed, which is for implementing the described approach. **PLAN4SEC** makes use of an ordinary planner. In this implementation the planning system Metric-FF (see Section 3.2) is used, which itself relies on the FF planner [7]. However, the approach is not limited and other planners can be used as well. PLAN4SEC was already introduced by the authors in [37]. The approach behind the automated execution in PURITY was extended in order to cover additional functionality. The improved PLAN4SEC 2.0 is depicted in Algorithm 1.

The algorithm uses the domain D and the set of problem files P as inputs. Moreover, other information is used as well, that is, the URL address (URL), the set of initial values (U) that encompasses the type of attack (T) and the HTTP method (M) to be used. Additionally, it takes the set of attack vectors (X) and concrete actions (C) as well as a function Φ mapping actions to their corresponding Java method. The output of the algorithm is a set of plans (PL), a set of obtained HTML elements (E) during execution. The final output is a table (V) with all attack vectors and SUT parameter values that lead to a vulnerability breach. The corresponding function *res* reports $FAIL$ whenever a test triggers a vulnerability, whereas $PASS$ is thrown otherwise.

The main improvement of **PLAN4SEC 2.0** is a dynamic PDDL generation, crawler consideration and processing of new outputs during the execution.

Algorithm 1 PLAN4SEC 2.0 – Improved plan generation and execution algorithm

Input: Domain D, set of problem files $P = \{\mathtt{p}_0, \ldots, \mathtt{p}_n\}$, address URL, set of initial values $U = \{(\mathtt{t},\mathtt{m}) | \mathtt{t} \in T, \mathtt{m} \in M\}$ with a set of attack types $T = \{\mathtt{t}_0, \ldots, \mathtt{t}_n\}$ and set of HTTP methods $M = \{\mathtt{m0}, \ldots, \mathtt{m}_n\}$, set of attack vectors $X = \{\mathtt{x}_0, \ldots, \mathtt{x}_n\}$, set of concrete actions $C = \{\mathtt{c}_0, \ldots, \mathtt{c}_n\}$ and a function $\Phi = a \mapsto c$ that maps abstract actions to concrete ones.
Output: Set of plans $PL = \{\mathtt{A}_0, \ldots, \mathtt{A}_n\}$ where each $A_i = \{\mathtt{a}_0, \ldots, \mathtt{a}_n\}$, set of HTML elements $E = \{\mathtt{e}_0, \ldots, \mathtt{e}_n\}$ and a table with positive test verdicts V.

```
1: $PL = \emptyset$
2: for $SELECT\ URL, X, C, U, p \in P, D$ do
3:     while $URL.hasNext()$ do
4:         $E =$ parse$(URL)$                              ▷ Identify user input fields
5:         while $U \neq \emptyset$ do
6:             $A =$ makePlan$(p, D)$
7:             $PL = PL \cup \{A\}$
8:             $res(A) = FAIL$
9:             for $x' \in X$ do
10:                for $e' \in E$ do
11:                    for $a \in A$ do                    ▷ Execute plan
12:                        $a' =$ ConcreteAct$(a, \Phi, x', e')$
13:                        if Exec$(a')$ fails then
14:                            $res(A) = PASS$
15:                        else
16:                            $res(A) = FAIL$
17:                            $V = V \cup res(A)$
18:                        end if
19:                    end for
20:                end for
21:            end for
22:            $p =$ makePDDL$(U, p, D)$                  ▷ New problem
23:            $P = P \cup p$
24:        end while
25:        $URL =$ crawler.next$()$                        ▷ Pick next URL
26:    end while
27: end for
28: Return $(V)$ as result
```

The last point represents information that cannot be foreseen before the testing starts. However, it is applied dynamically into the testing process.

The idea behind this algorithm is the following one. For every URL address the program parses user input elements from the website as well as the current initial values from the problem's PDDL. Additionally it initializes the crawler, which in time returns all hyperlinks from the website in form of URLs. Now the program checks whether HTML elements have been encountered during the parsing of the website in step 4 (E). As mentioned before, these are the input fields where the user is supposed to interact with the SUT. The goal is to test every of these elements separately before continuing the execution. Since these values cannot be known at the beginning, the program has to identify them for every incoming URL. Now the planner returns the first sequence of actions from the domain and problem files (step 6). Afterwards the first attack vector is picked from the input files.

The function Φ takes as arguments the abstract action (a) from the plan and maps it to its concrete counterpart in Java (c).

During plan execution, the test case generator assigns the attack vector (x') to one of the HTML inputs (e') from the website. Afterwards, when the plan execution terminates, the program still remains in the loop of that vector but assigns it now to another HTML element in E and repeats the plan execution again from the beginning (steps 10-20). Now, generated abstract actions are read one by one from the saved plan. PURITY traverses through all concrete Java methods in order to find the corresponding action implementation (step 12). When encountered, it is executed and eventually generates new values.

In such a way a plan is re-run for a certain address and a certain attack vector several times. Only after testing of all user input elements, the execution proceeds further. In case that no input elements are available, the program switches immediately to the next part. After the tests have been executed for all input elements, eventually a positive test verdict is saved into the table (step 17). A concrete example for this table is given in the Section 6.2.4.

A major difference to the initial version of the algorithm is the fact that PURITY generates and executes several problem definitions. In fact, a new

PDDL file is generated dynamically after all attack vectors have been executed against one web page. For this case let's take a look at the initial values of an individual problem definition. These specify the starting conditions for further plan generation. A sample of a few initial values is given below:

```
(:init
(inInitial x)
(Logged no)
(not (statusinit two))
(Type sqli)
(= (sent se) 0)
(not (Empty url))
(GivenSQL sqli)
(GivenXSS xssi)
(Method post)
(Response resp)
(not (Found exp resp))
(not (FoundScript script resp))
...
)
```

Initial values description in PDDL

The goal is to generate new problem files with different initial values so that different plans are generated as well. With a new sequence of actions the test execution will also differ from the previously plan. During execution the problem file is parsed in search for an already set initial value, which will be replaced by a new one from the corresponding data set from U. It should be mentioned that the same set of values is specified twice, once in the PDDL files and at the concrete level in Java.

For demonstration purposes the change of two of the initial values are explained, namely `Type` and `Method`. Here `Type` can have three values, namely `sqli`, `rxss` and `sxss` whereas `Method` encompasses only `get` and `post`. Both sets are implemented in the PDDL files as well in Java on the concrete side. If `sqli` was the initial value of `Type` in the first problem file then another will take its place, for example `rxss`. The program will keep the implementation of the current problem (p) but will replace its current value (e.g. `sqli`) with a new one (`rxss`) in step 22. Then, a new PDDL file is saved (step 23) with this specification and marked as next in line for procession. The plan generation is invoked again as well as the attacking sequence.

However, one important attribute of PURITY is that a method for the generation of initial values is directly invoked from another method of the same kind. This means that for every value of `Type` several files with different initials are obtained. After all of them have been executed, new files for a new value of `Method` is generated. For example, for three values of `Type` and two different values of `Method` a total of six PDDL files are generated, which results in six plans and attack executions. Theoretically, by taking the values of one more initial predicate, e.g. `(inInitial x)` with 22 possible values for `x`, a sum of ($3 \times 2 \times 22 =$) 132 problem files can be generated and so on. It is important to note that for every abstract value the corresponding concrete values have to be set as well. For example, if the method from the plan is `post` then the website will be tested only with that method despite the fact that `get` might be its default value. But since there has to be at least one file where `(Method get)` is specified, this will be executed as well.

Actually, in this way eventually an unwanted behavior of the SUT is triggered and tested whether this might lead to a security breach as well. The more initial values are specified and manipulated, the more different tests are carried out for this sake. This PDDL generation process will continue as long as all combinations of objects from U are executed.

In fact, this principle can be applied to every initial value so a huge number of test cases are generated. Of course, the last method would call no other because all value combinations would have been already executed. However, it remains the task of the programmer to implement new generation methods.

The entire testing process will last as long as the crawler returns new hyperlinks. Afterwards, the execution terminates permanently.

6.1.2 Running Example

A demonstrate of the approach is given by using DVWA. The plan generation process is explained as well as the execution of plans. As mentioned in the previous section, first the problem and domain files are specified manually, accordingly to the testing purpose and the current SUT. Have a look at the following PDDL description of the problem:

```
(define (problem mbt-problem)
(:domain mbt)
(:objects
x - active
s - server
si - status-si
lo - status-lo
se - status-se
type - type
url - address
m - method
a - action
exp - expect
un - username
pw - password
sqli - sqli
xssi - xssi
script - script
resp - response
)
(:init
(inInitial x)
(Logged no)
(not (statusinit two))
(Type sqli)
(= (sent se) 0)
(not (Empty url))
(GivenSQL sqli)
(GivenXSS xssi)
(Method post)
(Response resp)
(not (Found exp resp))
(not (FoundScript script resp))
)
(:goal (inFinal x))
)
```

Problem description in PDDL

This PDDL description of the problem contains the problem definition, the domain reference, the objects that are used in the domain specification, the initial values and finally, the goal specification. The objects are of certain types, which are set in the domain definition. For the initial values various necessary parameters like the type of attack, the used HTTP method, the current position in the execution process, the indicator whether an input for SQLI or XSS is specified etc. are taken into consideration.

When using this PDDL description together with the following partially description of the domain, the planner is able to generate a plan.

```
(define (domain mbt)
(:requirements :strips :typing :equality :fluents :adl)
(:types active address server status-si status-lo
   status-se type expect result method integer sqli
   xssi response script)
(:constants init - active no yes - status-lo two -
   status-si sqli rxss sxss - type get post - method)
(:predicates
(inInitial ?x)
(inAddressed ?x)
(inSentReq ?x)
(inRecReq ?x)
(inParse ?x)
(inSQLI ?x)
(inRXSS ?x)
(inSXSS ?x)
(GivenSQL ?sqli)
(GivenXSS ?xssi)
(inFinal ?x)
)
(:functions
(statusinit ?si - status-si)
(Method ?m - method)
)
(:action Start
:parameters(?x - active ?url - address ?lo - status-lo)
:precondition (and (inInitial ?x)(not (Empty ?url)))
:effect (and (inAddressed ?x)(not (inInitial ?x))
   (Logged yes))
)
(:action SendReq
:parameters(?x - active ?lo - status-lo ?se - status-se
    ?si - status-si)
:precondition (and (inAddressed ?x) (Logged yes))
:effect (and (inSentReq ?x)(not (inAddressed ?x))
   (assign(sent ?se)1)(statusinit two)))
)
(:action Finish
:parameters (?x)
:precondition (inFound ?x)
:effect (inFinal ?x))
)
```

Domain description in PDDL

In the domain description at the beginning all possible requirements are listed, in order to allow different planners to use PDDL. Afterwards, all object types are initialized whereas the objects themselves are defined in the problem definition. In the PDDL code constants define special values for some of the types. Predicates are logical functions that affect certain objects, whereas functions specify entities that can change their value during plan execution. Finally, the actions are constructed with the definition of used parameters within that action, the precondition and the postcondition. At the beginning of plan generation, the planner will take the initial values and search in the action table for satisfied preconditions. If such an action can be found, it will be added to the current plan. When taking this action, the corresponding effects might change some relation, for example the action *Start* changes the active position from *inInitial* to *inAddressed*. After updating the state, the planner searches for new possible actions to be taken. This process might continue as long as the goal from the problem definition is not reached or the planner notices that it cannot be attained. In the latter case, no plan can be delivered back.

For the running example the following plan can be computed using Metric-FF:

```
0: START X URL LO
1: SENDREQ X LO SE SI
2: RECREQ X SI
3: PARSE X M USERNAME PASSWORD TYPE
4: CHOOSERXSS X TYPE
5: ATTACKRXSS X XSSI M UN PW
6: PARSERESPXSS X SCRIPT RESP
7: PARSERESPXSSCHECK X SCRIPT RESP
8: FINISH X
```

Generated plan

Such a generated plan is read by the parser from JavaFF (see Section 3.2). Names of the actions are translated into names of the corresponding Java functions. Note that in the implementation the action names and the names of their corresponding Java functions are the same. The Java functions are executed step by step. In addition the Java functions use concrete values for parameters and communication with the web application.

For example, let's assume that the URL address of the SUT is specified and SQLI is chosen in the program. In this case the initial values $(not(Empty\ url))$ and $(GivenSQL\ sqli)$ are satisfied, *sqli* being the attack vector. The object x from the type *active* is meant to give the current status of the execution, for example *inInitial* states that the execution has just started. When analysing the action definitions, all preconditions are met in order to satisfy the action *Start*. Now the program runs its corresponding method and then reads the next action, executing its next methods afterwards,

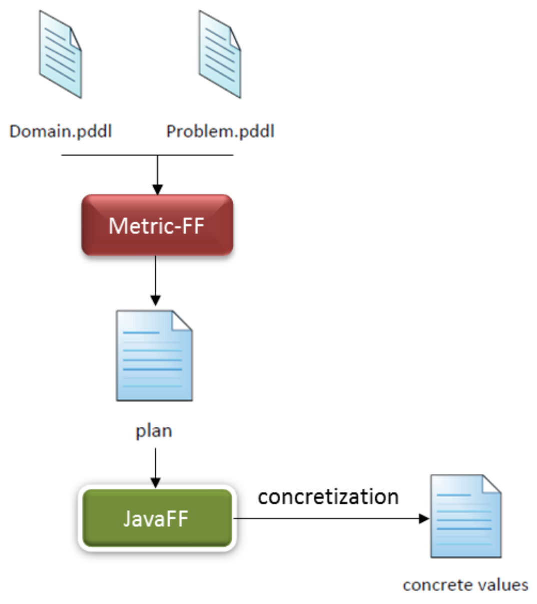


Figure 6.1: Plan generation and concretization

thereby manipulating concrete variable by it's own. When picking the action *SendReq*, the program will send a HTTP request with HttpClient to the URL address. But, if the tester has not specified this in the program, a discrepancy emerges between the plan and the program. In that case, the program will not be able to follow the plan until the end and the execution stops immediately, setting the plan execution to *PASS* because of not reaching a vulnerable state. Otherwise, execution continues until reaching the final action of the plan.

Figure 6.1 depicts the entire plan generation and concretization process from the abstract point of view.

6.1.3 Evaluation

In order to provide an evaluation, the proposed planning approach to security testing has been tested on some of the already familiar web applications. For the evaluation, the **PLAN4SEC** algorithm was implemented in Java.

A domain and problem file similar to the ones in Section 6.1.2 were used but they were extended substantially for the evaluation. In sum, 19 action definitions as well as more predicates and initial values were taken into consideration. For carrying out the whole evaluation, three values for the attack type, two for the method, two for the login status and 20 for the current status were used. For every type of attack, only one attack vector was used for the concrete test cases. The objective of the evaluation was to show the applicability of the approach both in running time as well as the capabilities of detecting vulnerabilities.

The obtained results are depicted in Table 6.1 where for each SUT, the difficulty level (DL), the total time (T) for carrying out the tests, the number of generated plans (#P), the total planning time (planT), the average plan generation time (avgPT), the number of generated actions (#A), the total plan execution time (execT), the average number of generated actions per plan (avgA), and information of how often SQLI, RXSS, and SXSS attacks have been successful, are given. All time values are in second (s).

When executing **PLAN4SEC** on DVWA and Mutillidae, vulnerability could only be triggered on the first two security levels. In both cases, the third one remains impervious. Because of this reason, no row is added for the third level of these two applications in Table 6.1. It is worth noting that stored XSS could not be detected on the second level of DVWA too. This is due to the fact that the used input string was successfully filtered by the application. The second application BodgeIt has much more SQLI leaks but seems to be more resistant against XSS.

It can be seen that the time for executing **PLAN4SEC** is acceptable. The approach is performed automatically only requiring the user to specify case specific information, like different URL addresses and different expected values for SQLI. Beside the small amount of adaptation, no further changes were required. This holds especially for the domain specification, which is the same for all SUTs in this evaluation. Note that the Java methods, which correspond to the actions, have to be slightly changed. The unchanged domain specification is also the reason behind the same number of generated plans for all applications.

The success of the exploitations heavily depends on the used input. Despite the fact that the system is being tested by using different interactions for

Table 6.1: Evaluation results for planning-based testing

SUT	DL	T	#P	planT	avgPT	#A	execT	avgA	SQLI	RXSS	SXSS
DVWA	1	355.10	273	292.06	1.07	972	49.41	3	29	30	30
	2	835.70	273	739.70	2.71	972	57.51	3	29	30	0
BodgeIt	na	357.38	273	308.53	1.13	972	20.99	3	53	18	20
Mutillidae	1	309.56	273	288.44	1.06	972	13.44	3	31	30	25
	2	316.91	273	292.89	1.07	972	13.76	3	31	30	20

checking exploits, there is still a need for convert values to be executed. These values have to be adapted (maybe randomly) during execution, which is currently not done. As mentioned before, only one attack vector has been used for the evaluation. However, it is worth noting that this is not a principle restriction of the proposed approach.

Because of the relatively high number of potential combination of different input parameters, a higher number of plans was received but also a larger number of successful tests. What might also be interesting is the fact that the average number of actions is rather small. This indicates that the action definitions in the domain specifications use only a small number of preconditions and also originates from the underlying plan generation technique, that is, Metric-FF.

In this evaluation, the algorithm has to terminate assuming that all called functions terminate. This is due to the fact that all input sets are finite, determining the number of iterations. The algorithm is polynomial in space and time when assuming the execution of external functions in unit time. Hence, when using the generated test suite for regression testing purposes only the execution time has to be considered, which is the result of the **PLAN4SEC** algorithm.

When this approach and the results given in Table 6.1 are compared with the results obtained using the previous method relying on models of attack patterns, a similar behavior is obtained regarding the detection capability for vulnerabilities for the same web applications. Hence, when considering the much more easy adaptation of the model to different SUTs, the planning based approach is indeed an improvement and worth being further investigated.

6.2 PURITY: a Planning-based secURITY testing tool

In order to make security testing of web application easier, the penetration testing tool *Planning-based secURITY testing tool* (PURITY) is proposed. It has been developed for testing of websites to detect potential SQLI as well as reflected and stored types of XSS security issues in an either manually or automated fashion (or something in-between). The tester is asked for a minimum amount of informations. PURITY also offers the possibility to define all test parameters if desired. The tool is partly built upon previously work already explained. It combines Attack Pattern-based Combinatorial Testing with planning. In fact, PURITY improves test case generation using a planner and makes use of the communication implementation and test oracles from the previously works. The tool presents the obtained test results in detail after the test execution terminates. It represents a security testing tool that is easy to use but also provides high configurability and offers extendibility. It is a research prototype written in Java.

In Figure 6.2 the tool is depicted in context with its surrounding environment. It takes inputs from the user like the www address of the application, PDDL files, which define the initial state and potential attack actions, and potential concrete attack vectors used when testing the application. PURITY generates plans from which concrete test cases to be submitted to the SUT. However, it also analyses the received feedback from the SUT in order to detect vulnerable behavior.

PURITY encompasses several elements that interact with each other as well as with the user, which will be described later in more detail. It offers additional possibilities for the tester to define test parameters like the type of attack, the used attack actions, the test data etc. The tests can be carried out both manually and automatically. Accordingly to the implemented test oracle, the program gives a verdict whether the vector succeeded in triggering a vulnerability. Also, the corresponding tested element is shown to the tester so she or he gets a visual expression of the output.

In fact, the tool offers a great deal of configurability with regards to the implemented technology. The tester can interact with the program on a minimum scale that is by setting only the initial configuration like URL address. On the contrary, a test can be carried out completely manually by assigning specific values to selected parts of the website.

In the following the underlying techniques and the internal architecture of PURITY will be briefly described.

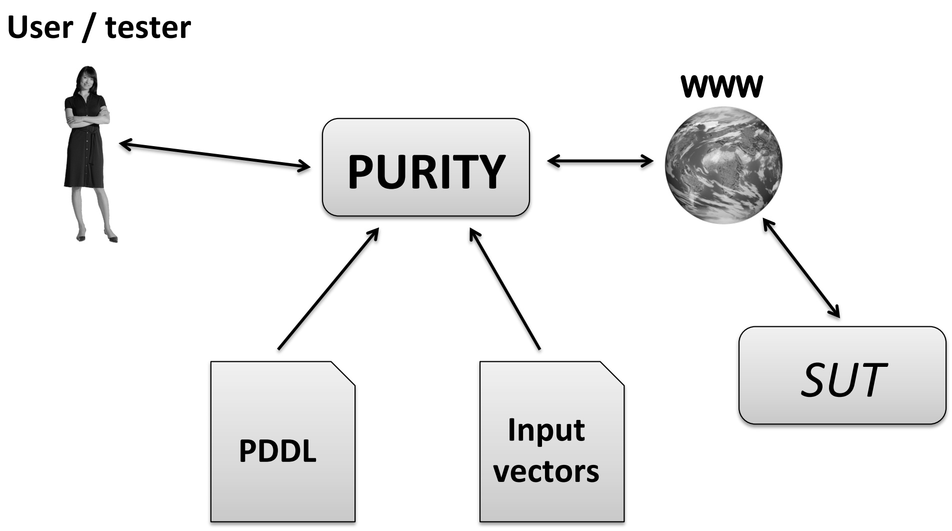


Figure 6.2: PURITY in context with its environment

6.2.1 Key Components

PURITY encompasses a variety of different components. These have been implemented using different Java libraries as well as other external programs. While some of them were already explained in Section 3.2, the most important new ones are the following:

- **Crawler**: The implemented crawler offers the possibility to define the crawl depth and number of pages to fetch. (In the initial version of PURITY the open source Web crawler Crawler4j[1] has been used but in the meantime it got replaced by the author's own implementation.)
- **Test oracles**: They are implemented inside PURITY. Detection mechanisms for both SQLI and XSS are discussed in detail in Section 3.4 as well as in the authors' related works. However, it is important to understand how SQLI works in order to know why an expected value is asked for in PURITY's GUI. The general fact is that after a malicious vector has been submitted, the outcome is always hard to predict. In this case the tester is asked to specify a unique value which will be searched for in the HTTP response after the attack occurs. For example, the tester might already know (e.g. by social engineering) the username of a victim. In that case, this could be specified as an indicator value. On the other hand, the detection of both types of XSS is handled automatically without necessary feedback specification.

Figure 6.3 depicts the internal software architecture of the proposed tool. All interaction between tester and PURITY proceeds over the GUI. This represents the front end from where the entire functionality can be accessed. However, below that layer the implementation is responsible for the data flow between the individual components and the user.

A web application is accessed either over the World Wide Web or locally, wherever it might be deployed. The URL acts as the starting point once the testing process is started. The communication between PURITY and SUT is handled dynamically by HttpClient like in the previous approach.

The Web crawler browses the SUT and identifies hyperlinks in websites that are connected to the initial URL. It takes the submitted URL as a starting

[1] `https://github.com/yasserg/crawler4j`. Accessed: 2015-12-28.

seed and eventually returns all ongoing addresses. It should be mentioned that the tester can restrict the crawl depth and define a maximum number of pages to fetch. During test execution, all incoming data from the crawler is submitted directly to HttpClient.

However, concrete inputs are needed for a test case. PURITY encompasses two initial test sets, one with SQL injections and the other containing XSS vectors. New ones can be obtained externally by attaching them to the tool. During execution, these TXT files are read line by line and sent to HttpClient, which puts them inside HTTP requests as well. If the tester wants to create new input files, he should take care of the data structure for both SQL and JavaScript. Otherwise the data will be sent to the SUT anyway; however no meaningful results would be obtained.

It is very important to note that until now all of the described components from Figure 6.3 work as parts of the Java implementation. This means that concrete test cases are built automatically from the current URL address, web component data and attack vectors by the test case generator.

Plans are generated by the planner, as described in Section 6.1. A plan cannot be used for testing purposes unless there are concrete values that somehow correspond to abstract values from the planning domain. The test case generator reads the abstract actions and searches for their concrete counterpart in the implementation. Once found, it is executed. One advantage of this approach is the fact that for one abstract object from an action a dozen of concrete attack vectors can be applied. For example, the implementation picks one attack vector from the input files and applies it to a variable in the implementation. After the plan is executed on the concrete level, PURITY reads the next vector and repeats the plan execution with the new value.

The implementation calls the planner by submitting the two PDDL files. However, PURITY also generates new files of this kind. Since every plan is constructed according to a specific data configuration, a different configuration would also result in a different plan. Exactly that is what PURITY focuses on: It creates new problem definitions with somehow different initial values. Now Metric-FF delivers a new plan that is parsed by the tool, which carries out the concrete execution. The way PURITY creates new PDDL files will be elaborated further.

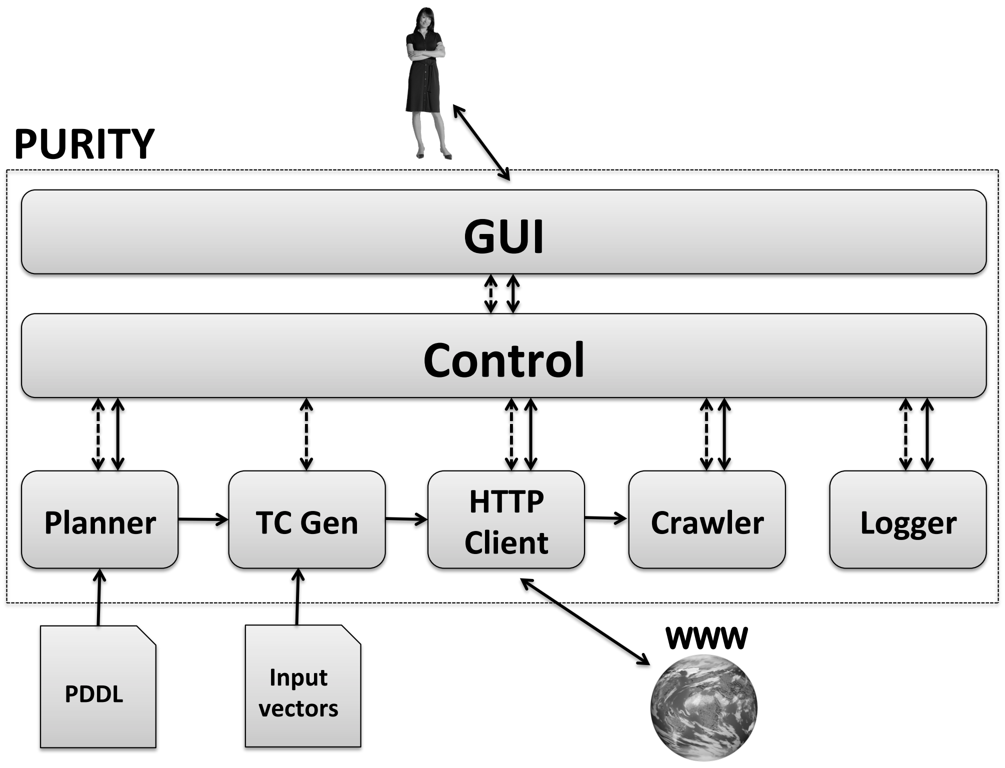


Figure 6.3: PURITY's internal software architecture

Generally speaking, the implementation generates concrete data accordingly to abstract ones. On the other hand, it also creates abstract data that is meant to be processed by the planner. The planner produces in turn new abstract data for the implementation. This is a cyclic process that continues as long as plans are generated and attack vectors are available.

Finally, the logger collects all relevant data produced during the execution. The tester has the choice whether she or he wants to log all events during the execution or just critical messages like exceptions.

6.2.2 Structure of Inputs in PURITY

As mentioned before, one of the primary motivations for this tool was to ease the effort for the tester to effectively test a program. For this case, the amount of interaction is kept as small as possible. For instance, it is completely sufficient just to give the URL address of the SUT and click a button in order to start the testing process. The rest will be handled by the program automatically. This fact increases the usability of the tool while keeping the execution time relatively low.

On the other hand the tester might want to know the functionality behind PURITY and interacts with the system. If this is the case, the tool offers several possibilities to realize that. However, first a description has to be provided about what types of user inputs are used in PURITY:

Type of attack: Can be either SQLI or XSS. According to this choice, the program expects different attack vectors and applies different test oracles. However, if the tool is run completely automatically, the SUT will be tested for both vulnerabilities.

Attack vectors: A text file is attached to the PURITY and read line by line. Every row should contain one vector that resembles either one SQL query or JavaScript code. The tester can attach TXT files with attack vectors to PURITY before the testing process starts. Otherwise the tool will make use of two files that are already included.

Domain.pddl: This file is already attached to the tool and it encompasses predicate and function definitions as well as actions. Once set, the domain file will remain unchanged during automated test execution that is all further plans will be constructed according to the same definition.

Problem.pddl: The problem file is part of the tool and specifies objects and initial values. During execution these values will be replaced with new ones, thus continuously creating new problem files. Every generated plan will be derived from a different problem specification.

Here the symbolic names `domain.pddl` and `problem.pddl` will be used for both specifications regardless of the files' name.

6.2.3 Modes of Use

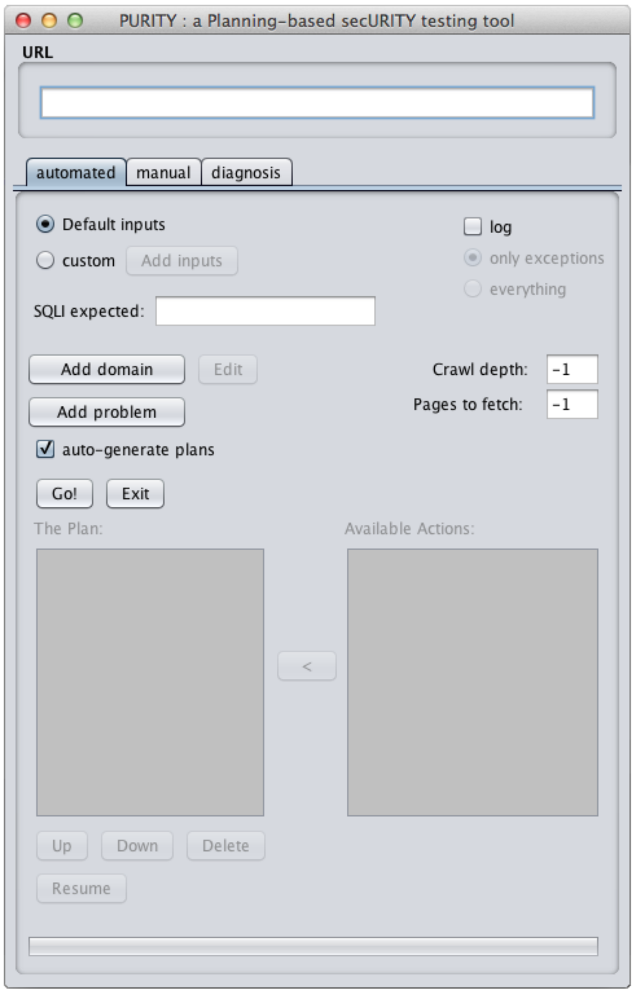
After starting PURITY, the tester has the choice between four different test execution modes. Figure 6.4 depicts the GUI of the tool.

The minimum requirement for every one of them is to specify the URL address of the SUT. The initial values for the crawler are initially set to -1 for both the crawl depth and number of pages to fetch. These values decide about how deep the crawler will go into the application by starting from the initial URL and how many pages will be fetched during that search (-1 stands for unlimited.). In the following a brief description is given of each mode of use PURITY offers.

Completely Automatic

This mode is the most extensive one because it performs the execution in a completely automated manner. It will be picked per default if the checkbox `auto-generate plans` is selected.

If she or he desires, the tester can load own input files into the tool before pushing the start button. Since this mode tests automatically for both SQLI and XSS, the tester can use test sets for both vulnerabilities. The user can edit the initial domain specification and delete actions in a simple editor if desired. In such way fewer actions are taken into consideration by Metric-FF so the plans will get simpler as well. From now on the reduced domain file will remain unchanged during the entire testing process.



[✓ Default inputs] custom

[✓ auto-generate plans]

Crawl depth: -1

Pages to fetch: -1

Figure 6.4: PURITY's GUI

When the execution starts, PURITY submits the initial PDDL files to Metric-FF that in turn generates the first abstract test case. If no plan could be generated, the user will be notified. From now on the procedure follows **PLAN4SEC 2.0** as long as attack vectors are available and plans are generated. An example of this type of execution is demonstrated in Section 6.1.2.

Of all modes, this one covers most of the functionality of PURITY and demonstrates the adaptation of planning in testing at its best.

Partly Automatic

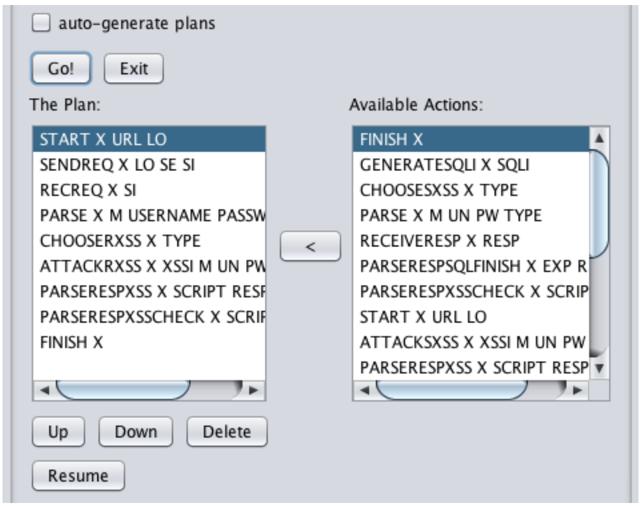
This selection relies on testing as well but to a much lesser extent. In this case the tester generates just one plan, which actions are displayed in a separate window. Additionally, all available actions are parsed from `domain.pddl` and displayed as well. Now the tester can make experiments by deleting and adding actions or changing the order of their appearance.

The new list will be sent to the test case generator and carried out automatically as would be the usual case. However, the difference to the completely automated approach lies in the fact that this time only one plan is executed. To be precise, only one execution is carried out per attack vector. Figure 6.5 depicts the section that contains the generated plan. As can be seen, plan actions can be either removed or added from the menu.

Actually this mode is meant for the tester to manipulate planning related data and to check the corresponding effects.

Completely Manual

With this choice the tester can test a single website by manually writing values for all its user input fields. HttpClient parses the page from the specified URL and displays all HTML elements that could be tested. Now the user can add or remove parameters if she or he wishes. For example, sometimes it is proven to be useful to submit one parameter twice in a request, for example by submitting `username=Ben&username=[malicious script]`. This configuration can be defined manually in the table. In order to realize this, the tester adds a row in the elements table and writes the name of the parameter and its value. Afterwards she or he might initialize the testing process.



The Plan:

```
START X URL LO
SENDREQ X LO SE SI
RECREQ X SI
PARSE X M USERNAME PASSW
CHOOSERXSS X TYPE
ATTACKRXSS X XSSI M UN PW
PARSERESPXSS X SCRIPT RESF
PARSERESPXSSCHECK X SCRIF
FINISH X
```

Available Actions:

```
FINISH X
GENERATESQLI X SQLI
CHOOSESXSS X TYPE
PARSE X M UN PW TYPE
RECEIVERESP X RESP
PARSERESPSQLFINISH X EXP R
PARSERESPXSSCHECK X SCRIP
START X URL LO
ATTACKSXSS X XSSI M UN PW
PARSERESPXSS X SCRIPT RESP
```

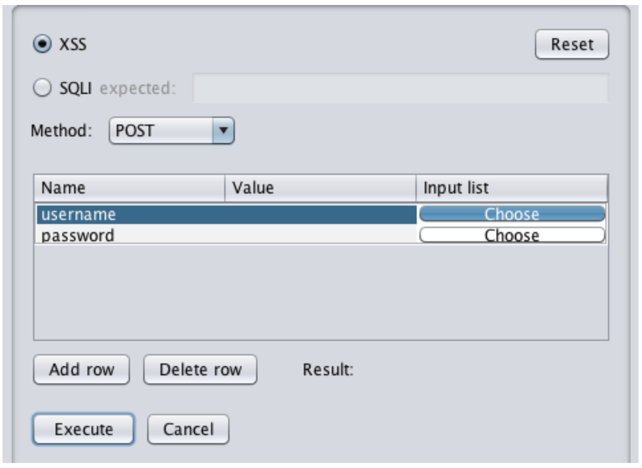
Figure 6.5: Section for partly automated testing

However, there will be no result table displayed since the user has a clear insight what parts of the SUT are tested with a known attack vector. Since no planning and crawler are used and no test files are attached, only one test case will be executed per attack. Figure 6.6 shows the manual testing section. All extracted HTML elements are shown in the table and concrete values are added in the cells from the corresponding column.

In fact, this functionality and the following one define PURITY as a manual testing tool as well.

Partly Manual

If user input fields are encountered during parsing of a website in the completely manual mode, the tester is also offered the possibility to test one specific element against a list of vectors in an automated manner. In order to accomplish this, a button is located in the table beside the field that is wished to be tested. This opens a file chooser where one or more vector files can be selected. After the desired vulnerability is checked as well, the testing process can be started. Now the desired website's input element will be tested automatically against all vectors from the input file(s).



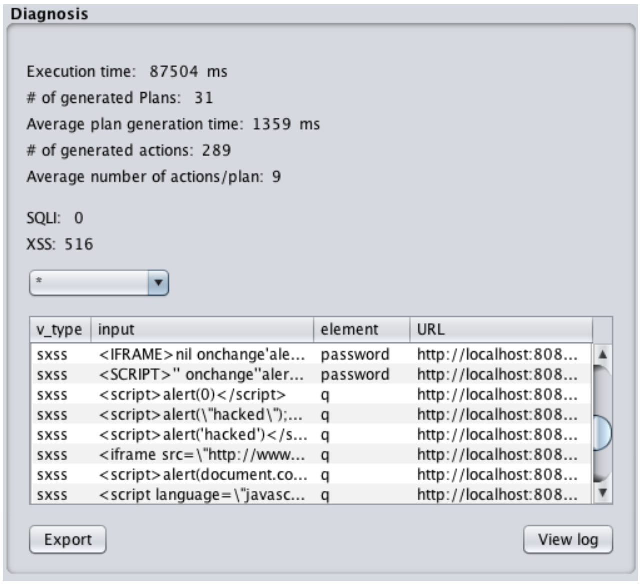
Name	Value	Input list
username		Choose
password		Choose

Figure 6.6: Menu for manual testing

6.2.4 Case Study

The functionality of PURITY is demonstrated by testing one of the SUTs from Section 3.5 by choosing the automated mode. The tester is asked to specify an URL, the input data sets, PDDL files and the specifications for the crawler. In order to select the completely automated mode, the tester has to select the corresponding checkbox. Otherwise the program chooses the partly automated mode. The tester can edit the domain specification or add several test input files.

Once the specification is selected, testing can be started. In this example the URL represents the local address where the SUT has been deployed. First, the planner is called with the selected PDDL's and the corresponding output is read. As explained above, the planning system will return a plan to the implementation which will start immediately the concrete test execution. First the initial values of the current `problem.pddl` are parsed in search for critical objects, like `type` or `method`. The configuration is saved and the execution continues by fetching the first vector from the TXT file as explained in Section 6.1.1. Since it may be hard to predict what hyperlinks are connected to the initial URL, it can be expected to encounter sites with no input fields at all. In this case the current test run is terminated and the next one starts.



Execution time: 87504 ms
# of generated Plans: 31
Average plan generation time: 1359 ms
# of generated actions: 289
Average number of actions/plan: 9

SQLI: 0
XSS: 516

v_type	input	element	URL
sxss	<IFRAME>nil onchange'ale...	password	http://localhost:808...
sxss	<SCRIPT>" onchange"aler...	password	http://localhost:808...
sxss	<script>alert(0)</script>	q	http://localhost:808...
sxss	<script>alert(\"hacked\");...	q	http://localhost:808...
sxss	<script>alert('hacked')</s...	q	http://localhost:808...
sxss	<iframe src=\"http://www...	q	http://localhost:808...
sxss	<script>alert(document.co...	q	http://localhost:808...
sxss	<script language=\"javasc...	q	http://localhost:808...

Figure 6.7: Output table

After all tests have been executed for the initial URL, the crawler fetches the next ongoing address. For example, in the above demonstration the new address would be `http://localhost:8080/bodgeit/home.jsp`. The concrete test execution will now continue by parsing a new problem definition after which the vector files will be read from the beginning and so on.

After execution, all positive test verdicts are displayed in the table. Figure 6.7 depicts such a table where every row contains the type of triggered vulnerability, the responsive attack vector, name of the vulnerable HTML element and the corresponding URL address. The table can be exported as an Excel sheet. Besides that, the diagnosis window also shows some statistical data like the generation and execution time for all tests, the number of generated plans and actions etc. Also, the total amount of successful tests for both SQLI and XSS is shown. The reasons behind the high number of XSS vectors lies in the fact that HTML input elements were very vulnerable for the tested SUT. On the other side, not a single SQLI leak was detected. The reason for this is either that the expected value hasn't been the right choice or that software intern filtering mechanisms were efficient enough to escape the malicious code in the first place. Another reason for a failure is usually the fact that a website does not use a database or no user input fields are available.

6.3 Summary

This chapter introduces a planning-based security testing approach for web application vulnerabilities. The algorithm PLAN4SEC is described in detail, which is meant for security testing of web applications. Besides test case generation the algorithm also allows for automated test execution. The underlying method from predefined specifications is discussed and its realization explained. Eventual changes can be straightforwardly integrated into the algorithm. The first initial results indicate that the approach can be used in testing of web applications.

Additionally, the research prototype PURITY is presented that is meant for either automated or manual testing (or something in-between). For this tool some several new features were added so a unique planning based testing framework was made.

However, there is space for improvement for this approach when considering the concretization of test cases. Adding more actions of finer granularities in the initial specification might increase the number of tests and make the approach more effectively in practice. In particular, the influence on the certain improvements to the vulnerability detection would be interesting to determine.

According to the explained approach, some of the research questions will be answered.

RQ1: How can all necessary attack information be formalized?

In addition to the attack representation in Chapter 5, the second type of attack depiction is realized in the following way.

All attack information is implemented as part of the planning problem. This is done by defining specific individual actions that encompass pre- and postconditions, which are defined by predicates with corresponding parameters. The necessary information is defined inside the domain and problem specifications in PDDL. These definitions are used by action specifications. According to the implementation, a planner generates the plan that acts as a test case for the later testing process. Actually, the resulting plan represents one attacking attempt. During execution, the individual values from the specification might change, which will result in a different plan, that is, a different attack. Since the research is based around SQLI and XSS, the generated plan will represent actions for these types of vulnerabilities.

RQ2: What testing methodologies are developed for vulnerability detection in web applications?

A novel approach to security testing based on planning is introduced. In particular, the test case generation problem is formalized as a planning problem and a planner is used for generating plans that represent abstract test cases. However, the limitations of the approach are not reached. The number of plans to be generated can be much higher when using either more initial values or planners that deliver different plans and not only one. In contrast to other testing methods the mapping to planning increases reuse of knowledge used for test generation and also makes adaptations to specific languages much easier. In particular, new actions can be added to the domain definition.

The planning-based security testing tool PURITY is used to test web applications for both SQLI and XSS. The tester can use it on black- or white-box basis. PURITY encompasses the novel testing approach, which is based on the test case generation technique that is based on automated plan generation. On the other hand, the tester is also offered the possibility to execute the tool in a manual manner. She or he can set test parameters and see how the SUT reacts on different inputs.

RQ4: How does planning-based testing perform for vulnerability detection?

The empirical evaluation using the SUTs from Section 3.5 indicates that the approach can be used in practice. The planning time including plan generation and execution is high but acceptable for the application domain.

Also, the capabilities of detecting vulnerabilities are in line with other approaches to automated testing of web applications.

Although a research prototype, PURITY succeeded in testing of several web applications. It offers a high degree of compatibility, which is demonstrated in the tool description above. However, it remains possible to improve it further by including additional features, for example adding more actions into the domain specification and increasing the configurability by allowing more manual intervention. PURITY can be experimented with so even new test scenarios can be adapted. Although it does not represent a demand, the tester can add own attack vectors or manipulate existing ones. As was demonstrated before, the execution time is relatively low when applying thhis technique.

7 Conclusion

From the global point of view this thesis addressed several open problems, including, but not limited to:

- Testing of web applications
- Conceptualizing attacks
- Test case generation and execution

In order to contribute to these topics, several methods were applied and combined. First, some of the state-of-the-art topics from model-based testing have been presented and elaborated on an example. Then, the notion of pattern modeling has been applied to attacks on web applications, that is, testing for vulnerabilities. Information about the attack procedure is gathered manually by observing the interaction between the client and a web application. Then, the obtained information is implemented in form of a UML state machine. These models are commonly used in model-based testing so the tester might be familiar with their application. The resulting representation acts as an attack pattern, that is, a unique sequence of actions with specific pre- and postconditions. Since SQLI and XSS are among the most common vulnerabilities, these two attack scenarios have been chosen for depiction. Both can be chosen either separately by the tester during execution or put together into one single graphical representation. Since Java methods are implemented and run in the background during execution, additional functions can be programmed and adapted into the testing framework easily. In addition to the attack, corresponding test oracles were defined with respect to the vulnerabilities. The attack pattern is executed in an automated fashion (or semi-automated; according to the choice of the tester) against a SUT, thus generating a verdict according to the oracles. The goal of test automation is to release the tester from time-consuming manual work and to fasten the entire process. Another advantage of this approach is that it should not depend upon a particular SUT but can be used on a wide scale. The tester is asked only for a minimum of interaction with the testing framework.

Then, an overview is given about the functionality of combinatorial testing. This is a testing technique that combines values from parameters according a specified input model. One of its main advantages is the fact that a good coverage is achieved by generating a lesser number of test cases. In the case of vulnerability detection this means that they might be triggered by executing fewer test cases. Here combinatorial testing was applied for the generation of concrete attack vectors for XSS. First the attack vector information is structured in form of an attack grammar. Afterwards, the grammar is implemented as an input model for the test case generation tool ACTS, which subsequently generates corresponding attack vectors. Then, these were executed against the SUTs in an automated manner by attaching them to an attack pattern. The new approach, which combines the attack pattern-based test execution with combinatorial testing-based test case generation, is called Attack Pattern-based Combinatorial Testing. However, in order to improve the quality of attack vectors, constraints were put upon the grammar in the input model so that new test sets were generated. Constraints are used in combinatorial testing in order to restrict and guide the parameter interaction so that more meaningful attack values are generated in the aftermath. This mechanism contributes to test suite reduction and increases the vulnerability detection likelihood as well. Since combinatorial testing uses different strategies for test suite generation, the output will differ depending of what algorithm was used in the first place. For this case, two of the most used algorithms were used, namely IPOG and IPOG-F, for test case generation for XSS. After executing the two generated sets of attack vectors, it has been concluded that the obtained results from the test suites from IPOG-F were overall slightly better. As mentioned before, this was the first time that such a comparison between the algorithms was drawn for web security testing.

Finally, Attack Pattern-based Combinatorial Testing was compared to a state-of-the-art manual testing tool. While slightly better results were obtained with Burp Suite, the automated approach, naturally, was faster during execution. For this sake, some additional effort was required for Burp in order to execute tests in an automated manner. Further, a comparison between the proposed approach and fuzzing indicated that, in the author's opinion, the quality and diversity of the attack vectors is higher when being produced by CT, especially when taking constraints into consideration. Additionally, since a lesser number of test cases still achieves good coverage, less time is needed for vulnerability detection than by applying fuzzing.

The summary of several evaluations confirmed the author's assumption that Attack Pattern-based Combinatorial Testing can be taken into consideration as an alternative for the current established testing doctrines and their corresponding implementations.

Furthermore, a planning-based approach was introduced for testing web applications as well. Again, all necessary attack information is implemented. In this case however it is defined in PDDL, a language for domain and problem specifications in planning. Since a security breach can be represented as a sequence of actions, where one of them causes another, this can be represented as a plan. Every action consists of specific parameters, pre- and postconditions that are constructed with predicates, objects and other elements. Then, the planner constructs a plan for every type of attack where every plan represents an abstract test case. This abstract test case is combined with concrete test cases in order to be executed against a SUT. The algorithm PLAN4SEC was introduced, which is meant for plan generation and execution and explains the whole planning-based approach. The evaluation confirmed the applicability of the idea for vulnerability testing of web applications.

In addition, the tool PURITY is introduced, which automatizes the entire planning-based testing approach and offers the tester a great amount of configurability. Although it still represents a prototype, the tool can be used for automated, semi-automated and manual testing. The tester is given the possibility to configure the testing process, for example by adding other attack vectors or by constructing or modifying a plan.

When comparing both approaches, several conclusions can be drawn:

- **Specification effort**: Since both approaches demand some initial manual work, the question remains which of the methods would be easier for a tester to adapt. The graphical representation is clearly easier to define, since elements can be added and deleted with less effort from the state machine. On the contrary, planning specifications require some knowledge from PDDL.

- **Execution time**: Once all attack information is implemented, attack vectors are executed. By evaluating both approaches, it is noted that testing based on planning clearly outperforms its graphical counterpart with regard to time duration. However, the planning based approach has the drawback that some generated plans cannot lead to the goal state, that is, security breach but these will be executed anyways, thus consuming some execution time in vain. Although it should be mentioned that such abstract plans are traversed very quick because of the absence of values on the concrete level so no preconditions will be satisfied.
- **Extensibility**: Both approaches can be extended by the tester. However, the issues are the same as with the specification effort. PDDL requires more coding effort while state machine manipulation demands less time and offers visual impression about the order of execution as well. On the other hand, in PURITY the tester can manipulate the plan manually, thereby easily constructing new test cases.
- **Configurability**: Both approaches offer the tester the possibility to intervene with the framework or the execution. In the attack pattern, user input or manipulation of values can be done during execution by stopping the process, eventually leading testing into different directions. However, PURITY offers the greatest amount of configurability since the tester can use several modes and also manipulate the plans. Since PURITY is a manual testing tool as well, the tester gets an immediate feedback about the vulnerable elements and the causing attack vector.

However, there is still room for improvement for both new approaches. Since the attack model comprises a certain number of states, this can be extended by adding new ones. In such way, other attack types may be depicted or added to the existing testing procedure, thereby executing several attacks one after another. More preconditions could be incorporated into the model so that more complex applications might be tested by extracting more information etc. Regarding the application of combinatorial testing, a refinement of the attack grammar might lead to better attack vectors as well as the definition of additional constraints upon the input model. Until now, attack vectors were generated that were meant for testing a SUT. However, malicious inputs, that is, inputs that exploit the system could be generated. Since there is no actual limit for the construction of XSS inputs, the input model can encompass a much greater number of parameters and values. On the other hand, imposing constraints on the input model decreases the number but increases the quality of generated inputs. As demonstrated in the evaluations, this means that a lesser number of tests is more likely to trigger a vulnerability, which directly influences the necessary testing costs. Since ACTS relies on different test generation algorithms, a combination of several combinatorial strategies would also deliver a different kind of test sets. Implementing a new algorithm might generate different attack vectors and reduce the number of generated tests, thereby increasing their quality. However, one of the still unhandled topics is the encounter of false positives. Since the behavior of a SUT is hard to predict, a false verdict could be triggered by the interaction of several background elements. In the evaluation in some cases a vulnerability was triggered for every of the manual tests, which leads to the conclusion that it might not be the specific attack vector that causes the vulnerability but something else. For this case, an extension to the defined test oracles may be needed. The PDDL definitions require some manual work in order to be specified. By extending the individual specifications longer plans could be generated as well. For example, actions and initial parameters can be added into the respective files. Actually, plans can be generated and executed continuously as long as a vulnerability is not triggered.

Finally, it is the hope of the author that the two presented approaches offer a new perspective on model-based security testing. Eventually, both ideas can be developed further, either by extending the attack patterns or re-defining planning specifications.

Bibliography

[1] Anchor cms. http://anchorcms.com/. Accessed: 2015-12-01.

[2] Automated combinatorial testing for software (acts). http://www.nist.gov/itl/csd/scm/acts.cfm. Accessed: 2014-01-28.

[3] Bitweaver. http://www.bitweaver.org/. Accessed: 2015-12-01.

[4] Bodgeit. https://code.google.com/p/bodgeit/. Accessed: 2015-12-01.

[5] Burp suite. http://portswigger.net/burp/. Accessed: 2014-01-28.

[6] Damn vulnerable web application (dvwa). http://www.dvwa.co.uk/. Accessed: 2015-12-01.

[7] Fast-forward. http://fai.cs.uni-saarland.de/hoffmann/ff.html. Accessed: 2015-12-01.

[8] Gruyere. http://google-gruyere.appspot.com/. Accessed: 2015-12-01.

[9] Httpclient. http://hc.apache.org/httpcomponents-client-ga/. Accessed: 2015-12-01.

[10] Java ff. http://www.inf.kcl.ac.uk/staff/andrew/JavaFF/. Accessed: 2015-12-07.

[11] jsoup: Java html parser. http://jsoup.org/. Accessed: 2015-12-01.

[12] Metric-ff. http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html. Accessed: 2015-02-10.

[13] Nowasp (mutillidae). http://sourceforge.net/projects/mutillidae/. Accessed: 2015-12-01.

[14] Omg unified modeling language infrastructure version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Infrastructure. Accessed: 2015-12-01.

[15] Omg unified modeling language superstructure version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure. Accessed: 2015-12-01.

[16] Owasp top ten project. https://www.owasp.org/index.php/OWASP_Top_10. Accessed: 2015-12-01.

[17] Owasp webgoat project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project. Accessed: 2015-12-01.

[18] Phrack magazine, volume 8, issue 54. http://phrack.org/issues/54/8.html#article. Accessed: 2015-12-01.

[19] Spacios project no. 257876, fp7-ict-2009-5. http://www.spacios.eu. Accessed: 2015-12-01.

[20] sqlmap. http://sqlmap.org/. Accessed: 2014-01-28.

[21] Webscarab. https://www.owasp.org/index.php/Webscarab. Accessed: 2015-12-01.

[22] Wordpress. http://wordpress.org/. Accessed: 2015-12-01.

[23] Yakindu statechart tools. http://statecharts.org/. Accessed: 2015-12-01.

[24] A. Armando, L. Compagna, and P. Ganty. Sat-based model-checking of security protocols using planning graph analysis. In *Proceedings of the 12th International Symposium of Formal Methods Europe (FME), LNCS 2805. Springer-Verlag*, pages 875–893, 2003.

[25] A. Beer, S. Mohacsi, and C. Stary. IDATG: An Open Tool for Automated Testing of Interactive Software. In *Proceedings of the COMPSAC'98 - 22nd International Computer Software and Applications Conference*, 1998.

[26] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pages 427–430, March 2011.

[27] A. Bernauer, J. Bozic, D. E. Simos, S. Winkler, and F. Wotawa. Retaining consistency for knowledge-based security testing. In *Proceedings of the 27th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE'14)*, pages 88–97, 2014.

[28] A. Blome, M. Ochoa, K. Li, M. Peroli, and M. T. Dashti. Vera: A flexible model-based vulnerability testing tool. In *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST'13)*, 2013.

[29] E. Boerger, A. Cavarra, and E. Riccobene. Modeling the dynamics of uml state machines. In *International Workshop on Abstract State Machines (ASM'2000)*, pages 223–241, 2000.

[30] M. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of acts: A case study. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST'12)*, pages 591–600, 2012.

[31] M.N. Borazjany, Linbin Yu, Yu Lei, R. Kacker, and R. Kuhn. Combinatorial testing of acts: A case study. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 591–600, 2012.

[32] J. Bozic, B. Garn, I. Kapsalis, D. E. Simos, S. Winkler, and F. Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'15)*, 2015.

[33] J. Bozic, B. Garn, D. E. Simos, and F. Wotawa. Evaluation of the ipo-family algorithms for test case generation in web security testing. In *Proceedings of the Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15)*, pages 1–5, 2015.

[34] J. Bozic, D. E. Simos, and F. Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test (AST'14)*, 2014.

[35] J. Bozic and F. Wotawa. Model-based testing - from safety to security. In *Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12)*, pages 9–16, October 2012.

[36] J. Bozic and F. Wotawa. Xss pattern for attack modeling in testing. In *Proceedings of the 8th International Workshop on Automation of Software Test (AST'13)*, 2013.

[37] J. Bozic and F. Wotawa. Plan it! automated security testing based on planning. In *Proceedings of the 26th IFIP WG 6.1 International Conference (ICTSS'14)*, pages 48–62, 2014.

[38] J. Bozic and F. Wotawa. Security testing based on attack patterns. In *Proceedings of the 5th International Workshop on Security Testing (SECTEST'14)*, 2014.

[39] J. Bozic and F. Wotawa. Purity: a planning-based security testing tool. In *Workshop on Trustworthy Computing*, 2015.

[40] M. Brcic and D. Kalpic. Combinatorial testing in software projects. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 1508–1513, 2012.

[41] M. Busch, R. Chaparadza, Z. R. Dai, A. Hoffmann, L. Lacmene, T. Ngwangwen, G.C. Ndem, H. Ogawa, D. Serbanescu, I. Schieferdecker, and J. Zander-Nowicka. Model transformers for test generation from system models. In *Conquest 2006. Hanser Verlag, Berlin*, 2006.

[42] D. Byers and N. Shahmehri. Unified modeling of attacks, vulnerabilities and security activities. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems (SESS'10), IEEE*, 2010.

[43] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li. Uml activity diagram-based automatic test case generation for java programs. In *The Computer Journal*, 2007.

[44] J. Clarke, R. M. Alvarez, D. Hartley, J. Hemler, A. Kornbrust, H. Meer, G. OLeary-Steele, A. Revelli, M. Slaviero, and D. Stuttard. *SQL Injection Attacks and Defense*. Syngress, Syngress Publishing, Inc. Elsevier, Inc., 30 Corporate Drive Burlington, MA 01803, 2009.

[45] J. Clarke, K. Fowler, E. Oftedal, R. M. Alvarez, D. Hartley, A. Kornbrust, G. O'Leary-Steele, A. Revelli, S. Siddharth, and M. Slaviero. *SQL Injection Attacks and Defense, Second Edition*. Syngress, 2012.

[46] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, July 1997.

[47] Charles J. Colbourn. Covering arrays. In Charles J. Colbourn and Jeffrey H. Dinitz, editors, *Handbook of Combinatorial Designs*, Discrete Mathematics and Its Applications, pages 361–365. CRC Press, Boca Raton, Fla., 2nd edition, 2006.

[48] Atlee M. Cunningham Jr., Jon Hagar, and Ryan J. Holman. A system analysis study comparing reverse engineered combinatorial testing to expert judgment. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 630–635, Washington, DC, USA, 2012. IEEE Computer Society.

[49] ThanhBinh Dao and Etsuya Shibayama. Coverage criteria for automatic security testing of web applications. In Somesh Jha and Anish Mathuria, editors, *Information Systems Security*, volume 6503 of *Lecture Notes in Computer Science*, pages 111–124. Springer Berlin Heidelberg, 2010.

[50] J. D. DeMott, R. J. Enbody, and W. F. Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In *Black Hat USA 2007, Las Vegas*, 2007.

[51] I. Dominguez Mendoza, D.R. Kuhn, R.N. Kacker, and Yu Lei. CCM: A tool for measuring combinatorial coverage of system state space. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 291–291, 2013.

[52] F. Duchene, R. Groz, S. Rawat, and J. Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 815–817, April 2012.

[53] Fabien Duchene, Roland Groz, Sanjay Rawat, and Jean-Luc Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 815–817, Washington, DC, USA, 2012. IEEE Computer Society.

[54] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *CODASPY*, pages 37–48. ACM, 2014.

[55] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[56] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[57] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Proceedings of the 2007 IEEE Symposium Pacific Rim Dependable Computing (PRDC 2007)*, 2007.

[58] M. Forbes, J. Lawrence, Yu Lei, Raghu Kacker, and D. Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113:287–297, 2008.

[59] P. Froehlich and J. Link. Automated test case generation from dynamic models. In *Proceedings of the 14th European Conference on Object-Orinted Programming (ECOOP'00)*, 2000.

[60] S. J. Galler, C. Zehentner, and F. Wotawa. Aiana: An ai planning system for test data generation. In *1st Workshop on Testing Object-Oriented Software Systems*, pages 30–37, 2010.

[61] Bernhard Garn, Ioannis Kapsalis, Dimitris E. Simos, and Severin Winkler. On the applicability of combinatorial testing to web application security testing: A case study. In *Proceedings of the 2nd International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation (JAMAICA'14)*. ACM, 2014.

[62] L.S.G. Ghandehari, M.N. Bourazjany, Yu Lei, R.N. Kacker, and D.R. Kuhn. Applying combinatorial testing to the siemens suite. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 362–371, 2013.

[63] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[64] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[65] A. Heinecke, T. Brueckmann, T. Griebe, and V. Gruhn. Generating test plans for acceptance tests from uml activity diagrams. In *Proceedings of the 17th International Conference on Engineering Computer-Based Systems, IEEE*, 2010.

[66] J. Hoffmann. Extending ff to numerical state variables. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 571–575, 2002.

[67] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. In *Journal of Artificial Intelligence Research 14*, pages 253–302, 2001.

[68] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004. ISBN: 0-201-78695-8.

[69] A. E. Howe, A. von Mayrhauser, and R. T. Mraz. Test case generation as an ai planning problem. In *Automated Software Engineering, 4*, pages 77–106, 1997.

[70] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software*

Engineering, ICSE ’09, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[71] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 30th International Conference on Software Engineering (ICSE’09)*, 2009.

[72] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. In *IEEE Proceedings-Software, 146(4)*, pages 187–192, 1999.

[73] D.R. Kuhn, R.N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, 2013.

[74] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, 2008.

[75] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG-IPOG-D: Efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18(3):125–148, September 2008.

[76] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE ’98, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society.

[77] A. Leitner and R. Bloem. Automatic testing through planning. Technical report, Technische Universität Graz, Austria, 2005.

[78] Andreas Leitner. Strategies to automatically test eiffel programs. Master’s thesis, Technische Universität Graz, Austria, 2004.

[79] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. Security test generation using threat trees. In *Proceedings of the ICSE Workshop on Automation of Software Test (AST’09)*, pages 62–69, 2009.

[80] J.R. Maximoff, M.D. Trela, D.R. Kuhn, and R. Kacker. A method for analyzing system state-space coverage within a t-wise testing framework. In *Systems Conference, 2010 4th Annual IEEE*, pages 598–603, 2010.

[81] Manish Mehta and Roji Philip. Applications of combinatorial testing methods for breakthrough results in software testing. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '13, pages 348–351, Washington, DC, USA, 2013. IEEE Computer Society.

[82] A. P. Moore, R. J. Ellison, and R.C. Linger. Attack Modeling for Information Security and Survivability. In *Technical Note CMU/SEI-2001-TN-001*, March 2001.

[83] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.

[84] N. J. Nilsson. Teleo-reactive programs for agent control. In *Journal of Artificial Intelligence Research, 1*, pages 139–158, 1994.

[85] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 26–36, 2013.

[86] C. Phillips and L. Swiler. A graph-based system for network vulnerability analysis. In *ACM New Security Paradigms Workshop*, pages 71–79, 1998.

[87] Elisa Puoskari, Tanja E. J. Vos, Nelly Condori-Fernandez, and Peter M. Kruse. Evaluating applicability of combinatorial testing in an industrial environment: A case study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, JAMAICA 2013, pages 7–12, New York, NY, USA, 2013. ACM.

[88] S. Rawat and L. Mounier. An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light. In *Proceedings of the 2010 European Conference on Computer Network Defense (EC2ND)*, pages 37–45, October 2010.

[89] S. Rawat and L. Mounier. Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 531–533, March 2011.

[90] D. Richard Kuhn and Vadim Okum. Pseudo-exhaustive testing for software. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, SEW '06, pages 153–158, Washington, DC, USA, 2006. IEEE Computer Society.

[91] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an oo model with an ai planning system. In *Proceedings of The 10th International Symposium on Software Reliability Engineering. IEEE Computer Society, Washington, DC, USA*, pages 250–259, 1999.

[92] I. Schieferdecker. Model-based fuzz testing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, page 814, April 2012.

[93] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. In *Proceedings of the Model-Based Testing Workshop at ETAPS 2012. EPTCS*, pages 1–12, 2012.

[94] M. Schnelte and B. Gueldali. Test case generation for visual contracts using ai planning. In *INFORMATIK 2010, Beitraege der 40. Jahrestagung der Gesellschaft fuer Informatik e.V. (GI)*, pages 369–374, 2010.

[95] I. Segall, R. Tzoref-Brill, and A. Zlotnick. Common patterns in combinatorial models. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 624–629, 2012.

[96] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[97] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley Publishing, Inc., 2011. Second Edition.

[98] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[99] A. Takanen. Fuzzing: the past, the present and the future. In *SSTIC'09*, 2009.

[100] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, USA, 2008.

[101] W. Tian, J.-F. Yang, J. Xu, and G.-N. Si. Attack model based penetration test for sql injection vulnerability. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, 2012.

[102] Omer Tripp, Omri Weisman, and Lotem Guy. Finding your way in the testing jungle: A learning approach to web security testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 347–357, New York, NY, USA, 2013. ACM.

[103] F. van der Loo. Comparison of Penetration Testing Tools for Web Applications. Master's thesis, University of Radboud, Netherlands, 2011.

[104] A. Vernotte, F. Dadeau, F. Lebeau, B. Legeard, F. Peureux, and F. Piat. Efficient detection of multi-step cross-site scripting vulnerabilities. In *Proceedings of the 10th International Conference on Information System Security (ICISS'14)*, pages 358–377, 2014.

[105] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu. Automated security test generation with formal threat models. In *IEEE Transactions on Dependable and Secure Computing 9 (4)*, pages 526–540, 2012.

[106] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[107] Linbin Yu, Yu Lei, R.N. Kacker, and D.R. Kuhn. Acts: A combinatorial test generation tool. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 370–375, 2013.

[108] Jian Zhang, Zhiqiang Zhang, and Feifei Ma. The IPO family. In *Automatic Generation of Combinatorial Test Data*, SpringerBriefs in Computer Science, pages 41–49. Springer Berlin Heidelberg, 2014.

[109] Zhiqiang Zhang, Xiaojian Liu, and Jian Zhang. Combinatorial testing on id3v2 tags of mp3 files. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 587–590, Washington, DC, USA, 2012. IEEE Computer Society.