Web Application Security Testing by Logic Programming

Dissertation

by

Philipp Zech

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

supervisor:
Prof. Dr. Ruth Breu

Institute of Computer Science
University of Innsbruck
August 2014
“Quality is never an accident. It is always the result of intelligent effort.”

John Ruskin
Abstract

This dissertation introduces a new method for non-functional security testing by logic programming and its model-based tool implementation for non-functional security testing of web applications. Our work is motivated by severe drawbacks in current non-functional security testing, viz. neglecting negative requirements and corresponding risks as well as its prevalent penetration testing-like, i.e., unstructured and artisanal, style. This is by virtue of a general lack of security relevant knowledge and structured as well as reproducible methods when doing security testing. Our work addresses these deficiencies by introducing the concept of a Vulnerability Knowledge Base, an expert system which both, stores necessary security vulnerability knowledge and realizes a security risk analysis. Our method, starting from a model of the system at an interface-level, then (i) performs a security risk analysis on this very model for establishing a set of application security risks, (ii) generates a risk profile of the application, and (iii) uses this risk model as a test model for non-functional security testing. We have evaluated our method and its model-based tool implementation by two case studies, which show its effectiveness in detecting vulnerabilities in web applications and thus, also its value in making web applications more secure.
Acknowledgments

The years I spent working on my PhD degree at the University of Innsbruck have undoubtedly been some of the most joyful and rewarding in my life. They provided me with everything I need to thrive: challenging research problems, excellent company, and a supportive environment. I am deeply grateful to the people who shared this experience with me and to those who made it possible.

Special thanks go to my advisor, Prof. Dr. Ruth Breu. Ruth has provided me with the invaluable support and the independence I needed. She has helped me develop as a better researcher, critical thinker, writer, and presenter. Her insistence on excellence has influenced me profoundly and helped me aspire to become better at everything I do.

I am very grateful to Dr. Michael Felderer. He has been a tireless supporter of me before, during, and after my stay at University of Innsbruck. Michael’s integrity has been an inspiration to me and I see him as an example to follow on how to approach research problems and collaboration.

I have had great colleagues at University of Innsbruck: Basel Katt, Hannes Mösl, Christian Haisjackl, Florian Häser, Berthold Argreiter, Matthias Gander, Thomas Schrettl, Matthias Farwick, Thomas Trojer. Their supportive companionship has been very important to me, and I learned something unique and valuable from each of them.

Philipp Kalb, my office mate, deserves a special thank-you. We have collaborated on more than I can remember, and working with him has always been rewarding and fun. He has been an excellent friend and I admire his efficiency, sense of humor, and intelligence.

My final thanks go to my family. I am indebted to my parents for the fantastic love, support, and opportunities they have given me. My lovely girlfriend has provided me with her boundless love and support. My stay at University of Innsbruck has been a great adventure and there is no one in the world I would rather have shared it with.
# Contents

1 Introduction 6  
1.1 Problem Statement and Research Challenges 10  
1.2 Problem Solution and Contributions 11  
1.3 Structure of this Thesis 15  

2 Foundations and State-of-the-Art 18  
2.1 Foundations 18  
2.1.1 Logic Programming 18  
2.1.1.1 SLD-Resolution 20  
2.1.1.2 Definite-clause Grammars 22  
2.1.2 Knowledge Engineering 26  
2.1.3 Web Applications 27  
2.1.4 Security Vulnerabilities and Attacks 29  
2.1.4.1 SQLI and XSS Attacks 31  
2.1.5 Security Testing 37  
2.1.5.1 Functional Security Testing 41  
2.1.5.2 Non-functional Security Testing 42  
2.1.5.3 Model-based Security Testing 44  
2.2 State-of-the-Art 53  
2.2.1 Logic Programming in Testing 54  
2.2.2 Knowledge Engineering in Security Engineering 55  
2.2.3 Security Testing 56  
2.2.3.1 Penetration Testing 56  
2.2.3.2 Model-based Security Testing 58  
2.2.3.3 Web Application Security Testing 58  

3 Method 62  
3.1 Model-based, Non-functional Security Testing of Web Applications 63  
3.1.1 Security Problem 66  
3.1.2 Vulnerability Knowledge Base 68  
3.1.2.1 Extensional Database 69  
3.1.2.2 Intensional Database 71  
3.1.2.3 Test Data 77  
3.1.2.4 Establishment and Maintainability of the VKB 80  

4
<table>
<thead>
<tr>
<th>3.1.3 Risk Profile</th>
<th>84</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.4 Test Generation</td>
<td>86</td>
</tr>
<tr>
<td>3.1.5 Test Execution</td>
<td>87</td>
</tr>
<tr>
<td>3.1.6 Test Evaluation</td>
<td>89</td>
</tr>
<tr>
<td>3.2 High-level Modeling Facilities</td>
<td>96</td>
</tr>
<tr>
<td>3.2.1 Web Spider</td>
<td>97</td>
</tr>
<tr>
<td>3.2.2 System Description Language</td>
<td>98</td>
</tr>
<tr>
<td>3.2.3 Risk Description Language</td>
<td>99</td>
</tr>
<tr>
<td>3.2.4 Model Translators</td>
<td>101</td>
</tr>
<tr>
<td>3.3 Formal Foundations</td>
<td>103</td>
</tr>
<tr>
<td>4 Experimental Evaluation</td>
<td>109</td>
</tr>
<tr>
<td>4.1 Case Studies</td>
<td>109</td>
</tr>
<tr>
<td>4.1.1 Damn Vulnerable Web Application</td>
<td>109</td>
</tr>
<tr>
<td>4.1.2 Lab Setup</td>
<td>111</td>
</tr>
<tr>
<td>4.1.3 Case Study Execution</td>
<td>111</td>
</tr>
<tr>
<td>4.2 Findings</td>
<td>117</td>
</tr>
<tr>
<td>4.2.1 Testing with Low Security Level</td>
<td>117</td>
</tr>
<tr>
<td>4.2.2 Testing with Medium Security Level</td>
<td>121</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>125</td>
</tr>
<tr>
<td>5.1 Interpretation of Findings</td>
<td>126</td>
</tr>
<tr>
<td>5.2 Positioning w.r.t. State-of-the-Art</td>
<td>128</td>
</tr>
<tr>
<td>5.3 Implications</td>
<td>132</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>134</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>134</td>
</tr>
<tr>
<td>6.2 Key Findings and Implications</td>
<td>135</td>
</tr>
<tr>
<td>6.3 Future Work</td>
<td>136</td>
</tr>
<tr>
<td>Bibliography</td>
<td>139</td>
</tr>
<tr>
<td>A Extensional Database</td>
<td>153</td>
</tr>
<tr>
<td>B Intensional Database</td>
<td>155</td>
</tr>
<tr>
<td>C Risk Profile</td>
<td>160</td>
</tr>
<tr>
<td>D Definite-clause Grammars</td>
<td>164</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Before beginning, plan carefully.

Marcus Tullius Cicero

Security testing is a process to determine that an information system protects data and maintains functionality as intended [Com04]. Yet, despite this quite mundane definition, security testing is a hard task [Tho03]. This is due to the fact that, traditionally, security testing is subclassified into functional and non-functional security testing [TyYsYy10]. Functional security testing aims at verifying positive requirements, addressing security functionality, whereas non-functional security testing aims at verifying negative requirements, targeting (hidden) software vulnerabilities, i.e., (for the most part) severe security bugs [Ale03]. Whereas for functional security testing, well-established techniques exist [MR05], for non-functional security testing those are scarce. Thus, in doing non-functional security testing, usually, a penetration testing-like, i.e., unstructured and artisanal style by simply black-box probing a software system is applied [MP04]. However, for non-functional security testing to be effective, incorporating negative requirements and corresponding risks is vital [MP04, VM04]. Unfortunately, coming up with such negative requirements for non-functional security testing is tricky. This results from that negative requirements are far more difficult to elicit than positive requirements, as they require security vulnerability knowledge [BM05], i.e., a security expert. Usually, humans are not programmed to anticipate and enumerate all of the bad circumstances that can pop up and that we need to protect against. Instead, we are wired to think about the “good” things we want the software to do [Voa08]. Hence, for non-functional security testing there is not only a great potential, but even greater need for improvement.

Web applications nowadays are widely used software systems that eminently suffer from this lack of effective non-functional security testing methods. These applications generally share and process sensitive user data which must be protected by all means. Thus, such multi-user applications are attractive targets for attackers.
Unfortunately, currently more than 90% of these applications are vulnerable, with a median number of 13 vulnerabilities per application according to a recent study from Cenzic [Cen13].

Among the possible attacks against web applications, two classes of attacks stand out due to their potentially severe damage and prevalence, viz. SQL injection (SQLI) and Cross-site scripting (XSS) attacks [Cen13, OWA13]. In SQLI, an attacker exploits the circumstance, that user input is not properly sanitized by the application, thus it can carry malicious database statements which, if executed, exploit the back-end database of the application. In XSS, an attacker also exploits the circumstance that user input is not properly sanitized, yet with the goal that malicious input is reflected by the application (e.g., in some HTML output) to be then executed in the victim’s browser.

Needless to say that these are not the only attacks, yet, according to OWASP they are the most severe as shown in Figure 1.3. In 2010 (Figure 1.1a), OWASP ranked SQLI (part of A1 - Injection) as one of the most severe application security risks, followed by XSS. In 2013 (Figure 1.2a), the situation changed a little, still, both vulnerabilities, viz. SQLI and XSS, are among the top three application security risks. Interestingly, A2 - Broken Authentication and Session Management, can be considered a consequence of SQLI and XSS. This is owing to that an attacker, on the one side, can manage to circumvent authentication by SQLI and, on the other side, steal a session using XSS. This circumstance further illustrates the seriousness of SQLI and XSS.

Apparent from Figure 1.3, as of the high number and complexity of existing application security risks, doing effective non-functional security testing requires thorough security vulnerability knowledge [BM05, ZFFB13]. This in turn results in high costs for testing. Consequently, application providers can decide either between investing such costs or taking the risk of offering a potentially vulnerable application if testing is done by a non-security expert tester. Unfortunately, as the (earlier mentioned) study of Cenzic shows, the latter usually seems to be the case [Cen13].

Existing techniques in preventing vulnerabilities like SQLI and XSS include various approaches, among them defensive coding, static code analysis, runtime monitoring, and testing. However, despite the merits of these techniques, they all have their disadvantages. Defensive coding [GVW03] is a labor intensive and error-prone task which requires rewriting applications to use secure libraries. Static code analysis [Bar94, Lou06] often produces false negatives and does not yield concrete example inputs which exploit a detected vulnerability. Runtime monitoring [HO05, PB06] comes with the cost of incurring additional overhead on the monitored application and does not detect any vulnerabilities prior to deploying the monitored application in a production environment. Further, runtime monitoring often produces false positives. Testing usually is done by black-box probing the system during plain penetration testing, i.e., inevitable negative requirements are
A1 - Injection
A2 - Cross-site Scripting (XSS)
A3 - Broken Authentication and Session Management
A4 - Insecure Direct Object References
A5 - Cross Site Request Forgery
A6 - Security Misconfiguration
A7 - Insecure Cryptographic Storage
A8 - Failure to Restrict URL Access
A9 - Insufficient Transport Layer Protection
A10 - Unvalidated Redirects and Forwards

(a) OWASP top ten application security risks for 2010 [OWA10].

A1 - Injection
A2 - Broken Authentication and Session Management
A3 - Cross-site Scripting (XSS)
A4 - Insecure Direct Object References
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross Site Request Forgery
A9 - Using Components with Known Vulnerabilities
A10 - Unvalidated Redirects and Forwards

(a) OWASP top ten application security risks for 2013 [OWA13].

Figure 1.3: OWASP top ten application security risks for 2010 [OWA10] and 2013 [OWA13].

neglected on the grounds of missing expert knowledge [AV12, MP04]. Consequently, testing often is not very effective by virtue of penetration testing's random nature. Further, penetration testing for the most part does not consider a specification and thus lacks a structured and reproducible approach.

Hence, for assuring the security of web applications w.r.t. their negative requirements novel methods are needed. Especially with regard to the key principles of information security, viz. confidentiality, integrity and availability such novel methods are needed to assure securely running systems. Further, these novel methods are required to be easy to use, i.e., highly automated, effective, i.e., detect existing vulnerabilities with a low rate of false positives, structured and reproducible [Ern12]. This dissertation attempts to solve these issues by introducing a novel non-functional security testing method by combining logic programming, knowledge engineering and model-based testing.

Logic programming comprises the representation of knowledge using facts and
further, the reasoning about this very knowledge by rules to infer new knowledge [BG94]. Its origins are in the processing of natural language [Col78]. Despite its origins however, logic programming has found its application in software testing, i.e., test case and test data generation [BM93, CGRSP10, Den91, GZAP10, LP00, VK95, ZCR02]. These testing approaches at the most follow a black-box testing style [Bei95], i.e., testing a system solely based on its specification without knowledge of its inner mechanisms (e.g., by utilizing its source code). Additionally, besides being applied for testing, logic programming techniques also found their usability in test data generation [GBR98, JBW94, Meu01]. Again, in large part only a black-box view of the system is used/available. So far, logic programming’s applicability for security testing has not been investigated. This dissertation shows novel applications of logic programming for non-functional security testing by a rule-based security risk analysis and test data generation by definite clause grammars (DCG) [Col78].

Knowledge engineering is the development and practical application of expert systems. Expert systems are programs that capture expertise that is used by experts to solve problems in a narrow domain [ZTV86]. In this sense, knowledge engineering recently has been applied in security engineering. Currently, three straws of research can be identified, (i) approaches that use formalized vulnerability knowledge to create attack graphs [JNO05, LIS06, RCSM09], (ii), methods that use vulnerability signatures (or attack patterns) to be matched against models of a system to identify potential flaws using static analysis [AGI12] and, (iii) expert systems for security analysis of networks [EZ01, OGA05]. Two prominent examples of vulnerability databases are the National Vulnerability Database (NVD) [NIS14] and the Common Vulnerabilities and Exposures (CVE) database [MIT14]. Although their contents are valuable for security engineering, the informal nature of their contents makes it tedious to integrate them efficiently into an automated method. This dissertation introduces a novel expert system by a Vulnerability Knowledge Base (VKB) that formalizes security vulnerability knowledge by attack patterns. This knowledge is used during our security risk analysis to identify potential flaws in a system model that are later evaluated during online testing.

Model-based testing (MBT) is the application of explicit models for software testing. In MBT, selected algorithms are used to generate test cases from models, i.e., formal specifications, of a system under test (SUT) or its environment in an automated and systematic manner [Sch12]. Thus, MBT can be classified as a form of black-box testing, as specifications instead of source code are used to generate test cases. Model-based security testing (MBST) then is the logical extension of MBT focusing on security requirements. MBST is a relatively new research field [SGS12]. Existing work in MBST currently for the most part focuses on functional security testing [BBNC01, J08, JW01, MFBLT08, WJ02], yet, non-functional security testing has only been investigated little [WWX07, SKR07]. The obvious advantages of MBT and MBST, respectively, are, among others, a high degree of automation, early detection of software bugs already at the design level, and
high coverage of the SUT by the resulting high quality test cases [UL10]. This dissertation presents a novel approach in MBST of negative requirements.

Existing research shows that the application of logic programming for testing is promising. This is thanks to its search-based nature to infer a solution which immediately can be mapped to the problem of designing test cases which eventually can also be considered a search problem, i.e., the search for meaningful test cases. Further, applying knowledge engineering by an expert system for security engineering is fruitful in that it eradicates the need for a security expert to do actual testing (clearly a security expert is needed to fill the expert system with knowledge, as well as maintaining this knowledge). The combination of these two techniques for designing a novel approach for non-functional security testing is compelling. Finally, if combined with techniques from MBT, an effective, i.e., structured and reproducible, non-functional security testing method can be established that makes security testing feasible for testers that are inexperienced in security.

1.1 Problem Statement and Research Challenges

Motivated by the discussion of the previous section, the research problem tackled in this dissertation is to develop a novel method for non-functional security testing of web applications that provides a high degree of automation with a special focus on autonomously generating high-quality negative security test cases. This is achieved by the integration of knowledge engineering and automated reasoning, i.e., logic programming, into model-based security testing. Although logic programming in testing, knowledge engineering in security engineering, and MBT already have been investigated on their own behalf, the combination of all three to enhance automation in non-functional security testing has not been considered so far.

Besides developing and evaluating our non-functional security testing method itself, the main research challenges tackled in this dissertation are:

**RC1 - Codification of security vulnerability knowledge for non-functional security testing.** This contains the definition of logical predicates to formalize and store (i) attack behavior and (ii) the format of attack data inside a knowledge base, e.g., a Vulnerability Knowledge Base (VKB). Further, for (ii), this requires the definition of deduction rules to generate test data during testing.

**RC2 - Automated security risk analysis and risk assessment.** This contains both, the design of (i) a security risk analysis for software systems and (ii) a risk assessment procedure that enables to rank (assess) the risks, resulting from the security risk analysis. Such an assessment yields that test cases that are derived from risks (which were identified by our security risk analysis) can be ranked by, e.g., prioritizing them by the severity of an objected risk.
**1.2 Problem Solution and Contributions**

**RC3** - *Automated, non-functional security testing of web applications to detect SQLI and XSS vulnerabilities.* This contains the design of an automated, non-functional security testing method for non-functional security testing of web applications to detect SQLI and XSS vulnerabilities. RC3 is subdivided into the following sub-challenges:

**RC3.1** - Design of system and testing artifacts as well as their declarative modeling languages by logical predicates.

**RC3.2** - Design of domain-specific languages (DSL)s for the just mentioned system and testing artifacts. This also requires the design of model translators between the declarative and domain-specific representations of those very artifacts. Declarative representations are necessary for that our VKB can deal with system and testing artifacts, whereas a domain-specific representation is necessary for that humans can read and understand the models. Further, a domain-specific model in a high-level modeling language (compared to low level logical predicates) is easily processible by our test engine (see RC4).

**RC3.3** - Design of an oracle mechanism to detect successful SQLI and XSS attacks (i.e., test cases) against the SUT.

**RC4** - *Automated generation, execution and evaluation of negative security test cases.* This contains the design of a test case generator and a test engine that both use facilities of our VKB, viz. the security risk analysis and test data generation (see RC1 and RC2), as well as mechanisms for test evaluation of executed test cases.

---

The **key idea** of this dissertation is to show the successful application of logic programming and knowledge engineering, together with MBT, to design a novel method for non-functional security testing of web applications. We thus introduce a novel method in MBST for the successful verification of existing, hidden security vulnerabilities in web applications. Foundations of MBT provide the necessary mechanisms to formally describe a system under test (SUT) by a model for then algorithmically derive test cases, thus advancing non-functional security testing away from its penetration-testing like style to a structured and reproducible testing process. Our VKB occupies a key capacity, with a three-fold purpose:

(i) provide the necessary means for storing security vulnerability knowledge (attack behavior and data) in a usable manner by its extensional database (the facts),
(ii) provide the necessary functionality to perform an automated security risk analysis and risk assessment by its intensional database (the rules), and

(iii) provide the necessary mechanisms to generate test data for testing.

We first apply techniques from knowledge engineering, i.e., knowledge representation, to formalize security vulnerability knowledge. This knowledge then is used in accordance with a security problem, a declarative system model of a SUT, to reason on potential existing security risks in the SUT, i.e., to infer new knowledge by a risk profile of the SUT. This reasoning step comprises our security risk analysis and the risk assessment. Although the idea of using a security risk analysis to establish a risk profile (or model) for a software system is well explored [AD02, DRRS02, KS05], currently its main drawback is its dependence on a security expert. We detach this dependence by our VKB. The resulting risk profile from the security risk analysis then acts as an executable specification for offline non-functional security testing of the web application, described by the security problem. Thus, our security risk analysis already comprises test generation. Additionally, we show how to use DCGs to synthesize malicious inputs, injected into the SUT during testing to verify the identified and potentially existing, hidden security vulnerabilities in the SUT. DCGs are a powerful mechanism in logic programming that support the formalization of context-free or context-sensitive grammars to then process or generate sentences in a language. For automated test data generation in security testing, well-known approaches like fuzzing [TDM08], which generate data either based on mutating valid input strings or by generating malicious ones from scratch, exist. However, for fuzzing, custom tools need to be implemented, i.e., there exist no off-the-shelf customizable test data generators [MYGG12].

According to Landwehr et al. [LBMC94] vulnerabilities in software systems can be classified as either inadvertent and intentional (Figure 1.4). The former class, i.e., inadvertent, essentially subsumes bugs, whereas the latter, i.e., intentional, consists of both “malicious” vulnerabilities, which were deliberately inserted (e.g., trapdoors or logic bombs) and “non-malicious” vulnerabilities that are side-effects of features that were deliberately added to the system (e.g., covert channels or inconsistent attack paths) [LBMC94]. In this dissertation we address inadvertent vulnerabilities, i.e., vulnerabilities due to insecure coding and the like, as listed in Figure 1.3.

In 2011 Felderer at al. [FAZB11] proposed a classification for MBST as shown in Figure 1.5. This classification has two dimensions, viz. automated test generation and risk. The first dimension, viz. automated test generation, considers how much of the SUT and the security requirements is captured by (semi-)formal models. Obviously, the fewer information on the SUT and security requirements is available, the more individual knowledge by a test engineer and a security expert is needed to design meaningful test cases. Fully automated test generation is only possible with formal and complete models which typically are not available. Fortunately, the
1.2 Problem Solution and Contributions

Figure 1.4: Classification of software security flaws w.r.t. Landwehr et al. [LBMC94]. Our proposed method as introduced in this dissertation considers inadvertent vulnerabilities, i.e., vulnerabilities due to insecure coding and the like, as listed in Figure 1.3. Consideration of risk models can efficiently support the test engineer in test design and prioritizing of test cases for test execution. This is reflected by the second dimension, viz. risk.

W.r.t. the MBST classification of Felderer et al. as shown in Figure 1.5, our proposed method can be classified as automated RB security test generation (where RB stands for risk-based). This is because our proposed method builds on codified security vulnerability knowledge to automatically generate a risk profile. Then, as of being fully integrated into the testing process, this risk profile allows for fully automated risk-based security test generation.

Our method leads to efficient tools to eradicate the need for a security expert for non-functional security testing, except for creating and maintaining the expert system, i.e., the VKB. The availability of efficient off-the-shelf tools for building such expert systems [SBF98] makes our method further compelling. Additionally, to the best of our knowledge, so far, no off-the-shelf tools for performing an

<table>
<thead>
<tr>
<th>Genesis</th>
<th>Intentional</th>
<th>Non-Malicious</th>
<th>Malicious</th>
<th>Trojan Horse</th>
<th>Non-Replicating</th>
<th>Replicating (Virus)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Trapdoor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logic/Time Bomb</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Covert Channel</td>
<td>Storage</td>
<td>Timing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Other</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inadvertent</th>
<th>Validation Error (Incomplete/Inconsistent)</th>
<th>Domain Error (Including Object Re-use, Residuals and Exposed Representation Error)</th>
<th>Serialization/Aliasing (Including TOCTTOU Errors)</th>
<th>Identification/Authentication Inadequate</th>
<th>Boundary Condition Violation (Including Resource Exhaustion and Violable Constraint Errors)</th>
<th>Other Exploitable Logic Error</th>
</tr>
</thead>
</table>

|                  |                                           | Destination Error (Including Input parameter Error)                          | Inconsistencies in Purpose (Including Logic Flaw) | Inconsistencies in the Design (Including Logical Flaw) | Inconsistencies in the Code (Including Syntax Flaw) | 
|------------------|-------------------------------------------|---------------------------------------------------------------------------------|--------------------------------------------------|--------------------------------------|---------------------------------------------------------------------------------|-----------------------------|
1.2 Problem Solution and Contributions

automated security risk analysis or which implement a fully customizable test data generator are available.

The research paradigm adopted in this dissertation is design science research [25, 26]. According to van [26], the main purpose of design science research “is achieving knowledge and understanding of the problem domain by building an application of a design artifact”. This essentially motivates the development of a prototype implementation of a proposed novel method and its evaluation w.r.t. usability, effectiveness, and efficiency. We have evaluated our method and its tool implementation by two case studies using a vulnerable web application that is designed to implement a number of vulnerabilities w.r.t. OWASP’s 2013 top ten list [27].

Contributions. This dissertation contributes to current and ongoing research in the area of non-functional security testing in web applications. Its core contribution is the VKB, an expert system that blatantly decreases the knowledge necessary for doing effective non-functional security testing. Further, the application of techniques from MBT advances non-functional security testing to an effective, i.e., structured and reproducible testing method, away from its prevalent penetration testing-like, i.e., unstructured and artisanal style. Hence, our method and, in particular, its tool implementation provide a valuable contribution in the area of MBST.

Figure 1.5: MBST classification w.r.t. Felderer et al. [28]. Our proposed method falls into the category automated RB security test generation (where RB stands for risk-based).
The main contributions of this dissertation are as follows:

**C1 -** An expert system, i.e., our VKB, for formalizing and storing security vulnerability knowledge. This expert system further comprises a test data generator (contributes to RC1 and RC4) [ZFB12a, ZFB13, ZFFB13, ZFB14].

**C2 -** A security risk analysis and risk assessment procedure for evaluating the risks resulting from the security risk analysis by their severity. The resulting risk assessment allows for prioritization of test execution (contributes to RC1, RC2 and RC4) [ZFB12a, ZFB12c, ZFB13, ZFFB13, ZFB14, ZFB14].

**C3 -** Automatic generation of a test suite for non-functional security testing. This test suite is available as an executable specification that results from our security risk analysis by the risk profile for a SUT (contributes to RC3 and RC4) [ZFB12c, ZFFB13, ZFB14, ZFB14].

**C4 -** Sets of logical predicates for declaring a security problem, i.e., the system model of a SUT and a risk profile (or executable specification). Further, two DSLs for representing those two declarative models in a human-readable way. Necessary model translators are also implemented (contributes to RC1, RC2 and RC3) [ZFB12b, ZFB12c, ZFB14, ZFB14].

**C5 -** A posteriori evaluation of test runs by test oracles. The mechanism is capable of describing and inferring different attack outcomes, i.e., the result of exploiting a vulnerability by a single test run (contributes to RC3 and RC4) [ZFB12b, ZFB12c, ZFB14, ZFB14].

**C6 -** Successful detection and verification of SQLI and XSS vulnerabilities in web applications (contributes to RC3) [ZFB13, ZFB14, ZFB14].

**C7 -** A tool implementation and its evaluation for model-based, non-functional security testing of web applications on the grounds of our VKB (contributes to RC1, RC2, RC3 and RC4) [ZFB14, ZFB14].

1.3 Structure of this Thesis

This dissertation at hand introduces and discusses a novel model-based, non-functional security testing method for web applications and its tool implementation. We discuss latest state-of-the-art in topics relevant to this dissertation, the underlying theory of our method, as well as our novel method and its formal foundations. Further, we provide case studies and a thorough discussion of our results and their implications. The contents of each chapter are as follows:
Chapter 1. Chapter 1 starts by introducing and motivating the problem discussed in this dissertation. Next, the problem statement is formulated (Section 1.1). On the grounds of the problem statement we then sketch our solution and list the resulting contributions (Section 1.2). Finally, we outline the structure of this dissertation (Section 1.3).

Chapter 2. Chapter 2 discusses foundations (Section 2.1) and state-of-the-art (Section 2.2) in topics relevant to this dissertation. First, we give a brief refresher on logic programming (Section 2.1.1), its solution strategy (Section 2.1.1.1) and definite-clause grammars (DCG; Section 2.1.1.2), followed by a brief discussion of knowledge engineering (Section 2.1.2). Next, we give a basic introduction to web applications (Section 2.1.3) followed by introducing our notion of software vulnerabilities and attacks (Section 2.1.4), with a special emphasis on SQLI and XSS attacks, the two kinds of attacks targeted in this dissertation (Section 2.1.4.1). We then introduce the mechanics of security testing (Section 2.1.5) by detachedly considering the two strains in security testing, viz. functional (Section 2.1.5.1) and non-functional security testing (Section 2.1.5.2). We conclude Section 2.1 with a discussion of the fundamental ideas of MBST (Section 2.1.5.3). We then discuss state-of-the-art (Section 2.2). We start by giving a review on the application of logic programming in testing (Section 2.2.1) and knowledge engineering in security engineering (Section 2.2.2). We then survey various areas in security testing (Section 2.2.3), relevant for this dissertation, viz. penetration testing (Section 2.2.3.1), MBST (Section 2.2.3.2), and web application security testing (Section 2.2.3.3).

Chapter 3. Chapter 3 introduces our model-based, non-functional security testing method for web applications (Section 3.1). We start with a discussion of the security problem, the model of the SUT (Section 3.1.1). Next, we present our VKB (Section 3.1.2) and its major components, the extensional database (the memory, or facts; Section 3.1.2.1), the intensional database (the reasoning entity comprising the deduction rules for our security risk analysis; Section 3.1.2.2), test data generation (Section 3.1.2.3), and establishment and maintainability of the VKB (Section 3.1.2.4). Following we discuss the risk profile (Section 3.1.3), the result of our security risk analysis. We then briefly discuss test generation (Section 3.1.4), followed by a more thorough discussion of test execution (Section 3.1.5) and test evaluation (Section 3.1.6). Before we conclude this chapter with the formal foundations of our method (Section 3.3), we introduce the high-level modeling facilities of our method (Section 3.2), viz. a web spider for automatically establishing a security problem (Section 3.2.1), the already mentioned DSLs (Sections 3.2.2 and 3.2.3) and respective model translators (Sections 3.2.4).

Chapter 4. Chapter 4 discusses the experimental evaluation of our model-based non-functional security testing method for web applications. We start by a discussion of our case studies (Section 4.1), in particular, the application we used to
1.3 Structure of this Thesis

evaluate our method, viz. Damn Vulnerable Web Application (DVWA) \cite{Ran14},
the lab setup (Section 4.1.2), and the concrete execution of the case studies (Section 4.1.3). We then present the findings of our two case studies (Section 4.2), in which we used the tool implementation of our method to detect SQLI and XSS vulnerabilities in DVWA at two different security levels, viz. low (Section 4.2.1) and medium (Section 4.2.2).

Chapter 5. Chapter 5 is devoted to the discussion of our method and of the findings from Chapter 4 w.r.t. state-of-the-art as discussed in Chapter 2. First, we interpret our findings from Chapter 4 (Section 5.1), then relate our work to state-of-the-art as discussed in Chapter 2 (Section 5.2). We conclude with a discussion of the implications of our work (Section 5.3).

Chapter 6. Chapter 6 concludes this dissertation. We first summarize our work (Section 6.1), followed by a discussion of the key findings and implications of this dissertation (Section 6.2). We conclude with an outlook on future work (Section 6.3).
Chapter 2

Foundations and State-of-the-Art

Knowledge is the life of the mind.

Abu Bakr

In this chapter we introduce the relevant foundations (Section 2.1) and state-of-the-art (Section 2.2) in topics relevant to this dissertation, viz. logic programming, knowledge engineering and security testing.

2.1 Foundations

Following we introduce the necessary foundations for this dissertation. We start by giving a refresher on logic programming (Section 2.1.1), its solution strategy (Section 2.1.1.1), as well as Definite Clause Grammars (DCGs; Section 2.1.1.2), the underlying mechanism of our test data generator. Next, we briefly discuss knowledge engineering, a practical application of logic programming (Section 2.1.2). We then give a short introduction to web applications (Section 2.1.3) followed by introducing our notion of software vulnerabilities and attacks (Section 2.1.4) and an illustrative discussion of SQLI and XSS attacks (Section 2.1.4.1), the two kinds attacks our method detects (Chapter 3). We then continue (Section 2.1.5) with detached discussions of the methodology of functional (Section 2.1.5.1) and non-functional security testing (Section 2.1.5.2). We conclude this Section with a discussion of the fundamental ideas of MBST (Section 2.1.5.3).

2.1.1 Logic Programming

Logic programming is a quite powerful, yet simple declarative programming paradigm build upon the notion of deduction from automated theorem proving. However, contrary to automated theorem proving, which uses deduction to prove or disprove a mathematical theorem, in logic programming, actual values are computed as a solution during deduction. Values necessary for computation of a solution are
declared as facts, i.e., knowledge, and the actual computation steps to infer a solution as rules, i.e., the logical reasoning procedure. Thus, logic programming is a valuable formalism to deal with problems in knowledge engineering and reasoning to infer new knowledge.

Logic programs are sets of rules (clauses) of the form

$$A \leftarrow L_1, \ldots, L_m$$

where $A$ (the head) is an atom and $L_1, \ldots, L_m$ (the body) are literals (atoms or negated atoms). Rules with an empty body, i.e., only consisting of a head, are called facts. Atoms are defined to be well-formed formulas of the form $p(t_1, \ldots, t_n)$ where $p$ is a predicate symbol of an alphabet $\Sigma$ and $t_1, \ldots, t_n$ are terms over alphabet $\Sigma$. Further, for each term $t$, $t \equiv c \mid X \mid p(t_1, \ldots, t_n)$. Hence, terms are defined recursively to be either (i) a constant $c$ of alphabet $\Sigma$, (ii) a variable $X$ bound to an element of alphabet $\Sigma$ or (iii), an n-ary predicate $p$ of alphabet $\Sigma$ with terms $t_k$ over alphabet $\Sigma$, e.g., $\Pi_1 = \{q(a), q(b), q(c), s(a), s(c), w(a), w(b), p(X) \leftarrow q(X), \neg r(X), r(X) \leftarrow w(X), \neg s(X)\}$.

With $\Sigma = \{a, b, c, p, q, r, s, w\}$.

Binding (or unifying) a variable $X$ to a term over alphabet $\Sigma$ is called substitution and denoted as $X \mapsto t$ (read $X$ “is mapped” to $t$). The alphabet $\Sigma$ may also be called the domain of discourse, i.e., the universe of things to which the variables of a formal theory may be bound. If a solution of a logic program can be found, i.e., all variables are bound to elements of an alphabet $\Sigma$ and hence, all rules are ground, a program is called satisfiable. Contrary, if there exists no solution, a program is not satisfiable.

To now intuitively show whether $p(a)$, $p(b)$ or $p(c)$ is a consequence of program $\Pi_1$, i.e., calculate a solution for program $\Pi_1$, we draw three sets of proof trees under all possible substitutions for variable $X$, i.e., $X \mapsto a$, $X \mapsto b$, and $X \mapsto c$, as shown in Figure 2.1.

We now see that $p(a)$ and $p(c)$ (across) are consequences, i.e., solutions, for

---

1We follow Prolog’s convention that variable names start with an uppercase letter; additionally, such a naming scheme fosters understanding of discussed programs.
2.1 Foundations

Figure 2.1: Proof trees for program \( \Pi_1 \). Using all possible substitutions for \( X \), i.e., \( X \mapsto a \), \( X \mapsto b \), and \( X \mapsto c \), we get all possible proof trees for program \( \Pi_1 \) to infer its solution(s).

Program \( \Pi_1 \) whereas \( p(b) \) is not. In logic programming, \( p(X) \) then traditionally is called a goal and denoted as \( \leftarrow p(X) \). Observe that the proof trees are drawn so that negative nodes \( \neg g \) are leaves, and new trees rooted at \( g \) are investigated.

Such a simple, but effective solution strategy as just applied on program \( \Pi_1 \) allows to also compute solutions to non-trivial problems (e.g., graph coloring) efficiently with logic programming. In fact, Prolog applies a strategy which can be captured by the notions of proof trees as briefly discussed in the following.

2.1.1.1 SLD-Resolution

Classic SLD resolution (Selective Linear Definite Clause resolution) first was described by Kowalski \cite{Kow74}. Kowalski interpreted SLD resolution as procedure invocation, which, for a given selected procedure call \( A_i \) in a procedure

\[
A \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n
\]

and given procedure

\[
B \leftarrow B_1, \ldots, B_m,
\]

whose name matches (with a substitution \( \Theta \)) the selected procedure call \( A_i \), derives a new procedure

\[
(A \leftarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\Theta
\]

with \( \Theta = \{ A_i \mapsto B_1, \ldots, B_m \} \). Thus, SLD resolution can be applied to derive new procedures from old procedures \cite{Kow74}.

More informally, for a given selected procedure call, i.e., the currently selected goal, SLD resolution delineates a search tree of alternate computations rooted in
2.1 Foundations

1. \( q(a) \).
2. \( q(b) \).
3. \( q(c) \).
4. \( s(a) \).
5. \( s(c) \).
6. \( w(a) \).
7. \( w(b) \).
8. \( p(X) \leftarrow q(X), \neg r(X) \).
9. \( r(X) \leftarrow w(X), \neg s(X) \).

Figure 2.2: Program \( \Pi_1 \) reproduced in Prolog syntax. The syntax of Prolog more or less resembles our formal notation of logic programs except for \( \leftarrow \), which is replaced by \( :- \) and that each rule ends with a period.

the goal. Associated child nodes for new procedures are added to the search tree, if a literal of the goal unifies with a given procedure call \( B \) (i.e., the new procedure). A leaf node whose derived new procedure is the empty set (i.e., \( \emptyset \)) is a success node. Contrary, if a node’s literals do not unify with any elements of alphabet \( \Sigma \) or any term \( t_k \) over alphabet \( \Sigma \), and thus also not with the initial goal (i.e., the root), it is a failure node. A solution to the given selected root procedure (i.e., a substitution \( \theta \)) then is defined by branches of the search tree, which yield in success nodes.

In a more mathematical sense, the solution to a logic program \( \Pi \) is retrieved by calculating its least Herbrand model, i.e., \( H^\Pi_L \subset H_B \). Here, \( H_L \) denotes the least Herbrand model, a consistent subset (i.e., containing no refutable goal) of the Herbrand base \( H_B \). The Herbrand base \( H_B \) contains the set of all possible ground goals, which can be represented by the alphabet \( \Sigma \) of a logic program \( \Pi \). Then, if a least Herbrand model \( H^\Pi_L \) for a program \( \Pi \) does not contain any free variables (i.e., all variables are bound and thus, all terms grounded), it is called a solution to the program \( \Pi \).

SLD resolution in Prolog uses a depth-first search strategy to select a new goal. Hence, Prolog always unifies the given selected goal with the topmost matching rule — the new goal — encountered in a program and further (if present), the rule’s body from left to right. In case of reaching a failing literal, Prolog uses backtracking to reach the last known successful literal to restart searching from this point along an alternate branch of the search tree. Remember program \( \Pi_1 \), reproduced in Figure 2.2 in Prolog syntax. With \( p(X) \) as the currently selected goal, Prolog selects the first literal of the body, i.e., \( q(X) \) to match next. From program \( \Pi_1 \) we know that \( q(a) \) thus, \( q(X) \) is unified. As \( p(a) \) is already ground, Prolog then would continue with the next ungrounded literal, i.e., \( \neg r(X) \).

As a side note, Prolog’s efficient resolution strategy comes at the cost of omitting the occurs check. The occurs check normally would cause unification to fail if a possible substitution contains the given selected goal, i.e., it avoids entering an
2.1 Foundations

endless loop. Prolog loops with left recursion traditionally are very prone to this problem. Consider Figure 2.3 for a small example. For the goal \( p(X) \) the program on the left contains left recursion in its first rule, i.e., it always matches \( p(X) \) with itself and enters an infinite loop. Contrary, the program on the right avoids this problem due to a different order of rules, i.e., for the goal \( p(X) \) it first matches \( p(a) \) and avoids an infinite loop. This pitfall sometimes makes writing Prolog programs a little tricky.

(a) Prolog program with infinite loop. 
(b) Prolog program avoiding infinite loop.

Figure 2.3: Examples of Prolog programs with and without an infinite loop. The Prolog program on the left contains an infinite loop by the first rule, whereas the program on the right avoids this loop.

2.1.1.2 Definite-clause Grammars

Prolog naturally provides support for dealing with context-free grammars (CFGs) due to its original intent to build a system for conversing in French [Col78]. Thus, using DCGs, Prolog can be used for both, to parse but also to generate strings of a language, described by a CFG [PW80, PW83]. Consider the CFG from Figure 2.4.

\[
\begin{align*}
\langle \text{sentence} \rangle & \rightarrow \langle \text{noun_phrase} \rangle \langle \text{verb_phrase} \rangle \\
\langle \text{noun_phrase} \rangle & \rightarrow \langle \text{determiner} \rangle \langle \text{noun} \rangle \mid \langle \text{noun} \rangle \\
\langle \text{verb_phrase} \rangle & \rightarrow \langle \text{verb} \rangle \langle \text{noun_phrase} \rangle \mid \langle \text{verb} \rangle \\
\langle \text{determiner} \rangle & \rightarrow \text{the} \\
\langle \text{noun} \rangle & \rightarrow \text{man} \mid \text{woman} \mid \text{coffee} \\
\langle \text{verb} \rangle & \rightarrow \text{drinks} \mid \text{likes}
\end{align*}
\]

Figure 2.4: A CFG sample by a small subset of the English language. Using its production rules, this CFG allows to produce or parse valid, although not always meaningful, English sentences.

Among others, this grammar allows to produce the following sentences:

- the man drinks coffee, or
In reverse however, we can also use the CFG from Figure 2.4 to check whether a given sentence is valid w.r.t. it, e.g.,

- the man likes the woman, or
- the man woman.

Obviously, the first sentence is valid w.r.t. the CFG from Figure 2.4, but the second, the man woman, is not, as a sentence requires a verb phrase between to noun phrases.

In Prolog, we can represent CFGs by DCGs, which internally use lists, more detailed, difference lists. Lists are an important concept in non-numeric programming languages. They represent ordered sequences of elements with any length. In Prolog, the elements of a list can be any term (constants, variables, structures), including lists themselves [CM84]. Prolog’s lists can be interpreted as trees, i.e., the list (a,b,c,[]) can be represented as shown in Figure 2.5. The end of a Prolog list commonly is represented as the empty list, i.e.,[].

Figure 2.5: Prolog interpretation of the list (a,b,c,[]) as a binary tree. Each element of the list is a left subtree with one leaf node, whereas the list itself grows along its right subtrees, ending with the empty list, i.e.,[] as the only right leaf node.

However, using Prolog’s plain lists yields in that a list, which is a result to a goal, is ordered backwards. This is due to SLD resolution, which at the end of the day boils down to recursively stepping down while matching goals until it reaches \( \top \) or \( \bot \). When returning, i.e., stepping aback to the initial goal, a list then is build up in reverse order. Thus, if using plain lists, Prolog would also generate sentences in reverse order, e.g., woman some hates man every. Contrary to plain lists, difference lists are Prolog lists, whose last element is a variable paired with itself. From a more operational point of view, a difference list in Prolog is a possibly open-ended list that maintains an explicit end pointer, i.e., (a,b,c|Z). As long as \( Z \) is not instantiated, i.e., \( Z = [] \), the list remains open-ended. The trick now is that when stepping aback the end pointer \( Z \) can be instantiated to anything, i.e., \( Z = [d|w] \),
resulting in the list \((a, b, c, d | W)\). Observe that this yields another open-ended list which can be further extended by instantiating \(W\). If however \(W\) then is instantiated to a symbol, i.e., \(W = [e, \square]\), the resulting list, i.e., \((a, b, c, d, e, \square)\) becomes closed. Thus, processing DCGs with such difference lists whose end pointer is instantiated during unification when stepping aback, allows to generate lists in order, e.g., \(\text{every man hates some woman}\). For the sake of brevity, the last \(\square\) that indicates the end of a list generally is omitted.

We thus can represent our sample CFG from Figure 2.4 with program \(\Pi_2 = \)

\[
\text{sentence}(S, S_0) \leftarrow \text{noun_phrase}(S_0, S_1), \text{verb_phrase}(S_1, S), \\
\text{noun_phrase}(S, S_0) \leftarrow \text{determiner}(S_0, S_1), \text{noun}(S_1, S_2) \\
\text{noun_phrase}(S, S_0) \leftarrow \text{noun}(S_0, S), \\
\text{verb_phrase}(S_0, S) \leftarrow \text{verb}(S_0, S_1), \text{noun_phrase}(S_1, S), \\
\text{verb_phrase}(S_0, S) \leftarrow \text{verb}(S_0, S), \\
\text{determiner}(S_0, S) \leftarrow \text{connects}(S_0, \text{the}, S), \\
\text{noun}(S_0, S) \leftarrow \text{connects}(S_0, \text{man}, S), \\
\text{noun}(S_0, S) \leftarrow \text{connects}(S_0, \text{woman}, S), \\
\text{noun}(S_0, S) \leftarrow \text{connects}(S_0, \text{coffee}, S), \\
\text{verb}(S_0, S) \leftarrow \text{connects}(S_0, \text{drinks}, S), \\
\text{verb}(S_0, S) \leftarrow \text{connects}(S_0, \text{likes}, S).
\]

More literally, we can read the first rule (clause) to be a “sentence which extends from \(S_0\) to \(S\) if there exists a noun_phrase from \(S_0\) to \(S_1\) and a verb_phrase from \(S_1\) to \(S\)” [PW80]. The attentive reader might have realized that we introduced an additional predicate \(\text{connects}/3\), whose purpose is to embed terminal symbols into sentences, e.g., terminal symbol \(\text{the}\) lies between \(S_0\) and \(S\).

For the sake of convenience, Prolog offers syntactical sugar in declaring DCGs, i.e., we must not explicitly declare any difference lists, e.g., \(S, S_0, S_1, S_2\), but instead just write down a CFG in a more natural form, resembling its definition in BNF. Thus, in Prolog, we declare our sample CFG as shown in Figure 2.6.

So far, we have only considered CFGs in Prolog. However, by declaring grammar rules with additional parameters, Prolog allows us to represent context-sensitive grammars (CSG). Then, in a grammar rule’s body, we can use normal Prolog predicates as contextual predicates by putting them inside \{\ldots\}. This way, Prolog interprets them as normal Prolog predicates, i.e., not a grammar predicate, and thus allows us to consider a context in parsing and producing any sentences. Figure 2.7 shows a sample Prolog DCG implementing a CSG for parsing strings of the form \(a^n b^n c^n\).
2.1 Foundations

1. sentence $\rightarrow$ noun_phrase, verb_phrase.
2. noun_phrase $\rightarrow$ determiner, noun ; noun.
3. verb_phrase $\rightarrow$ verb, noun_phrase ; verb.
4. determiner $\rightarrow$ "the".
5. noun $\rightarrow$ "man" ; "woman" ; "coffee".
6. verb $\rightarrow$ "drinks" ; "likes".

Figure 2.6: Sample CFG from Figure 2.4 in Prolog notation. Prolog offers syntactical sugar by $\rightarrow$, which allows to avoid the explicit declaration of necessary difference lists.

1. s $\rightarrow$ a(N), b(N), c(N).
2. a(0) $\rightarrow$ "".
3. a(M) $\rightarrow$ "a", a(N), \{M is N + 1\}.
4. b(0) $\rightarrow$ "".
5. b(M) $\rightarrow$ "b", b(N), \{M is N + 1\}.
6. c(0) $\rightarrow$ "".
7. c(M) $\rightarrow$ "c", c(N), \{M is N + 1\}.

Figure 2.7: Using Prolog's DCGs to represent CSGs. The underlying CSG allows to produce or parse strings of the form $a^n b^n c^n$.

As in the following we also show logic programs in Prolog syntax, we just quickly discuss Prolog's syntactical differences to our formal notation of logic programs:

- In Prolog variable names start with an uppercase letter. This is to distinguish them from constants and predicates, which start with a lowercase letter.

- Logical negation, i.e., $\neg$, is done by the predicate not/1. For this purpose Prolog also offers the predicate \+/1. However, contrary to not/1, \+/1 only checks whether a literal has no provable instances, instead of checking whether it can be proven w.r.t. a substitution.

- Logical implication, i.e., $\leftarrow$, is denoted by :-.

- Production rules in DCGs use $\rightarrow$ as syntactical sugar, instead of :- . Further, by using $\rightarrow$, Prolog does not require defining any difference lists as parameters. Prolog internally transforms (rewrites) DCG rules to standard Prolog rules, i.e., rules using :- and difference lists.

- In Prolog, an initial goal to match, e.g. $\leftarrow p(X)$ is called a query and denoted by ?-p(X).

- We denote any standard Prolog predicate with predicate/X, where predicate stands for the predicate's name and X for the number of parameters. For a
DCG specific predicate, we write \texttt{predicate//x} to distinguish it from normal Prolog predicates.

- Finally, each rule in Prolog ends with a period.

### 2.1.2 Knowledge Engineering

Knowledge engineering is the development and practical application of expert systems. Expert systems are programs that capture expertise that is used by experts to solve problems in a narrow domain \cite{ZTV86}. A key factor for the performance of an expert systems is knowledge \cite{Fei84}. According to this, knowledge is of two types, viz.:

- \textit{facts of the domain}, i.e., the widely shared knowledge that is written in textbooks, and
- \textit{heuristic knowledge}, i.e., the rules of plausible reasoning (or inference procedures).

The attentive reader now may already realizes knowledge engineering’s relation to logic programming in that it comprises one of the main areas of application of logic programming. As introduced in Section 2.1.2, logic programs are sets of facts and rules, i.e., shared and heuristic knowledge. Thus, logic programming eventually is an approach to knowledge engineering by its declarative (facts) and procedural (inference procedures) representations of knowledge. Clearly, logic programming is not the only approach to knowledge engineering, another popular method is the applications of ontologies \cite{SBF98}, however, this is beyond the scope of this dissertation, and thus is not considered further.

Contrary however to logic programming, whose roots are in the processing of natural language \cite{Col78}, the roots of knowledge engineering are manifold, viz. \cite{SBF98}:

- as a transfer and transformation process of expertise from a knowledge source to a program,
- as a model construction process, or
- as a problem solving method.

Consequently, also the areas of application of knowledge engineering nowadays are manifold, e.g., for, as already mentioned, modeling and problem solving, but also in the field of planning systems \cite{CMGF+07} or machine learning \cite{LS95}.

We now however skip any further discussion of knowledge engineering, as we already described its mechanics by one of its possible approaches, viz. logic programming, in Section 2.1.1.
2.1.3 Web Applications

Web applications originally evolved from classic web systems [Con02]. Web systems, which have been used since the early days of public internet, i.e., when Tim Berners–Lee launched his first web sites at CERN, are static, content-centric systems with the sole purpose of providing static information to its users. Contrary to this, web applications extend web systems by an additional layer that introduces functionality, i.e., business logic, for that a task can be achieved. Figure 2.8 shows this difference. While a web system (Figure 2.8a) solely processes user requests by returning some document that is retrieved from accessing a web server’s underlying file system, a web application (Figure 2.8b) that runs inside a web server processes user requests by incorporating some back-end database and returns with a computed response. Such a response may well be some document, however, in case of a web application its content is not static anymore, but dynamically generated on the ground of user input and some computation(s) that is/are implemented by the web application.

Figure 2.8: Difference between a classic web system and a web application. Whereas a web system is content-centric and just provides information in a static manner (i.e., by forwarding documents stored in a local file system on the web server), a web application introduces functionality that allows to achieve some task (i.e., by the web applications implemented business logic).

Classic web systems are built upon HTTP (HyperText Transfer Protocol) [FGM+99] and HTML (HyperText Markup Language) [LHRJ99]. Whereas the former, i.e., HTTP, provides the means by which a client (e.g., a web browser) can communicate with a web server, the latter, i.e., HTML, specifies how content is rendered in the client’s web browser. Being a descendant of web systems, web applications naturally also build on those two enabling technologies, yet, in case of
2.1 Foundations

HTML with a different intent. Whereas for web systems, HTML only is used to render static content in a client’s browser, in case of web applications, HTML has additional purposes, viz.

- provide a user interface to a client by its various input elements (e.g., forms), i.e., enabling communication with the underlying web application, and
- encapsulate functionality that is provided by the web application through user interaction elements (e.g., push buttons).

Although both, classic web systems and web applications are multi-user applications, in case of web applications multi-user support is far more difficult to achieve. This is by virtue of that in case of web applications, content returned to a user is not static anymore, but dynamically generated on the grounds of what a user provided as input. Consequently, web applications need facilities to differentiate among different users, i.e., client state management, for that each user retrieves the response that was computed for him (observe that in case a user retrieves a response computed for another user, a case of data leakage is present). Following we briefly discuss the two major approaches for client state management, viz. cookies and sessions.

Cookies  Cookies are small pieces of data that are issued by a web server and that the web server can request the client to hold on to. Upon a subsequent request of the client to the web server the cookie is also submitted by the client. Subsequently, the server can identify the client. Initially, the cookie is submitted from the server to the client as part of an HTTP request. In the same manner, the client also provides the cookie as part of an HTTP request to the server. The typical size of a cookie is between 100 and 1K bytes. Both, the client and the web server may change the data that is stored inside a cookie. Unfortunately, the whole mechanism of using cookies is not without faults. First, using cookies allows to track activities of a user among multiple web sites. Second, upon an attacker manages to steal some other user’s cookie, authentication can be spoofed. This in turn enables an attacker to tamper around with other user’s data which may result in severe attacks, e.g., data leakage or impersonation.

Sessions  A session represents a single cohesive use of the system. Such a use generally involves multiple web pages and a fair bit of interaction with the web application’s business logic. A well-known example of web applications that use sessions for client state management are e-commerce sites and their use of virtual shopping carts, e.g., the user can access his/her shopping cart at any time during the session. Apparently, this requires from the application to maintain a state, which can be achieved in four ways, viz.

1. by placing all state values in cookies (see above),
2. by playing a unique key in the cookie to uniquely identify a user’s session (see above),

3. by including necessary state values as parameters in every URL of the web application, and

4. by including a unique key as a parameter in every URL of the web application.

Unfortunately, each of these approaches has some trade-off. In case of storing state values in a cookie, the application developer is restricted by the size limits of a cookie. Further, the security issues as discussed above in case of cookies further devaluate this technique of storing state values. Another drawback is that most browsers on the grounds of their security settings prohibit locally storing cookies. Similarly, also using unique keys suffers from the client’s willingness to locally store cookies. Notwithstanding, also the circumstance that a cookie may be stolen (see above) devaluates this technique of using unique keys as session identifiers. Further, if an attacker manages to reveal the structure of the keys that are used for client state management, using brute force attacks, other user’s session keys can be guessed. This again would result in spoofing authentication and follow-up attacks. Another drawback of using such unique keys are storage requirements w.r.t. to keeping the issued unique keys in memory. Thus, a typical session duration of 15 minutes is fixed after which a key is discarded at the server side. In case of URL redirection, i.e., including state values in the URLs of the web application, each URL is generated dynamically to include necessary parameters that reflect either the entire session state or contain a unique key. Alas, such URL parameters open up dangerous attack vectors for an attacker (see Section 2.1.4.1 on SQLI and XSS attacks).

### 2.1.4 Security Vulnerabilities and Attacks

In this section we introduce key concepts in non-functional security testing. Thereafter, we illustratively discuss SQLI and XSS attacks (Section 2.1.4.1). As there exists no common source of knowledge for the definitions we give, we have collected them from various sources (see below), yet they are in-line with each other.

In order to design a non-functional security testing system, it is necessary to understand the types of threats and attacks that can be mounted against a software system and how these threats may manifest themselves. Further, it is also important to understand threats and their origins from the viewpoint of identifying the vulnerabilities that enable a threat to occur. Upon this relationship, non-functional security testing then can be efficiently done.

To assist the reader, the following definitions are used in this thesis:

**Vulnerability** A vulnerability denotes a weakness in an information system, system security procedure, internal control, or implementation that could be exploited by a threat source.
Security Requirement  A security requirement is a requirement w.r.t. the security aspects of a software system. We distinguish between two types of security requirement, viz.:

- **Positive Requirement** A positive requirement stipulates that the system must do a certain thing [SK07]. It can be thought of as “the software shall do X” [Voa08].

- **Negative Requirement** A negative requirement insists on the system not doing something [SK07]. It can be thought of as “the software shall not do X” [Voa08]. Observe that such a negative requirement represents a security property rather than functionality.

Remember that *positive* requirements are testable, whereas *negative* requirements for the most part are not. This is due to that a positive requirement eventually describes a system functionality whereas a negative requirement states a property the system must possess, especially in face of an attack or misuse.

Software Requirements Specification  A software requirements specification (or just specification) is a specification for a particular software product, program, or set of programs that do certain things. It must correctly define all of the software requirements, but no more [83084].

Threat  A threat denotes any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, individuals or other organizations through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service [Com04].

Risk  A risk is a measure of the extent to which an entity is threatened by a potential circumstance or event, and typically a function of (i) the adverse impacts that would arise if the circumstance or event occurs and (ii) the likelihood of occurrence [Com04].

Attack  An attack comprises any kind of malicious activity that attempts to collect, disrupt, deny, degrade, or destroy information system resources or the information, comprised by those systems, itself [Com04], i.e., a plan to carry out a threat.

Attack Vector  An attack vector is the method used to gain access to a system to deliver a malicious payload [TTA05].

Exploit  The act of attacking a software system by utilizing a vulnerability is called an exploit. This comprises any piece of software, chunk of data, or sequence of commands that takes advantage of the vulnerability to alter the behavior of the targeted software system [DMS06].
**Attack Goal**  An attack goal eventually states the motivation for performing an exploit. In this dissertation and, in particular for our method as introduced in Chapter 3, we have identified four major attack goals as listed below:

- *Authentication violation*, if the goal of an attack is to fool the process of verifying the identity or other attributes claimed by or assumed of an entity (user, process, or device), or to verify the source and integrity of data [Com04].

- *Leakage*, if the goal of an attack is the unauthorized transmission of data (or information) from within an organization to an external destination or recipient [Gor07].

- *Tampering*, if the goal of an attack is to intentionally trigger an event resulting in modification of a system, its intended behavior, or data [Com04].

- *Denial–of–Service (DoS)*, if the goal of an attack is to cause the prevention of authorized access to resources or the delaying of time-critical operations (where time-critical may be milliseconds or hours, depending upon the service provided) [Com04].

### 2.1.4.1 SQLI and XSS Attacks

Following we illustratively discuss SQLI and XSS attacks and the necessary prerequisites. We first present the three most common variations of SQLI, followed by two variations of XSS.

**SQLI**  An SQLI vulnerability is the result of an application that consumes user input that is destined to be used in constructing database statements without sanitizing this very input first. Thus, an attacker can submit (partial) SQL statements as input which are then executed by the database. This allows an attacker to, among others, bypass authentication or, extract from or alter data in the database (i.e., tampering or leakage). To counter this class of attacks, any user input should be sanitized or rejected, if it appears malicious. Further, using parameterized queries instead of on-the-fly constructing database queries from user input mitigates a broad range of SQLI vulnerabilities.

**SQLI by signature evasion**  is the easiest and most effective kind of SQLI. Basically, what one needs is a web browser and a web application, allowing a user to input data, which is then further processed by a database in the application’s back-end.

As shown in Figure 2.9 in line 5, an SQL query is constructed from a string of text, which embeds input, specified by a user of the application by the parameters **uname** and **pword**. If all goes well, i.e., the user provides valid data (e.g., **uname** = *user* and **pword** = *pa55*), the resulting query is of the form
2.1 Foundations

```php
if (isset($_GET['login'])) {
    $uname = $_GET['uname'];
    $pword = $_GET['pword'];

    $query = "SELECT * FROM user WHERE username = '${uname}' AND password = '${pword}"
    $result = mysql_query($query);
    $num_rows = mysql_num_rows($result);
    ...
```

Figure 2.9: PHP code vulnerable to SQLI. Using a properly crafted input string, an attacker can successfully evade the programmer’s intended SQL query and perform an SQLI by signature evasion.

SELECT * FROM user WHERE username = 'user' AND password = 'pa55'.

Executing this query against the database returns, depending on whether an entry for the user exists in the table user, a result set, which contains as only object the user with user name user and password pa55, and its associated data. If no entry can be found, the database returns an empty result set. However, this does not mean that without such a valid user name/password combination, we cannot retrieve any data from the database.

The first thing to do in an attempt to exploit an SQLI vulnerability is to verify its existence. This can be achieved quite easily by using the most subtle injection string, viz. ‘‘ (a plain apostrophe). Providing the apostrophe as input for uname and leaving pword blank, results in a query of the form

SELECT * FROM user WHERE username = ' OR 1 = 1--

By this, an attacker evades WHERE username = ' OR 1 = 1--. The reason for the error is that the additional apostrophe closes the programmer’s first apostrophe too early, invalidating the whole query, as in the end, there is one apostrophe without a closing one extant. With this knowledge, an attacker then can manage to evade the programmer’s SQL query by using an input at the first position an apostrophe and subsequently his crafted input string, terminating with a comment to cut off the remaining query. In case of our code snippet from Figure 2.9 this means, that an attacker can input, e.g., ' OR '1' = '1'-- resulting in a query of the form

SELECT * FROM user WHERE username = ' OR '1' = '1'--.

It may not always be the case that an SQLI can be verified this easily, however, it is a good approach, which often yields in success.
it simply returns all entries of the table user. Although, there may not exist a user with an empty user name, still one equals one for all users. Thus, an attacker successfully has extracted data from the database without the use of a valid user name/password combination.

**Blind SQLI** is a little trickier than SQLI by signature evasion. In this case, the chicanery is to inject conditional (logical) statements, i.e., to ask a database true or false questions. This type of SQLI is valuable if the result of an attack manifests itself by the application to display entirely different content as of the results of the conditional statement. Common injection strings for this attack are, e.g., '1' OR '1'='1' in conjunction with '1' AND '1'='2'. Consider for example the vulnerable PHP code fragment from Figure 2.10.

```php
if (isset($_GET['topicid'])) {
    $my_topicid = $_GET['topicid'];

    $sqlstmt = "SELECT msg FROM messages WHERE topicid = \${my_topicid}";
    $result = mysql_query($sqlstmt);
    while($row = mysql_fetch_assoc($result)) {
        echo "Message " . $row['msg'];
    }
    ...
}
```

**Figure 2.10:** PHP code vulnerable to SQLI. Using a properly crafted input string, an attacker can successfully verify and exploit a blind SQLI by triggering different application responses.

With the former injection strings the goal is to cause the application to display differently by injecting both strings in subsequent database queries, e.g.,

\[
\text{SELECT msg FROM messages WHERE topicid}= '1' \text{ OR '1'=}'1',
\]

and

\[
\text{SELECT msg FROM messages WHERE topicid}= '1' \text{ AND '1'=}'2',
\]

Obviously, in the first case the resulting database query would always result to true, and the application thus would display all topics, whereas in the latter case it would return nothing (remember that one never equals two).

Another approach in blind SQLI is to use timing queries, e.g., '1' or `SLEEP(10)` that triggers the database to interrupt execution of the query for 10 seconds, which obviously leads in a measurable delay\(^3\). Such an injection string, which gets the database off to sleep, can even be used if the results of an attack do not even trigger the application to display differently.

---

\(^3\)Remember that `SLEEP` only works for MySQL databases. In case of, e.g., MsSQL one instead uses `WAITFOR DELAY`.
SQLI by incorrect type handling is caused if user input is not strongly typed or type checked. Such attacks usually are possible if the application, e.g., requires a numeric input but the programmer misses to check the proper type, e.g., numeric, of user provided input prior to sending it to the database. Consider for example the vulnerable PHP code fragment from Figure 2.11. The vulnerable code fragment is almost identical to the one from Figure 2.10, except that in the constructed query, the parameter \$my_topicid is not surrounded by quotes. Thus, this query expects a numeric parameter. As can be seen in Figure 2.11, the application does not check the type of the value of the \$my_topicid parameter, thus it is vulnerable to SQLI by incorrect type handling. An attacker now could inject a string like 1;DROP TABLE messages resulting in the query

```
SELECT msg FROM messages WHERE topicid=1;DROP TABLE messages;
```

Upon executing this query, the database drops the table messages, i.e., the attacker has managed to delete data from the database. Apparently, this type of SQL attack could easily be mitigated by simply checking the type of a user’s input and subsequently reject it, if it is of wrong type.

An effective technique when exploiting an SQLI vulnerability, is to apply filter bypassing. As stated earlier, programmers should sanitize any user input, and if necessary, even reject it. This generally is accomplished by searching in the user’s input for SQL owned language constructs and remove/replace them. Contrary, an attacker must find ways of how to circumvent such filters by creating statements that seem safe. Luckily, most SQL databases offer a rich set of features and supported encoding schemes that allow an attacker to obfuscate his injection, e.g., instead of submitting '1' or '1'='1' an attacker could submit 1'/**/|1|**/CHAR(49)=51-ASCII(2). If a programmer’s filter looks for the keyword or the classic conditional statement '1'='1' it would fail, as the statement contains none of those in plain but obfuscated.
To conclude, we want to mention a few words regarding prevention of SQLI attacks. Generally, one should not trust any user input. Following this advice, a broad range of SQLI attacks could be prevented against easily by applying various sanitation techniques on any user input (e.g., remove apostrophes, use a white list for filtering out invalid input,...). Further, using parameterized queries mitigates a broad range of “trivial” SQLI vulnerabilities. Nevertheless, what should be kept in mind at this point is that establishing prevention mechanisms once does not suffice. Attackers usually are one step ahead, thus, an application developer should keep in mind to advance the set of prevention mechanisms perpetually.

We deliberately do not mention second order SQLI attack, as it is not among the attacks, our method and its tool implementation test for. This stems from that such attacks cannot be verified without the availability of the application’s source code, especially its database schema, to identify both, where the injected statements are stored in the database and, later on, where they are loaded again and executed by the application.

**XSS** An XSS vulnerability, similarly to SQLI vulnerabilities, results from an application that consumes user input without sanitizing this very input first. Contrary however to SQLI, XSS attacks do not target an application’s database but instead a victim’s browser. The idea of XSS attacks is to place malicious code, e.g., Javascript, somewhere in the application with the goal that, at a later time, it will be loaded into a victim’s browser and executed there. This allows an attacker to, among others, steal sessions to bypass authentication. To counter this class of attacks, any user input should be sanitized or rejected, if it appears malicious.

**First-order XSS** or Type 1, or reflected XSS, results due to an application which mirrors user input in the next HTML page it generates without sanitizing it. The trick of a Type 1 XSS attack is to get a victim to clicking a URL which leads to a page containing malicious JavaScript code. Such URLs usually can be delivered by various means, e.g., via e-mail. At the time the victim’s browser renders the page it executes this fragment of malicious JavaScript code. Type 1 XSS attacks, among others, result in web page defacement, stealing of session cookies or passwords, or keystroke logging. As an example, consider the vulnerable PHP code fragment in Figure 2.12. As is obvious from Figure 2.12, the `name` parameter introduces the vulnerability as its contents are not sanitized whatsoever. If an attacker crafts a URL where the `name` parameter contains the value

```html
```

and further, a victim clicks the resulting URL, then the Type 1 XSS vulnerability has been exploited and the victim’s cookie gets stolen and submitted to a server that is under the control of the attacker. To mitigate Type 1 XSS users should be
2.1 Foundations

Figure 2.12: PHP code vulnerable to Type 1 XSS. Using a properly crafted input string, an attacker can successfully exploit a Type 1 XSS vulnerability.

```php
if (isset($_GET['name'])) {
    $name = $_GET['name'];
    echo "Hello " . $name . "!";
    ...
```

more careful when clicking on any links and applications, as in case of SQLI, should always sanitize or reject user input prior to processing it.

**Second-Order XSS** or Type 2, or persistent, or stored XSS, results from an application which takes user input, stores it in a database or a file and later displays this very user input to other application's users, e.g., in a bulletin board. The trick of Type 2 XSS then is to get a malicious piece of JavaScript (see Type 1 XSS) being stored in a database or a file and assure that this JavaScript subsequently is executed in victims' browsers, if the affected HTML page gets rendered. The PHP code fragment in Figure 2.13 apparently is vulnerable to a Type 2 XSS attack because of the `msg` parameter. If an attacker manages to submit a malicious piece of JavaScript code, i.e.,

```html
```

that further is stored in a database, i.e., in the `msg` field of the `messages` table, it alter on will be loaded again and presented to a victim as part of the complete list

```php
if (isset($_GET['msg'])) {
    $msg = $_GET['msg'];
    $insert_stmt = "INSERT INTO messages (msg) VALUES ('$msg');";
    mysql_query($insert_stmt);
}
...
//at some later point
$load_stmt = "SELECT * from messages";
$result = mysql_query($load_stmt);
while($row = mysql_fetch_assoc($result)){
    echo "Message " . $row['msg'];
}
...
```

Figure 2.13: PHP code vulnerable to Type 1 XSS. Using a properly crafted input string, an attacker can successfully exploit a Type 1 XSS vulnerability.
of messages stored in the msg table. Upon execution in the victim’s browser, as in case of Type 1 XSS, the victim’s cookie gets stolen and subsequently is submitted to a server controlled by the attacker.

One of the main problems with Type 2 XSS attacks is their prevention. This stems from the fact that Type 2 XSS attacks rely on an SQL database or a file that can be modified by the attacker. Put another way, especially in case of a Type 2 XSS attack that abuses a database, to prevent it, an application must not only sanitize user input destined for being processed or stored in a database against SQLI attacks, but also against potential second-order XSS attacks. This makes it far more complex for a developer to establish a set of sound and effective input filters.

Another interesting aspect of Type 2 XSS attacks, compared to Type 1 XSS attacks, is their ease in execution, once an appropriate vulnerability has been found. As soon as the attacker manages to place his malicious script code in the database and assures that it is going to be executed in victims’ browsers, the job is done. Thus, in case of second-order XSS attacks, there is no need for social engineering, i.e., to get a victim to click a URL. Further, once the script is placed in the database, it gets executed in many victims’ browsers.

Obviously, obfuscation techniques as discussed in case of SQLI also apply for XSS attacks.

Prevention techniques against XSS attacks merely are the same as for SQLI. The rule of thumb again is to never trust any user input. In this sense, programmers should always sanitize any user input by, e.g., escaping Javascript specific code fragments or HTML tags, or apply a white list (as for example in case of SQLI) to reject suspicious user input. Further, user supplied data should only be inserted in allowed locations. Besides prevention mechanisms that are in place, one should never sit back and be satisfied. As already mentioned in case of SQLI, attackers usually are one step ahead, thus, an application developer should keep in mind to advance the set of prevention mechanisms perpetually.

We again deliberately avoided to mention the third kind of XSS attacks, viz. DOM-based XSS attacks. This is by virtue of its nature which requires access to the application’s source code for successfully performing it. Remember that our tool and its underlying method are black-box, thus, we cannot perform the necessary source code analysis.

\[2.1.5\] Security Testing

Security testing is a process to determine that an information system protects and maintains functionality as intended \[Com04\]. At this, security testing comprises the evaluation of the management, operational, and technical security controls to determine the extent to which the controls are implemented correctly, operating as intended, and producing the desired outcome w.r.t. meeting the security require-
ments for the system or enterprise [Com04]. Generally, these security requirements are formulated within the scope of the hereinafter listed security concepts [Com04]:

- **Confidentiality**, i.e., the property that information is not disclosed to system entities (e.g., user, process, or device) unless they have been authorized to access the information.

- **Integrity**, i.e., the property whereby an entity has not been modified in an unauthorized manner.

- **Authentication**, i.e., the process of verifying the identity or other attributes claimed by or assumed of an entity (e.g., user, process, or device) or to verify the source and integrity of data.

- **Authorization**, i.e., access privileges granted to a user, program, or process or the act of granting those privileges.

- **Availability**, i.e., ensuring timely and reliable access to and use of information.

- **Non-repudiation**, i.e., the assurance that the sender of information is provided with the proof of delivery and the recipient is provided with proof of the sender’s identity, so neither can later deny having processed the information.

In doing security testing, a security tester can decide to either focus on positive or negative requirements [TyYsYy10]. A positive requirement specifies a security functionality to be evaluated, e.g., “a user can only log in to a website with a valid user name/password combination”. For the latter, i.e., negative requirements, the situation is utterly different. Here, a security tester cannot simply evaluate a required functionality, but rather must think out-of-the-box to verify that a security property holds, e.g., “the login mechanism must not be eluded”. Thus, in validating negative requirements, a security tester has to mimic an attacker’s behavior and try to think about potentially hidden security vulnerabilities to verify that a negative requirement is violated, e.g., “one can login by bypassing the login mechanism using some kind of SQLI”. This disparate view is more evident from Figure 2.14.

The specification states what is required by a software system. The implementation however defines what is actually available, i.e., what was realized w.r.t. the specification. As can be seen from Figure 2.14, the implementation does not always meet the specification entirely. Worse, the implementation may erroneously realize hidden side-effect functionality. Thus, security testing requires a classification into functional and non-functional security testing, i.e., security testing which verifies positive requirements (light green shape) and, security testing, which searches for side-effect functionality, i.e., negative requirements that are not met, a major source for security bugs. Unfortunately, as mentioned earlier, coming up with such negative requirements for non-functional security testing is difficult [BM03, Voa08].
2.1 Foundations

Formally said, $\mathcal{S}$ is a set of specifications $s$ over a set $\mathcal{R}$ of requirements $r$, and the implementation $\mathcal{I}$ is a set of programs $p$ as defined next.

**Definition 2.1 (Requirements).** $\mathcal{R}$ is a set of requirements with elements $r_i$, i.e.,

$$\mathcal{R} = \{r_1, \ldots, r_n : r_i \in \mathcal{R}^+ \lor r_i \in \mathcal{R}^-\},$$

with

$$\mathcal{R}^+ = \{r_1^+, \ldots, r_n^+\}$$

and

$$\mathcal{R}^- = \{r_1^-, \ldots, r_n^-\}$$

and

$$\mathcal{R}^+ \cap \mathcal{R}^- = \emptyset,$$

where $\mathcal{R}^+$ is the set of positive requirements and $\mathcal{R}^-$ the set of negative requirements.

**Definition 2.2 (Specifications).** $\mathcal{S}$ is a set of specifications with elements $s_i$, i.e.,

$$\mathcal{S} = \left\{ s_i \in (\wp(\mathcal{R}) \setminus \emptyset) : \bigcap_{s_i \in \wp(\mathcal{R}) \setminus \emptyset} s_i = \emptyset \right\}.$$

The elements $s$ of $\mathcal{S}$ are disjoint subsets of the power set of $\mathcal{R}$ ($\wp(\mathcal{R})$). Thus each $s \in \mathcal{S}$ covers a distinct set of requirements $r \in \mathcal{R}$. We deliberately remove the
empty set $\emptyset$ from $\emptyset (\mathcal{R})$ as an empty specification makes no sense. At first, defining $\mathcal{S}$ as a set of specifications seems off the wall, however, nowadays software systems do not consist anymore of just one program, but rather large sets of programs, each with its own specification.

**Definition 2.3 (Implementation).** $\mathcal{I}$ is a set of programs with elements $p_i$, i.e.,

$$\mathcal{I} = \{p_1, \ldots, p_n\}.$$ 

The realization of a specification $s$ by a program $p$ is denoted as $p \triangleright s$ (read “$p$ implements $s$”) with $\triangleright \subseteq \mathcal{I} \times (\mathcal{S} \cup \emptyset)$. We need the empty set $\emptyset$ to allow programs $p$, not implementing any specification $s$, i.e., programs with pure side-effect functionality. The description of a program $p$ by a specification $s$ is denoted $s \triangleright s p$ (read “$s$ describes $p$”) with $\triangleright \subseteq \mathcal{S} \times \mathcal{I}$. Observe that for $\triangleright$, we do not need the empty set, as an empty specification does not describe anything. Further, observe that $\triangleright \implies \triangleright$, but not the other way round. It is not always the case that a program implements the given specification.

We next describe the set of implemented specifications $\mathcal{S}\triangleright$ and the set of implemented programs $\mathcal{I}\triangleright$ under $\mathcal{S}$.

**Definition 2.4 (Implemented specifications).** $\mathcal{S}\triangleright$ denotes the set of implemented specifications with elements $s_i$, i.e.,

$$\mathcal{S}\triangleright = \{s \in \mathcal{S} : \exists p \in \mathcal{I} : (p, s) \in \triangleright \land (s, p) \in \triangleright\}.$$ 

Observe that the latter constraint, i.e., $(s, p) \in \triangleright$, assures that the proper specification is implemented.

**Definition 2.5 (Implemented programs).** $\mathcal{I}\triangleright$ denotes the set of implemented programs w.r.t. a set $\mathcal{S}$ of specifications with elements $p_i$, i.e.,

$$\mathcal{I}\triangleright = \{p \in \mathcal{I} : \exists s \in \mathcal{S} : (p, s) \in \triangleright \land (s, p) \in \triangleright\}.$$ 

Again, observe that the latter constraint, i.e., $(s, p) \in \triangleright$, assures that the proper specification is implemented. $\mathcal{S}\triangleright$ then contains all the specifications $s$ which are an element of $\mathcal{S}$ and part of $\triangleright$ and $\triangleright$. The definition of $\mathcal{I}\triangleright$ is analogous, yet elements $p$ of $\mathcal{I}$ instead of $s$ of $\mathcal{S}$ constitute the set.

We further define the set of missing specifications $\mathcal{S}_m$, i.e., what was missed to be implemented, and the set of programs with side-effect functionality $\mathcal{I}_{se}$, i.e., programs which do not implement a mandatory negative requirement w.r.t. $\triangleright$. 
**Definition 2.6** (Missing specifications). \( S_m \) denotes the set of specifications missed to be implemented with elements \( s_i \), i.e.,

\[
S_m = S \setminus S^I.
\]

**Definition 2.7** (Side-effect programs). \( I_{se} \) denotes the set of programs with side-effect functionality with elements \( p_i \), i.e.,

\[
I_{se} = \{ p \in I : \exists s \in S_m : \exists r \in R^- : r \in s \land (s, p) \in \delta \}.
\]

Hence, \( S^I \) describes the set of implemented specifications, functional security testing should at least verify and \( I_{se} \) the set of programs \( p \) with potential side-effect functionality; non-functional security testing should reveal. Ideally, functional security testing additionally establishes \( S_m \). The value of \( I_{se} \) initially is not known, i.e., \( I_{se} = \emptyset \); thus, \( I_{se} \) is theoretic and used to stress security testing’s relatedness to revealing programs \( p \) with side effect functionality (ideally by exploring the space of negative requirements).

Finally, for the sake of completeness, we define the set of missing requirements \( R_m \).

**Definition 2.8** (Missing requirements). \( R_m \) denotes the set of requirements missed to be implemented with elements \( r_i \), i.e.,

\[
R_m = \{ r \in R : \exists s \in S_m : r \in s \}.
\]

Using these definitions, we next discuss functional and non-functional security testing.

### 2.1.5.1 Functional Security Testing

Functional security testing is meant to test whether a software system behaves as specified \([MR05]\), whereat positive requirements specify the intended behavior. Thus, positive requirements generally should be used as a basis for deriving test cases in functional security testing. Doing so allows both to verify whether the implementation meets the specification, but also to what extent the specification was not implemented, i.e., missing functionality. This circumstance becomes clear from Figure 2.14. Again, the specification states what is required and the implementation what is available. The intersecting region, i.e., the *green* shape \( (S^I) \), depicts specified and implemented functionality, hence directly falls into the scope of functional security testing.
Consider Figure 2.15 for an illustrative example of functional security testing. Figure 2.15 depicts a trivial software system, where two processes A and B communicate via a buffer, where process A only has the permission to write to, and process B only the permission to read from the buffer.

Hence, we can infer that $R$ contains two positive requirements, viz.:

- $r_1^+$ - Process A has permission to write to the buffer and
- $r_2^+$ - Process B has permission to read from the buffer,

i.e., $S = \{s_1, s_2\}$, with $s_1 = \{r_1^+\}$ and $s_2 = \{r_2^+\}$. Next, we assume that $I$ contains two known programs, viz. $p_1$ and $p_2$ with

- $p_1$ - Realizing write access to the buffer for process A and
- $p_2$ - Realizing read access from the buffer for process B,

with $\delta = \{(s_1, p_1), (s_2, p_2)\}$ and $\iota = \{(p_1, s_1), (p_2, s_2)\}$. By this, we then can calculate the least scope for functional security testing using Definition 2.4, i.e., $S^I = \{s_1, s_2\}$.

More informally, for the software system described in Figure 2.15, functional security testing should at least cover the two positive requirements $r_1^+$ and $r_2^+$ of specifications $s_1$ and $s_2$, to verify that implementation $I$ meets all specifications $s \in S^I$.

### 2.1.5.2 Non-functional Security Testing

Contrary to functional security testing, non-functional security testing is meant to test whether a software system behaves in an unexpected way, i.e., provides hidden side-effect functionality. Non-functional security testing follows Dahl’s idea that program testing can be used to show the presence of bugs, but never to show their absence [DDH72]. Opposed to this, functional security testing not directly attempts

---

4 We do not know about hidden side-effect functionality at this point.

5 For the sake of convenience, $S_m = \emptyset$. 

---
to show the presence of bugs, but rather that a specification has been implemented properly. This is further illustrated by Figure 2.14. Consider again the green shape, i.e., $S^I$, which defines the least scope of functional security testing. In an ideal world, the green shape coincides with the specification, i.e., the implementation meets all specifications and has no side-effect functionality. However, as this, for the most part is not the case, there are elements of the specification, which were missed (blue regions), i.e., $S_m$ and further, parts of the implementation, which are not required (red regions), i.e., $I_{se}$, the scope for non-functional security testing. Unfortunately, this set initially is unknown. It is therefore the purpose of non-functional security testing to detect any program $p$ with side-effect functionality, which thus is a member of the set $I_{se}$.

Consider Figure 2.16 for an illustrative example. Figure 2.16 depicts the same trivial software system as Figure 2.15, i.e., two processes $A$ and $B$, which communicate via a buffer. However, this time we want to verify whether the application violates the specification as shown in Figure 2.15, i.e., process $A$ illicitly can read from and, process $B$ can write to the buffer. Thus, we define two negative requirements prohibiting such misbehavior, e.g.,

$r_1^-$ - Process $A$ cannot read from the buffer and

$r_2^-$ - Process $B$ cannot write to the buffer,

i.e., $S = \{s_3, s_4\}$ with $s_3 = \{r_1^-\}$, $s_4 = \{r_2^-\}$.

We thus define the basis for further deriving test cases for non-functional security testing. Obviously, these two requirements, viz. $r_1^-$ and $r_2^-$, are not the only two negative requirements, e.g., there could also be a third one that states that “no other process $C$ can read or write to the buffer”.

This does not mean that functional security testing does not reveal side effects. Yet, its primary focus is on verifying an implementation under a specification.
Suppose now further that any of the programs in \( I \) (\( I \) is the same as in case of our example for functional security testing, i.e., \( I = \{p_1, p_2\} \)) erroneously realizes side-effect functionality, e.g.,

\[ p_1 - \text{Realizing read access from the buffer for process A} \]
\[ p_2 - \text{Realizing write access to the buffer for process B}. \]

with \( \delta = \{(s_3, p_1), (s_4, p_2)\} \) but \( \iota = \emptyset \) (remember that both programs have side-effect functionality that is prohibited by the negative requirements \( r_1^- \) and \( r_2^- \)). Consequently, \( I_{se} = \{p_1, p_2\} \). With this, we have established the set of programs \( p \), which implement hidden side-effect functionality.

More informally, for the software system described and specified by Figures 2.15 and 2.16, non-functional security testing should address the negative requirements \( r_1^- \) and \( r_2^- \) of specifications \( s_3 \) and \( s_4 \), to reveal that programs \( p_1 \) and \( p_2 \) have implemented hidden side-effect functionality, i.e., \( r_1^- \) and \( r_2^- \) are violated. This is achieved by negating those two requirements, i.e., \( \neg r_1^- \) and \( \neg r_2^- \) to then obtain testing scenarios that may violate a negative requirement. For example, consider again \( r_1^- \) stating that “Process A cannot read from the buffer”. Negating \( r_1^- \) yields that “Process A can read from the buffer”. Thus, we have inferred a testing scenario that would violate \( r_1^- \). Similarly, we can derive a testing scenario that violates \( r_2^- \).

Recently, techniques from model-based testing have been applied for security testing \cite{SGS12}. Thus, model-based security testing approaches benefit from model-based testing in various aspects, e.g., a high degree of automation, potential early detection of software bugs already at the design level, and high coverage of the SUT by the resulting high quality test cases \cite{UL10}. These advantages clearly justify the additional effort of test model design and maintenance. We discuss model-based security testing hereafter.

2.1.5.3 Model-based Security Testing

Model-based testing (MBT) is the application of explicit models for software testing. Manually selected algorithms automatically and systematically generate test cases from a set of models of the system under test or its environment \cite{Sch12}. Figure 2.17 illustrates the common setting for MBT; it comprises three steps, viz.

1. **System or Environment Modeling** The definition of system or environment models is the basis for later test generation. System models provide a specification \( S \) of the SUT either by its architecture and behavior, whereas environment models state the, obviously, production environment of the SUT and its interactions with other components.

2. **Test Requirements Definition and Test Generation** Prior to test generation, test requirements need to be defined. Test requirements are inevitable for test
Figure 2.17: Model-based testing. MBT is the successful application of models for software testing and algorithms to generate test cases.

3. Test Execution and Evaluation Test execution either can be done online or offline. For the former, i.e., online test execution, as just mentioned, test generation and execution are combined in an MBT tool, i.e., test cases are executed during designing. For the latter, i.e., offline test execution, test cases are executed in an external platform. Test evaluation is done using test oracles which are mechanisms that allow to determine whether a test case passed or failed, i.e., whether the system under test behaves as specified or misbehaves. Evaluation itself is done either by the MBT tool (for online test execution) or by an external platform (for offline test execution).
In 2006 Utting et al. introduced a taxonomy for model-based testing (Figure 2.18) \cite{UPL06}. Their taxonomy considers seven dimensions according which MBT approaches can be classified, as briefly discussed subsequently:

- **Model Subject** This dimension is the subject of the model, i.e., the intended behavior of the SUT or the possible behavior of the SUT’s environment. The model of the SUT has two purposes, (i) provide an oracle for the SUT by encoding intended behavior, and (ii) aid in test generation by exploiting its structure. Contrary, the model of the environment is used to restrict the possible inputs to the model of the SUT, i.e., it restricts the possible set of behaviors and thus acts as a test selection criterion.

- **Model Redundancy** MBT can be instantiated to different variants, where those scenarios differ in the level redundancy between modeling for testing and/or implementation. Figure 2.19 shows six variants w.r.t. \cite{Sch07}.

- **Model Characteristics** Model characteristics relate to different characteristics of the model as shown in Figure 2.18, i.e., (non–) determinism, the incorporation of timing issues, and a continuous or event-driven nature of the model.

Figure 2.18: Model-based testing taxonomy according to Utting et. al \cite{UPL06}.
• **Model Paradigm** The purpose of this dimension is to specify the paradigm and notation that are used to describe the model. These paradigms and notations are crucial in that they both, settle how fine-grained behavior of the system can be modeled and, as a consequence, thus also settle test generation.

• **Test Selection Criteria** A test selection criterion states which facilities are used to control generation of tests. Observer that for this there exists nothing like an “best” criterion. It is thus the duty of the test engineer to choose adequate test selection criteria and test case specifications.

• **Test Generation Technology** This sixth dimension is the technology that is used for test generation. This is either done manually, i.e., by graphically designing test cases, or automatically by exploiting the model of the SUT together with necessary test case specifications.

• **On/Offline Test Execution** Finally, MBT approaches also can be classified according to the test execution paradigm, which is either offline or online as already discussed earlier for Figure 2.17.

Traditionally, models are not only used for testing purposes only but also for the development of the SUT (see Model Redundancy above). Thus, there is a strong relation between system and test models given by the nature of MBT. This strong relation also is the reason for one of the key attractors of MBT, i.e., MBT’s capability to detect software flaws already early at the design level [UL10]. Based on this relation between system and test models, six variants of MBT can be identified [Sch07] as shown in Figure 2.19 and described below.

**System model–driven** approaches (Figure 2.19, variant (a)) solely rely on a system model for both, system and test generation. Consequently, the system and generated tests are not independent. A major drawback of this variant is that faults in the system model cannot be recognized by the tests.

**Test model–driven** approaches use dedicated test models for test generation (Figure 2.19, variant (b)) and system development/generation (Figure 2.19, variant (c)). Contrary to variant (a), tests are now decoupled from a system model, and hence (if a system model is present) be able to detect faults in it.

**System and test model–driven** approaches are the most systematic MBT approaches. In this case two distinct models are used, viz. a system and a test model. An advantage of such approaches is that consistency between models can be checked already at the design level (i.e., before an implementation exists). This in turn yields high quality test cases which results in a higher system quality as of more efficient testing. Approaches of this type are either system model–driven
(Figure 2.19, variant (d)), test model-driven (Figure 2.19, variant (e)) or independent (Figure 2.19, variant (f)). W.r.t. [PP05], variant (f) implements an optimal MBT approach.

An important role in MBT is occupied by (i) modeling languages, i.e., the formal notions in which the system and/or environment and its behavior, as well as the tests and their behavior are described, and (ii) model transformations that allow to infer a test or system model from its counterpart, i.e., the system or test model, respectively.

**Modeling Languages** Modeling languages are formal languages that are used to express software systems in a structure that is defined by a consistent set of formal
2.1 Foundations

rules (i.e., the metamodel). Besides specifying the syntax (i.e., the notation) of a modeling language, these rules are further used for interpretation of the meaning of language elements, i.e., the rules also define the semantics (i.e., the meaning) of the language. The purpose of modeling languages is manifold, most important among them is to aid in

- understanding interesting characteristics of an existing or desired system and its environment,
- predicting interesting characteristics of a system by analyzing its models
- communicating the design intent to others, and
- specify the implementation of a system, i.e., the model is a blue print for a software system.

Apparently this requires from modeling languages a high degree of expressiveness by its semantics and at the same time an easy to understand and learn syntax. Thus, modeling languages must follow certain design criteria for being useful in software engineering, viz. being

- purposeful, i.e., constructed to address a specific set of concerns,
- abstract, i.e., emphasizing important aspects while removing irrelevant ones,
- understandable, i.e., expressed in a form that is easily understandable by humans
- accurate, i.e., properly represent the modeled system,
- predictive, i.e., provide means to answer question on the modeled system, and
- cost effective, i.e., it should be much cheaper to model the desired system compared to actually implement it.

Especially the understandability of a modeling language has led to two contrary design strains for modeling languages, viz. graphical and textual modeling languages as discussed in the following.

**Graphical Modeling Languages** A graphical modeling language is a formal language that uses a diagram technique with named symbols. These named symbols represent syntactical elements of the language with specific semantics. Using lines as further syntactical elements, named symbols can be connected to form more complex “expressions” in such a graphical language. Generally, further graphical notations, e.g., lines with arrowheads, are used to specify constraints among the named symbols. The most prominent representative of a graphical modeling language is UML.
These days, UML has grown to a general purpose modeling language, i.e., a modeling language that enables to model different kinds of software systems w.r.t. different aspects of the modeled system, i.e., its static structure (by a class diagram), its internal behavior (by an activity diagram) or the different states the system may possess during execution (by a state-machine diagram). This usage as a general purpose modeling language however introduced modeling difficulties and inconsistencies w.r.t. different system types as UML in fact originally was designed only for object-oriented systems. Consequently, it required modifications to become more general for modeling specifics of different system types which was resolved by extending UML for other system types using its profiling mechanism, e.g., for service centric system by SoaML [OMG12c] or embedded systems by SysML [OMG12d]. Further, UML also has been extended for MBT by the UML testing profile (UMLTP) [OMG13].

**Textual Modeling Languages** Contrary to graphical modeling languages, textual modeling languages use a set of defined keywords accompanied with parameters or plain natural language terms and phrases to specify expressions that are both, readable by a human and interpretable by a computer. The most prominent representatives of such textual modeling languages are so-called domain specific languages (DSL)\(^7\). Contrary to, e.g., UML, which is a general purpose modeling language, a DSL is a language tailored to a specific purpose. In this sense, it generally cannot be used for different purposes but just one single purpose. From a more technical point of view, a DSL can also be seen as a small programming language designed specifically to express solutions to problems in a certain domain. Using both views of a DSL, i.e., as a modeling and/or a programming language, two types of DSLs can be distinguished, viz.

- **stand-alone DSLs** which are independent, i.e., they do not represent extensions to some host language (see below) but rather can be implemented in any host language that meets its requirements. Well known representatives of stand-alone DSLs are SQL or \LaTeX\(\text{(which was used to typeset this thesis).}\)

- **embedded DSLs**, which are also known as fluent interfaces, require some host language whose concepts it can reuse to design a more specific language. Consequently, an embedded DSL cannot live without its host language. Examples for such DSLs are the UML testing profile [OMG13] or any kind of API\(^8\) that aims to provide for more readable code.

**Model Transformations** A model transformation can be thought of as a program that takes as an input a source model that conforms to a given metamodel and
produces as output another model, i.e., a target model that also conforms to a given metamodel. Generally, the metamodel of the input specifies which models are allowed as input whereas the metamodel of the output specifies the constraints a target model must meet. From a more formal point of view, a model transformation can also be seen as an automated way to ensure that a family of models is consistent w.r.t. constraints as defined by a software engineer. Figure 2.20 depicts the generic model transformation process. As can be seen, the transformation is based on a mapping between the source and the target model. This mapping specifies the relations between language elements of the source and target model, i.e., it specifies how to transform language elements of the source model into language elements of the target model.

![Figure 2.20: Generic view of the model transformation process.](image)

W.r.t. to the taxonomy of Mens and van Gorp [MVG06], model transformations can be classified along four dimensions, viz.:

- **Number of source and target models**, i.e., the number of input and output models that are involved in a model transformation. An example of a transformation that includes multiple input models is model merging where multiple source models are transformed into one target model. An example for the latter would be the generation of multiple platform specific models (i.e., the target models) from one platform independent model (i.e., the source model).

- **Endogenous vs. exogenous transformations** addresses the languages in which models are expressed. Endogenous transformations are transformations between source and target models expressed in the same language, whereas, exogenous transformations are model transformations between source and target models expressed using different languages. An example for an endogenous transformation is refactoring, i.e., a change to the internal structure of the modeled system. An example for an exogenous transformation is code generation.

- **Horizontal vs. vertical transformations** considers the level of abstraction of the source and target model. If the source and target models have different levels of abstraction, the underlying model transformation is a vertical one.
Contrary, if both models, i.e., the source and the target model reside at the same level of abstraction, the underlying model transformation is a horizontal one.

- **Syntactical vs. semantical transformations** finally considers whether a model transformation only transforms the syntax or also takes into account the semantics of the model. Obviously, the latter is more complicated. An example of a syntactical transformation is a parser that transforms some program into its abstract syntax tree. Upon these, further and more complicated semantical transformations can be applied.

A further important aspect that is not considered by the taxonomy of Mens and Van Gorp is the definition of the processing strategy that is applied on the source model during an exogenous model transformation. In their classification of model transformation approaches, Czarnecki and Hensen [CH03] also consider this dimension, i.e., *visitor*– and *template*–based approaches. In case of the former, i.e., *visitor*–based, the entire internal representation of the source model is traversed, whereas in case of the latter, i.e., *template*–based, only specific fragments of the source model (for which templates exist) are processed. Observe that a template–based approach more closely resembles the target model.

Apart from these four dimensions according to which model transformations can be classified, Mens and van Gorp further mention three important characteristics of a model transformation, viz.:

- **Level of automation** considers in how far a model transformation can be automated or requires manual intervention. An example for the latter is the transformation of a requirements document to an analysis model.

- **Preservation** considers which aspects of a source model are preserved during transformation into a target model. For example, in case of an endogenous transformation, e.g., a refactoring, the internal behavior of a model is well preserved whereas its structure changes.

- **Complexity of the transformation** considers whether a transformation is small, e.g., a refactoring, or substantially more heavy-duty, e.g., code generation.

**Model-based security testing** (MBST) is the successful application of MBT for security testing. For this, plain system or environment models are not sufficient anymore, but instead, security upholstered models like threat, fault and risk models, or weakness and vulnerability models are further required for generating effective security test cases [SGS12]. Figure 2.21 shows the adapted setting from MBT for MBST that incorporates security models.

**Threat, Fault and Risk Models** Contrary to system or environment models the purpose of threat, fault and risk models is to specify what can go wrong, i.e., causes
Figure 2.21: Model-based security testing. MBST is the concretization of MBT for security testing. It supports both, functional and non-functional security testing.

for and consequences of attacks. Additionally, they introduce means to prioritize resulting test cases by occurrence probabilities and potential impacts.

**Weakness and Vulnerability Models** Whereas threat, fault and risk models describe what can go wrong, weakness and vulnerability models aim at describing a weaknesses or a vulnerability itself. Necessary information is available from databases like, e.g., the National Vulnerability Database (NVD) [NIS14] or the Common Vulnerabilities and Exposures (CVE) [MIT14] database.

Especially during test generation, threat, fault or risk models are to be considered for test identification [SGS12]. Apart from that, MBST follows the same steps as classic MBT. Figure 2.21 illustrates the adapted setting for MBST as against MBT.

Obviously, the classification of Utting et al. (Figure 2.18) is not applicable anymore for MBST. However, we have already provided a classification for MBST in Chapter 1 (Figure 1.5) which is why we do not discuss it again at this point. Apart from that, the topics discussed above in case of MBT, viz. modeling languages and model transformations, also apply to MBST.
2.2 State-of-the-Art

In this section we review state-of-the-art that is relevant to topics in this dissertation. We start by reviewing the application of logic programming in testing (Section 2.2.1), followed by knowledge engineering in security engineering (Section 2.2.2). We then discuss related work in the area of security testing (Section 2.2.3), viz. penetration testing (Section 2.2.3.1), MBST (Section 2.2.3.2) and web application security testing (Section 2.2.3.3).

2.2.1 Logic Programming in Testing

In testing, logic programming has primarily been applied for two purposes, viz. test data generation and test case generation. For test data generation, constraint solving \textsuperscript{GBR98} together with either symbolic execution \textsuperscript{Meu01} or feasible path analysis \textsuperscript{JBW+94} have been harnessed.

In case of test case generation, existing approaches, which use logic programming, mainly build on constraint solving techniques. Ranga et al. \textsuperscript{VK95} present a method for generating design tests using path enumeration and constraint programming for VHDL programs. Using annotated control-flow graphs, paths are selected for which then constraints corresponding to the statements along the path are generated. Solving the constraints yields in design test specifications. Denney \textsuperscript{Den91} suggests a method for generating test cases from Prolog based specifications. A custom metainterpreter monitors and controls the execution of programs using specified paths. This then allows the generation of tests for specification-based testing. Bieker and Marwedel \textsuperscript{BM95} investigated retargetable self-test program generation for embedded processors. Their method works by matching test patterns on to a hardware description of a processor. In this manner, using constraint logic programming, their method thus generates executable test cases by self-test programs. Gómez-Zamalloa et al. \textsuperscript{GZAP10} suggest to use constraint logic programming as a symbolic execution mechanism to generate test cases for object-oriented programs. Using a declarative notation of the input program their method generates test cases according to a given coverage criterion (e.g., path or statement coverage). Lötzbeyer and Pretschner \textsuperscript{LP00} introduced an approach for testing executable system specifications (system models) of reactive systems. Their method translates such system models into constraint logic programs, which then are executed w.r.t. a predefined constraint to produce meaningful test sequences for specification-based testing. The work of Caballero et al. \textsuperscript{CGRSP10}, in contrast to the above, focuses on a specific language, viz., SQL. Given a database schema, their method generates a set of domain constraints, which, when solved, represent test database instances. These database instances allow to verify the correctness of, e.g., correlated SQL queries. Finally, the work of Gorlick and Kesselman \textsuperscript{GKMSP90} describes a complete testing methodology
2.2 State-of-the-Art

for message protocol testing. For this, their method employs both, a context-free message grammar (the specification) and a constraint system to either generate or verify messages (test cases).

In security testing in particular, the application of logic programming is rather humble. So far, existing work focuses on protocol verification. Blanchet [Bla01, Bla03] and Abadi and Blanchet [AB05] present an approach to automated verification of cryptographic protocols. Protocols are specified as logic programs to then check via resolution, if certain security properties, e.g., secrecy or authenticity, hold or are violated by a potential attack. Aiello and Massacci [CAM01] introduce a method to search for attacks against protocols using planning. Protocol traces are considered as plans to achieve goals, whereas the goals are the attacks (or security violations). Thus, the potential existence of an attack is modeled into the declarative specification of a protocol, and given that this resulting specification is satisfiable, i.e., the goal is achievable, an attack has been verified. Alberti et al. [ACG+06] consider the application of abductive logic programming for security protocol verification. Protocols are specified using atoms and logical constraints. These specifications then are used for both, static verification of protocol properties at design time and dynamic verification of compliance at runtime, i.e., to check that agents follow the protocol, by two abductive proof procedures.

2.2.2 Knowledge Engineering in Security Engineering

Ou et al. [OGA05] introduce MulVAL, a logic-based network security analyzer. Their tool consists of both, an extensional and an intensional database, while the former stores facts on the environment (e.g., bug specifications or configuration information of each machine and the network), whereas the latter stores rules which describe system behavior (e.g., the operating system behavior or interaction of components in the network). MulVAL returns with a list of vulnerabilities in the network that it has found during analysis of the facts w.r.t. the rules. Roschke et al. [RCSM09] extract vulnerability information from vulnerability databases like the Common Vulnerabilities and Exposures database (CVE) [MIT14] and unify the information for attack graph construction. These attack graphs further can be integrated in tools like, e.g., MulVAL, for security analysis. Jajodia et al. [JNO05] introduce Topological Vulnerability Analysis (TVA), a tool that performs a topological vulnerability analysis of network configurations. Network security conditions and attack techniques are modeled and populated to Nessus to then analyze exploit sequences that may lead to attack goals. TVA further produces a dependency graph of all possible attack paths. Lippmann et al.’s work [LIS+06] goes in the same direc-

---

9 Abductive logic programming is a high level knowledge-representation framework that extends normal logic programming by allowing predicates to be incompletely defined, declared as abducible predicates.

10 A vulnerability scanner.
tion as Jajodia et al.’s, yet with a different approach. Their tool, NetSPA, generates attack graphs for networks by combining information from (i) vulnerability scans, (ii) vulnerability databases (e.g., CVE), (iii) firewall rules and (iv) topological information that specifies how hosts and firewalls are connected together. The resulting attack graphs then are valuable in hardening the rules of involved firewalls. Almorsy et al. [AGI12] introduce a method that uses Object Constraint Language (OCL) to describe vulnerability signatures, i.e., codify security vulnerability knowledge. These vulnerability signatures are used in accordance with a static analysis tool to then identify potential flaws in software models. Eronen and Zitting [EZ01] present an expert system to analyze firewall rules. Their system is based on a knowledge base that stores information by facts and rules regarding packet structure, access lists and, topology and connection. The reasoning engine uses constraint logic programming and allows a user to interactively ask question about the permitted network traffic (though no actual traffic is generated). Thus a user can acquire new knowledge regarding the security status of a network. Sommestad et al. [SEH13] present CySeMol, the Cyber Security Modeling Language. This language is coupled with an inference system which then allows for further assessing the security of an enterprise architecture system that is modeled with CySeMol. Their work thus comprises an expert system that infers the probability that attacks on the modeled systems will succeed. Bernauer et al. [BBS+14] suggest a framework for retaining consistency in knowledge-based security testing. Essentially, they describe a method to resolve inconsistencies between general and system specific security knowledge with the goal to infer consistent knowledge to assure that sound test oracles are then used for testing. The work of Fenz and Ekelhart [FE09] addresses the definition of a security ontology for formalizing security knowledge. Their goal is to provide an efficient knowledge model useful for security risk management approaches.

2.2.3 Security Testing

Security testing is software testing of security requirements like confidentiality, integrity, authentication, authorization, availability, and non-repudiation [SGS12]. Subjected security requirements at this are divided into positive and negative requirements, i.e., requirements stating what the software shall do and what it shall not do [Ale03]. In the following we discuss state-of-the-art in penetration testing, i.e., security testing focusing on negative requirements (Section 2.2.3.1), MBST (Section 2.2.3.2), i.e., security testing using models but without a restriction on either positive or negative requirements, and web application security testing, i.e., security testing that (for the most part) focuses on the negative requirements of web applications (Section 2.2.3.3).
2.2 State-of-the-Art

2.2.3.1 Penetration Testing

Penetration testing is a kind of non-functional security testing that simulates attacks as performed by hackers. The main goal in this kind of testing is to compromise the security of a system [ASM05, Bis07]. It is one of the most prominent approaches to detect flaws in a software system, but unfortunately also one of the most inefficient [PPT89]. However, due to a lack of structured non-functional security testing approaches, penetration testing nowadays still is inevitable in discovering security flaws which require thinking out-of-the-box [Tho05].

Generally, penetration testing can be classified into (i) network penetration testing, i.e., the physical structure of a system, (ii) application penetration testing, i.e., the logical structure of a system and, (iii) social engineering, i.e., compromising a system by human interaction (e.g., tricking them into revealing passwords) [BYCJ11]. Moyer and Schultz [MS96], and Haeni [Hae97] for example present penetration testing approaches for firewall testing, i.e., targeting the physical structure of a system. Moyer and Schultz [MS96] apply a rather artisanal style by first attacking the firewall and then improve its rules based on their findings. Contrary, the idea of Haeni [Hae97] is to first gather relevant information regarding the targeted firewalls and then, perform both, attacks from the inside and the outside. The work of McDermott [McD01] introduces a penetration testing method that applies petri nets for describing attack nets. Nodes of these attack nets represent interesting states or modes of security relevant entities in the SUT. The tokens of the petri net represent whether an attacker can gain control over an entity or not. If a token reaches a goal node in the attack net, an attack has been found. Duan et al. [DZG08] present a platform for automated penetration testing of physical and logical structures (w.r.t. [BYCJ11]). Their platform consists of a control center and distributed clients. It operates by (i) gathering information of the a network an attached hosts (i.e., perform a vulnerability analysis), (ii) generate testing strategies and (iii) transform those strategies into real testing scripts that can be executed against the network and its hosts. Halfond et al. [HCO09] propose a method to vulnerability testing that, prior to testing, performs a static and dynamic analysis to discover potential input vectors (or attack patterns) in the SUT. Based on these findings, next, attacks are generated to be executed by testing against the SUT to, if existent, verify security vulnerabilities. Mainka et al. [MSS12] present a tool for automated penetration testing of XML-specific vulnerabilities in web services, e.g., SOAPAction spoofing. Their tool has a plugin based architecture, where each plugin represents an attack. Then, after loading a WSDL file, an operation is selected and, using one of the plugins, an attack is generated and sent to the web service. The work of Mainka et al. [MSS12] is quite similar to Metasploit [May11], the probably best know penetration testing tool. Metasploit offers both, facilities for developing and executing exploit code against a remote target machine. The idea of Metasploit is to (i) choose and configure
an exploit, (ii) checking whether a target system is susceptible to the exploit, (iii) create the necessary payload, (iv) choose an encoding and finally (v) execute the exploit.

Obviously, there is also much research done in the area of non-functional (or penetration) testing of web applications. However, we skip a discussion at this point, as we dedicate an own section to this topic (Section 2.2.3.3).

2.2.3.2 Model-based Security Testing

Model-based security testing is a relatively young research field, thus, related work in this area is rather scarce. The work of Blackburn et al. [BBNC01] describes a test automation framework (TAF) for model-based functional security testing of Java applications and database servers. It addresses modeling of functional security requirements in the SCRtool language, which then are transformed into test specifications and, further, test vectors and test drivers. Mouelhi et al. [MFBLT08] describe a method for the model-based testing of security policies in Java applications. Their method works in four steps, viz. (i) development of a platform independent security model, (ii) generation of a platform specific policy decision point (PDP), (iii) integration of the PDP into the application, and (iv) executing tests, generated from the platform independent model (from (i)) against the PDP implementation. Jürjens and Wimmel investigated the automated generation of test sequences from models in Focus. [11] They have applied their approach to both, model-based testing of firewalls [JW01] and reactive systems [WJ02]. Jürjens further extended UML by UMLSec, an extension to UML for secure systems development [Jü02], which he applied in model-based testing of the Common Electronic Purse Specification (CEPS) [10]. His method describes how to employ UMLSec annotated models to generate test sequences that can be used to test implementations of the CEPS for vulnerabilities. Wang et al. [WWX07] describe a threat model driven security testing approach using UML. Sequence diagrams are used to describe threat behavior, i.e., a sequence of events that is illicit. Next, on the basis of these threat models (the sequence diagrams) source code is instrumented and recompiled. As a last step, the recompiled code then is executed using random test cases. If a test trace matches a threat trace, described in any of the sequence diagrams, a vulnerability has been found. Salas et al. [SKR07] define an approach to model-based vulnerability testing using three different models, viz. (i) a state-transition model of the system, i.e., a behavior model, (ii) a faulty state-transition model of the system, i.e., an implementation model (which results from mutating the system model) and, (iii) another state-transition model for the attacker that describes the test purposes. Test cases then are generated by searching for the existence of transitions specified in the attacker model in the implementation model.

[11] The Focus language is a mathematical framework for the specification, refinement, and verification of distributed, reactive systems.
2.2 State-of-the-Art

2.2.3.3 Web Application Security Testing

The main purpose of web application security testing (as for security testing in general) is to validate whether key security concepts, e.g., confidentiality, integrity or availability (see Section 2.1.5 for a complete list) can be violated or not (e.g., by an attack). Generally it is assumed that for this, testing the network and its perimeter suffices, i.e., penetrating any firewall or deployed IDS. What however is not considered for the most part is that a majority of vulnerabilities in web applications arise from insecure and erroneous coding practices that ultimately make the application code the main source of vulnerabilities and thus, target of attackers\cite{Cen13,SBK12}. Hence, the main focus of web application security testing should be on the application and its implementation, rather than on its physical environment. For the latter, i.e., penetration testing of the physical infrastructure, techniques from penetration testing as discussed earlier (Section 2.2.3.1) can applied.

In face of existing web application security testing methods and tools we will restrict our discussion of related work to “real” testing approaches, i.e., we do not consider vulnerability scanners, static analysis, fuzzers or any other assessment methods whatsoever.

Avancini and Ceccato\cite{AC10,AC11} investigate the integration of taint analysis for security testing with genetic algorithms. The idea of their work is to use genetic algorithms to generate test cases based on the results of the taint analysis. Besides being practical in identifying vulnerabilities, their method also reduces the number of false positives, reported by taint analysis. Their method targets XSS vulnerabilities. The work of Büchler et al.\cite{BOP12a,BOP12b} motivates the use of a secure model formulated in ASLan++. This model then is mutated to introduce typical vulnerabilities in web applications, and subsequently passed to a model-checker which yields attack traces. These attack traces are then translated into test cases that are finally executed against the SUT. So far, there method was successful in detecting XSS vulnerabilities and flaws in RBAC policies. Tappenden et al.\cite{IBM+05} motivate the use of agile methods and HTTPUnit for security testing of web applications. Their approach requires the introduction of test layers in the SUT, which then allow to employ various security test patterns (e.g., bypass testing) at different layers to then test those layers w.r.t. given security requirements. Offutt et al.\cite{OWDH04} present a method for bypass testing of web applications, i.e., circumvent client-side input validation to then exploit the SUT. Their method generates client-side tests from HTML input units (e.g., forms or links) which intentionally violate explicit and implicit user checks of user input. The work of Wassermann et al.\cite{WYC+08} motivates the use of concolic execution for security testing web applications. Their method generates constraints on string values and operations which are then used during concolic testing. Upon violation of a constraint, a vulnerability has been detected. Xiong and Peyton\cite{XP10} propose a model-based framework

\footnote{ASLan++ is a specification language for model-checkers targeted to security analysis.}
for penetration testing of web applications. Their framework is linked to reference databases that are maintained by security experts. After enumerating entry points, their framework tests for known vulnerabilities by generating test cases using specified fuzz vectors. Further work of Xiong et al. [XSP09] and Stepien et al. [SXP11] consider the integration of TTCN-3 for specifying and executing of tests, resulting from the framework as introduced by Xiong and Peyton [XP10]. The work of Pethukov [PK08] describes an approach to web application penetration testing using dynamic analysis. In a first step, input test cases are executed against the SUT. During this, dynamic analysis generates traces of the data flow in the application. Then, the input test cases are optimized w.r.t. the results from the dynamic analysis, i.e., test cases and input data are fuzzed to reveal potential vulnerabilities. In a last step, the optimized test cases are executed again against the web application. Huang et al. [HTL+05] propose a black-box testing framework for web applications that is implemented as a web crawler for dynamically analyzing a web application under test. Data entry points of the web application are identified and further analyzed using fault-injection techniques, i.e., testing, to search for potential SQLI and XSS vulnerabilities. Armando et al. [ACC+10] motivate a method for web application security testing by model-checking. Using a description of the protocol used by the web application to coordinate components and respective security properties, model checking is applied to evaluate whether any of the properties can be violated. If so, a counterexample is generated that is used to eventually generate a concrete test sequence that is executed against the SUT. Bozic et al. [BSW14] combine attack pattern models as UML state diagrams with combinatorial testing. On the one side, attack patterns are used to detect potential vulnerabilities in the SUT, whereas techniques from combinatorial testing are used to capture different combinations of inputs to increase the likelihood of finding vulnerabilities in the SUT that can be exploited.

Apart from what is proposed by academia in web application security testing, there also exist tools for web application security testing that evolved from the IT industry. BurpSuite [Por14] is a platform for performing security testing of web applications. It is composed of a set of tools that support the complete testing process, from an initial semi-automatic mapping of the application and an analysis of its attack surface through finding and exploiting security vulnerabilities. Similar to BurpSuite, WebScarab [OWA14] also provides a platform for testing web applications. WebScarab is implemented as a set of plugins that intercept requests to a web application and allow a tester to review and modify these very requests before they are finally sent to the application. Further, WebScarab also allows a tester to review and inspect the responses returned from the application to search for signs of successful attacks that were crafted earlier when intercepting and modifying the requests. SQLNinja [Ice14] is another tool in a web application security tester’s

13In the following we only discuss the most prominent representatives, as providing a full survey would go beyond the scope of this thesis.
arsenal. However, compared to BurpSuite or WebScarab, SQLNinja does not target the application itself but only its back end database. Thus, SQLNinja can be used to exploit a web application’s database by more complex attacks than just plain SQLI for data extraction or manipulation (e.g., remote code execution). Contrary to the just discussed tools which are all dynamic in a sense that they execute the SUT, Fortify [HP14], another web application security assessment tool, is not dynamic as of implementing a static code analyzer. However, its purpose is similar, that is, detecting vulnerabilities in an application, yet by examining its source code by searching for vulnerable areas of code and finally reporting them to a tester for fixing them.

Our thorough discussion of state-of-the-art reveals that logic programming so far has not been applied in security testing, except in security verification of protocols, although the inherent reasoning procedures of logic programming are tempting w.r.t. automated test case generation which ultimately is an inference procedure. Further, the application of knowledge engineering by a VKB is promising, yet has not been considered in the way as we anticipate it, i.e., to store formalized security vulnerability knowledge and perform an automated security risk analysis and risk assessment of a software system by its model. Finally, in the area of web application security testing, lot of research exists and is done, however, for the most part, existing testing approaches require a security expert for successful security testing and a modeling expert for establishing a model of the SUT. Thus, our proposed method is compelling in that it (i) advances non-functional security testing of web applications using foundations of MBT, (ii) eradicates the need for a modeling expert due to automated model generation (as of the web spider) and transformation (as of the security risk analysis) and (iii) eliminates the need for a security expert to infer test cases and do actual testing (again, remember that a security expert is needed for establishing and maintaining the VKB).
Chapter 3

Method

Research is to see what everybody else has seen, and to think what nobody else has thought.

Albert Szent-Gyorgyi

In this chapter, we introduce our novel model-based approach for non-functional security testing of web applications by logic programming in Prolog. We first introduce our method (3.1). Next, we discuss the high-level modeling facilities of our method, i.e., its DSLs and supporting model transformations (Section 3.2). We conclude this chapter with the formal foundations (Section 3.3).

As already stated in Chapter 1, w.r.t. the MBST classification of Felderer et al. [FAZB11] (Figure 1.5) our proposed method as introduced in the following can be classified as automated RB security test generation (where RB stands for risk-based). This is due to that our proposed method builds on codified security vulnerability knowledge to automatically generate a risk profile. Then, as of being fully integrated into the testing process, this risk profile allows for fully automated risk-based security test generation.

Our proposed method for non-functional, model-based security testing of web applications (Section 3.1) essentially is a modification of variant (f) of the MBT approach as identified by Schieferdecker [Sch07] from Figure 2.19. Compared to Schieferdecker’s original approach, in our modified approach, the system is not generated from a model of the SUT but instead the model of the SUT is derived from an already implemented system. Our method thus represents a system-driven MBT testing approach for non-functional security testing. Figure 3.1 shows this adapted scenario.
3.1 Model-based, Non-functional Security Testing of Web Applications

Our model-based, non-functional security testing method tests for a web application if it contains known vulnerabilities. It is designed to detect SQLI and XSS vulnerabilities as described in Section 2.1.4.1 in web applications by searching for user-controllable operations, i.e., operations, which take user input, and thus, open up a potential vulnerability. The operations, more precisely, its parameters are then analyzed to infer, which kind of SQLI or XSS attack to employ. Our method, upon successful identification of a potential SQLI or XSS vulnerability, then generates malicious input strings that are fed as input to user-controllable operations to verify “truly” existing SQLI and XSS vulnerabilities by testing. Our method returns with a test log and test feedback in the risk profile $\mathcal{RP}$.

Figure 3.1 shows the artifacts of our proposed testing method and their relationships. W.r.t. Figure 2.21, our testing method represents an MBST approach. Necessary security vulnerability knowledge is codified into our Vulnerability Knowledge Base (VKB; Section 3.1.2) by (i) the extensional database ($\mathcal{EDB}$) that stores knowledge by facts, (ii) the intensional database ($\mathcal{IDB}$) that stores the reasoning rules which “use” the knowledge of the $\mathcal{EDB}$, and (iii) test data definitions for later test execution. These artifacts eventually subsume security test requirements.
more precisely, negative requirements, or security properties) that are provided by a security expert. The knowledge of the VKB is used during a security risk analysis (Section 3.1.2) on the security problem $SP$, i.e., the model of the SUT, (Section 3.1.1) which yields the risk profile $RP$ (Section 3.1.3), i.e., the risk profile $RP$ is the result of a model-2-model transformation from the security problem $SP$. The risk profile $RP$ further is used for test case generation (Section 3.1.4) and subsequent test execution (Section 3.1.5) by our test controller. Necessary test data and test oracles are also provided by the VKB in terms of test data definitions (Section 3.1.2) and potential attack goals (Sections 3.1.6 and 3.1.6). We do not consider the implementation of the system under test w.r.t. a security problem $SP$ which is indicated by the dashed line in Figure 3.2.

![Image of Figure 3.2]

Figure 3.2: Overview of the necessary artifacts of our proposed method and their relationship.

The testing process described by our method is shown in Figure 3.3. It can be
subsumed in four steps, viz.:

1. Automatically establish a security problem $SP$, the model of the SUT, by a web spider (Sections 3.1.1 and 3.2.1).

2. Execute a security risk analysis on the security problem $SP$ yielding the risk profile $RP$. This also contains test generation (Sections 3.1.2, 3.1.3 and 3.1.4).

3. Execute the risk profile $RP$ (resulting from step 2) against the SUT by testing (Section 3.1.5).

4. Evaluate each executed test by predefined oracles and log the corresponding verdict. This also contains generation of the test log and test feedback in the risk profile $RP$ (Sections 3.1.6 and 3.2.3).

Figure 3.3: Abstract overview of the testing process of our non-functional security testing method for web applications. After automatically establishing a security problem $SP$ by a web spider, a security risk analysis is executed on it yielding the risk profile $RP$. This risk profile $RP$ acts as executable specification for non-functional security testing. Our method returns with a test log and test feedback in the risk profile $RP$. 
In the testing process as described in Figure 3.3 all activities except knowledge codification are done automatically, which is symbolized by different fill colors for the various activities (rounded rectangles). Classic shaped rectangles denote any artifacts that are involved in the testing process. In the following, the testing process (Figure 3.3) and any involved artifacts (Figure 3.2) are discussed in detail.

3.1.1 Security Problem

A security problem \( SP \) describes a declarative system model of a SUT at an interface level. Its name is due to that we consider the SUT to be vulnerable, thus it comprises a security problem. It is the primary input to our testing method and its security risk analysis.

Generally, a software model is considered a formal abstraction of a real-world entity or process. In this sense, a model can be defined as a finite, enumerable set of facts with different properties. Thus, models formalize real-world knowledge (e.g., on an entity or process).

As introduced in Section 2.1.1, logic programs by definition are sets of rules in the form of \( A \leftarrow L_1, \ldots, L_m \), where rules may have an empty body, i.e., \( A \leftarrow \), then called a fact. Hence, supporting rules and facts, the semantics of logic programs obviously support both, dynamic and static modeling of software (i.e., entity and process modeling). Put another way, using semantics of logic programs allows to define the domain of discourse, i.e., describing a software model with a declarative syntax.

Consider the following grounded, normal logic program, which specifies a PHP application with the vulnerable code snippet from Figure 2.9, e.g., \( \Pi_{SP} = \)

\[
\begin{align*}
\text{module}(auth), \\
\text{uri}(auth, "http://www.victim.com"). \\
\text{operation}(auth;login,[uname,pword],void), \\
\text{parameter}(auth;login;uname;text), \\
\text{parameter}(auth;login;pword;password).
\end{align*}
\]

Program \( \Pi_{SP} \) represents a logic program, or, security problem \( SP \) over an alphabet \( \Sigma_{SP} \).

**Definition 3.1** (Security Problem). The security problem, \( SP \), describes a declarative system model, or specification, of a SUT by a logic program at an interface level. It is a set of facts with terms over the alphabet \( \Sigma_{SP} = \langle \mathcal{F}, \mathcal{E}, \mathcal{X} \rangle \), with disjoint
sets of symbols

\[ \mathcal{F} = \{ \text{module}, \text{uri}, \text{operation}, \text{parameter} \} \text{ is a finite set of predicates,} \]
\[ \mathcal{C} = \{ c_1, \ldots, c_m \} \text{ is a set of constants, and} \]
\[ \mathcal{X} = \{ X_1, \ldots, X_n \} \text{ is a set of variables} \]

Further, \( \mathcal{C} = \langle \mathcal{M}, \mathcal{O}, \mathcal{P}, \mathcal{T}, \mathcal{U} \rangle \), with disjoint sets of symbols, where

- \( \mathcal{M} \) is a set of software modules,
- \( \mathcal{O} \) is a set of operations,
- \( \mathcal{P} \) is a set of parameters,
- \( \mathcal{T} \) is a set of types, and
- \( \mathcal{U} \) is a set of uniform resource identifiers (URI).

The predicates described by set \( \mathcal{F} \) are functions\(^2\) over \( \mathcal{C} \) with

- \( \text{module} \subseteq \mathcal{M} \) to declare a new software module of a SUT,
- \( \text{uri} \subseteq \mathcal{M} \times \mathcal{U} \) to declare the URI of the module,
- \( \text{operation} \subseteq \mathcal{M} \times \mathcal{O} \times \varphi(\mathcal{P}) \times \mathcal{T} \) to declare an operation of a module with its parameters and its return type, and
- \( \text{parameter} \subseteq \mathcal{M} \times \mathcal{O} \times \mathcal{P} \times \mathcal{T} \) to declare a parameter of an operation with its type\(^4\).

Hence, by Definition 3.1, program \( \Pi_{SP} \), in a more literal sense, describes a software system, e.g., a web application, which realizes a module auth that offers an operation login with two parameters, viz. \text{uname} of type \text{text}, and \text{pword} of type \text{password}. As the parameters are controllable by the user, and thus open up a vulnerability, program \( \Pi_{SP} \) describes a security problem \( SP \), i.e., the knowledge of the real-world to be formalized, or domain of discourse, later subjected to a security risk analysis.

In the Herbrand theory, \( \Pi_{SP} \) describes a subset of the Herbrand base \( H_B \), i.e., \( \Pi_{SP} \subseteq H_B \), which is computed on the basis of \( \Pi_{SP} \)'s Herbrand universe \( H_U \). In logic programming, a least Herbrand model, i.e., \( H^L_{\Pi_{SP}} \subseteq H_B \) (the set of all

\(^1\)Variables are necessary for that terms can occur ungrounded.
\(^2\)Functions are predicates that return any value compared to predicates that can only return boolean values.
\(^3\)Observe that this predicate “transforms” a parameter from a symbol into something to reason about.
\(^4\)The code snippet in Figure 2.9 does not show the module name simply due to that it is the name of the file which contains the code. The same applies for the content of the predicate \text{uri}/2, as this information is not coded anywhere in the application, but inherently provided by the application’s server.
3.1 Model-based, Non-functional Security Testing of Web Applications

grounded predicates), is computed as a solution for a logic program $\Pi$. As this computation is done on the Herbrand base $H_B$, our definition of the security problem by a declarative system model utterly fits into the notion of logic programming.

For program $\Pi_{SP}$, set $\mathcal{C}$ of alphabet $\Sigma_{SP}$ contains the disjoint sets

- $\mathcal{M} = \{auth\}$,
- $\mathcal{D} = \{login\}$,
- $\mathcal{P} = \{uname, pword\}$,
- $\mathcal{T} = \{text, password\}$, and
- $\mathcal{U} = \{"http://www.victim.com"\}$.

3.1.2 Vulnerability Knowledge Base

The VKB is the linchpin of our security testing method. It is an expert system that (i) stores security vulnerability knowledge, (ii) implements an automated security risk analysis and a risk assessment procedure, and (iii) stores test data definitions w.r.t. known exploits (codified in the $\mathcal{EDB}$) for generating test data. Figure 3.4 shows its internal architecture.

![Diagram of VKB architecture]

The $\mathcal{EDB}$ (Section 3.1.2.1) stores declarations of exploits and their respective attacks as well as attack patterns, i.e., the description of an attack vector by facts. This knowledge is used by the inference engine (in our case a Prolog solver [DC00]).
during the security risk analysis and the risk assessment to satisfy the rules, i.e., logical predicates, of the $\mathcal{LDB}$ (Section 3.1.2.2) to establish a risk profile $\mathcal{RP}$ on the grounds of a provided security problem $\mathcal{SP}$. Finally, we use DCGs to codify the structure of test data definitions w.r.t. known exploits (Section 3.1.2.3). These test data definitions are necessary to later generate malicious inputs during testing to verify existing vulnerabilities in the SUT.

### 3.1.2.1 Extensional Database

The $\mathcal{EDB}$ forms the memory of our VKB. It stores security vulnerability knowledge, i.e., exploits, attacks and attack patterns as shown in Figure 3.5. Exploits have a name and associated a set of attacks (remember that, e.g., for SQLI we described three different variations in Section 2.1.4.1), as well as attack patterns. Each attack declares a name on its own and a set of potential attack goals (Section 2.1.4.1) that later (during test evaluation) occupy the role of test oracles. The purpose of an attack pattern is to describe an attack vector, i.e., the means by which a vulnerability in the SUT potentially can be exploited by an attack. For this, each attack pattern, besides its name also declares a vulnerable type that is exploited by a concrete attack.

![Figure 3.5: Relationship of exploit, attack and attack pattern as codified in the $\mathcal{EDB}$. Each exploit has associated different attacks (i.e., variations thereof) and attack patterns.](image)

The knowledge of the $\mathcal{EDB}$ is declared by logic programs over an alphabet $\Sigma_{\mathcal{EDB}}$ (Definition 3.2). It should be mentioned, that the knowledge of the $\mathcal{EDB}$ is system and language specific, i.e., it may not be applicable to different types of systems, e.g., web or embedded applications, as well as to different languages, e.g., PHP or C. This is to avoid misinterpretations during reasoning.

**Definition 3.2** (Extensional Database). *The extensional database, $\mathcal{EDB}$, denotes a formalization of security vulnerability knowledge by a logic program. It is a set of facts and rules with terms over the alphabet $\Sigma_{\mathcal{EDB}} = (\mathcal{F}, \mathcal{C}, \mathcal{X})$, with disjoint sets of*
3.1 Model-based, Non-functional Security Testing of Web Applications

symbols

\[ \mathcal{F} = \{ \text{exploit}, \text{attack}, \text{vul.type}, \text{attack.pattern} \} \] is a finite set of predicates,
\[ \mathcal{C} = \{ c_1, \ldots, c_m \} \] is a set of constants, and
\[ \mathcal{X} = \{ X_1, \ldots, X_n \} \] is a set of variables.\footnote{Again, variables are necessary for that terms can occur ungrounded (in particular for attack.pattern/5).}

Further, \( \mathcal{E} = \langle \mathcal{E}, \mathcal{A}, \mathcal{AP}, \mathcal{G}, \mathcal{T} \rangle \), with disjoint sets of symbols, where

\[ \mathcal{E} \] is a set of exploits,
\[ \mathcal{A} \] is a set of attacks
\[ \mathcal{AP} \] is a set of attack patterns,
\[ \mathcal{G} \] is a set of attack goals, and
\[ \mathcal{T} \] is a set of types.\footnote{Observe that this set is equal to set \( \mathcal{T} \) from Definition 3.1.}

The predicates described by set \( \mathcal{F} \) are functions over \( \mathcal{C} \) with

- \( \text{exploit} \subseteq \mathcal{E} \) to declare an exploit,
- \( \text{attack} \subseteq \mathcal{E} \times \mathcal{A} \times \mathcal{AP} \times \varphi (\mathcal{G}) \) to declare an attack
  - under an exploit, its attack pattern and goals
- \( \text{vul.type} \subseteq \mathcal{A} \times \mathcal{T} \) to declare a type, exploitable by an attack, and
- \( \text{attack.pattern} \subseteq \mathcal{AP} \times \mathcal{A} \times \mathcal{M} \times \mathcal{O} \times \mathcal{P} \) to declare an attack pattern
  - for an attack.

With alphabet \( \Sigma_{EDB} \) an EDB then is filled with knowledge as, e.g., in program \( \Pi_{EDB} = \)

\[
\begin{align*}
\text{exploit}(\text{sql.attack}), \\
\text{attack}(\text{sql.attack}, \text{signature.evasion}, \text{sqlap}, \\
\text{[authentication, leakage, tampering]}), \\
\text{vul.type}(\text{sql.attack}, \text{text}), \\
\text{vul.type}(\text{sql.attack}, \text{password}), \\
\text{attack.pattern}(\text{sqlap}, \text{sql.attack}, X_M, X_O, X_P) \leftarrow \text{module}(X_M), \\
\text{operation}(X_M, X_D, _, _), \\
\text{parameter}(X_O, X_P, X_T), \\
\text{vul.type}(\text{sql.attack}, X_T).
\end{align*}
\]

Program \( \Pi_{EDB} \) describes an SQLI exploit by signature evasion, i.e., evading a pro-
grammer’s intended SQL query. The list of its potential goals includes leakage or tampering. Program \( \Pi_{EDB} \) further declares an attack pattern for SQLI attacks, e.g., sqlap that matches any operation in a security problem \( SP \) that provides parameters of type text or password.

From the predicates of the \( \mathcal{E}DB \), attack_pattern/5 is the only rule. Although this predicate thus already infers new knowledge, it is custom for each attack (and exploit). Hence, it is declared as part of the \( \mathcal{E}DB \), as, contrary to the rules of the \( \mathcal{T}DB \), attack_pattern/5 cannot be declared “generic”. For this, see again program \( \Pi_{EDB} \). There, the declaration of attack_pattern/5 is partly grounded by the two constants sqlap and sql_attack. These parameters need to be bounded in the head of the rule to establish the necessary link between attacks and their respective attack patterns. However, for that attack_pattern/5 can be grounded completely, i.e., \( X_{\text{sql}}, X_O, X_P \) can be bound, a security problem \( SP \) needs to be present.

For program \( \Pi_{EDB} \), set \( C \) of alphabet \( \Sigma_{EDB} \) contains the disjoint sets

\[
\begin{align*}
C &= \{ \text{sql\_attack} \}, \\
A &= \{ \text{signature\_evasion} \}, \\
AP &= \{ \text{sqlap} \}, \\
\mathcal{G} &= \{ \text{authentication}, \text{leakage}, \text{tampering} \}, \text{ and} \\
\mathcal{T} &= \{ \text{text, password} \}.
\end{align*}
\]

### 3.1.2.2 Intensional Database

The \( \mathcal{T}DB \) contributes the rational reasoning procedures for the security risk analysis and the risk assessment. It contains the rules, necessary to “match” the knowledge of the \( \mathcal{E}DB \) against a security problem \( SP \) to deduce a risk profile \( RP \), that is, to infer new knowledge. Figure 3.6 illustrates this deduction procedure for one concrete risk based on the security problem \( SP \) from program \( \Pi_{SP} \) (Section 3.1.1), the \( \mathcal{E}DB \) from program \( \Pi_{EDB} \) (Section 3.1.2.1) and substitutions \( \theta_1, \theta_2, \theta_3, \theta_4 \) and \( \theta_5 \). Further, Figure 3.6 introduces the main predicates of our risk analysis, viz. blacklist/6, threat/6 and comp/4.

For deducing a risk profile \( RP \) (Section 3.1.3) our security risk analysis returns with multiple grounded instances of the risk/7 predicate, i.e., potential risks in a security problem \( SP \) (see top of Figure 3.6 for one such grounded instance with substitution \( \theta_1 \)). First, a solver (GNU Prolog \[DC00\]) unifies facts declared by a security problem \( SP \) with attack patterns from the \( \mathcal{E}DB \), i.e., free variables of the attack_pattern/5 predicate are bound to constants of the security problem \( SP \). This happens as part of satisfying the blacklist/5 predicate, whose purpose is to determine potentially vulnerable operations as of their parameters in the security problem \( SP \). The blacklist/5 predicate unifies information regarding a vulnerable operation \( (X_O) \) and the corresponding parameter \( (X_P) \) of a software module \( (X_M) \) with the matching attack pattern \( (X_{\text{AP}}) \) and its corresponding exploit \( (X_E) \). Figure 3.7 illustrates this “blacklisting”.
For each blacklisted operation, the \texttt{threat/6} predicate then instantiates threats for all attacks w.r.t. to the exploit that corresponds to the matching attack pattern. For this, the \texttt{threat/6} predicate unifies all the information necessary to later generate a risk, i.e., the software module’s (\texttt{X_M}), operation’s (\texttt{X_O}) and parameter’s name (\texttt{X_P}), as well as the exploit (\texttt{X_E}), a concrete attack (\texttt{X_A}) and potential attack goals (\texttt{X_G}). A concrete attack is inferred from the knowledge of the EDB that “knows” about various attacks w.r.t. an exploit. By virtue of Prolog’s solution strategy, each concrete attack, i.e., instance of the \texttt{attack/4} predicate, will be unified with instances of the \texttt{threat/6} predicate. For example, for the security problem \texttt{SP} from program \texttt{SP} (Section 3.1.1) and the EDB from from program \texttt{EDB} (Section 3.1.2.1) this would yield a total of two instances of the \texttt{threat/6} predicate due to the contents of the security problem \texttt{SP}, viz. one operation (\texttt{login}) with two parameters (\texttt{uname} and \texttt{pword}), and one declared attack in the EDB, viz. SQLI by...
signature_evasion. These threats next are subjected to a risk assessment that yields concrete risks (grounded instances of the risk/7 predicate). This risk assessment is achieved by satisfying the comp/4 predicate, whose outcome \( C \), the operation complexity, is used to calculate the potential impact of an attack, the probability for succeeding with it and, based on those values, the risk level for the resulting risk. \( \text{comp/4} \) calculates the operation complexity \( C \) by recursively iterating over the list of parameters \( (X_P) \) of an operation \( (X_O) \) declared by a software module \( (X_M) \). Finally, if threat/6 and \( \text{comp/4} \) are fully grounded, the risk/7 predicate (Section 3.1.3), the top-level predicate of our security risk analysis, can be satisfied, i.e., all information of an instance of the threat/6 predicate is unified with the corresponding risk assessment. Figures 3.8-3.11 show an algorithmic description of this deduction process, in future referred to as program \( \Pi_{IDB} \).

Figure 3.8 lists the main procedure, i.e., risk_analysis, of the risk analysis algorithm. Its inputs are (i) the security problem \( SP \), (ii) the set of known attacks \( A \), and (iii) the set of related attack patterns \( AP \). For each operation \( op \in SP \) it evaluates, whether there exists an attack pattern \( ap \in AP \) that matches the attack vector, described by the operation’s input parameters, i.e., \( ap \mu op \) (read “ap matches op” with \( \mu \subseteq AP \times O \)). If a matching attack pattern is found, operation \( op \) is added to the set \( OP' \) of blacklisted operations. Next, in line 9, for each of the blacklisted operations of set \( OP' \), all applicable attacks \( a \) are collected in the set \( A_{op} \), i.e., \( ap \mu op \land ap \sigma a \) (read “ap matches op and ap subsumes a” with \( \sigma \subseteq AP \times A \)). Then, for each operation-attack pair, a threat \( thr \) is created which then is evaluated by calling the assess procedure (Figure 3.9). Its return value is used to create a new risk, which finally is added to the set \( RP \), which describes
3.1 Model-based, Non-functional Security Testing of Web Applications

the result of the risk analysis, the inferred risk profile $\mathcal{RP}$.

**procedure** RISK\_ANALYSIS($\mathcal{SP}, \mathcal{A}, \mathcal{AP}$)

1. $\mathcal{OP}' \leftarrow \emptyset$
2. $\mathcal{RP} \leftarrow \emptyset$
3. for each $op \in \mathcal{SP}$ do
   4. if $\exists ap \in \mathcal{AP}: ap \mu op$ then
      5. $\mathcal{OP}' \leftarrow \mathcal{OP}' \cup op$
   6. end if
7. end for
8. for each $op \in \mathcal{OP}'$ do
   9. $\mathcal{A}_{op} \leftarrow \{ a : ap \mu op \land ap \sigma a : ap \in \mathcal{AP}, a \in \mathcal{A} \}$
10. for each $a \in \mathcal{A}_{op}$ do
    11. $\mathcal{thrt} \leftarrow (op, a)$
    12. $\mathcal{risk} \leftarrow (\mathcal{thrt}, \text{ASSESS}(\mathcal{thrt}))$
13. $\mathcal{RP} \leftarrow \mathcal{RP} \cup \mathcal{risk}$
14. end for
15. end for
16. return $\mathcal{RP}$
17. end procedure

Figure 3.8: Main procedure of the risk analysis algorithm. It “blacklists” vulnerable operations $op$ for subsequent risk assessment and risk profile deduction. It returns with a new risk profile $\mathcal{RP}$.

The main task of the risk\_analysis procedure, aside from deducing a new risk profile $\mathcal{RP}$, is to identify potentially vulnerable operations w.r.t. their parameters in a security problem $\mathcal{SP}$ on the grounds of existing knowledge from the EDB. Thus, our risk analysis may only detect what it knows, i.e., it is no silver bullet against all kinds of cyber attacks, only against what we already know (and have not missed to formalize in the EDB).

As just discussed, the risk\_analysis procedure calls the assess procedure (Figure 3.9) to evaluate a threat $\mathcal{thrt}$. For this, it receives a threat $\mathcal{thrt}$ as input parameter. The assess procedure then first calls the comp procedure to calculate the complexity of the threat’s subjected operation (Figure 3.11). Based on the returned complexity value $c_{op}$, the assess procedure next calculates the potential impact of threat $\mathcal{thrt}$’s underlying attack, the probability of succeeding with the attack and, based on those two former values, the overall risk level for threat $\mathcal{thrt}$.

The potential impact of an attack is the direct relation to the operation’s complexity, i.e., the higher the complexity of an operation (or rather its input parameters), the higher the potential impact of a successful attack. Thus, the impact reflects the complexity value $c_{op}$. The probability of a successful attack then is calculated next by taking the inverse of the impact. For example, a complexity value
parameter: Threat thrt
result : Risk assessment asmnt

19: procedure ASSESS(thrt)
20: \( \text{imp} \leftarrow \text{COMP}(\text{thrt} \rightarrow \text{op} \rightarrow p) \oplus \text{SEVERITY}(\text{thrt} \rightarrow a) \)
21: \( \text{prob} \leftarrow \text{PROBABILITY}(\text{impact}) \)
22: \( \text{rl} \leftarrow \text{RISK\_LEVEL}(\text{impact}, \text{probability}) \)
23: \( \text{asmnt} \leftarrow (\text{imp}, \text{prob}, \text{rl}) \)
24: return asmnt
25: end procedure

Figure 3.9: Risk assessment procedure of the risk analysis algorithm. It deduces for each operation and attack pair a risk assessment by the operation complexity.

of high, i.e., \( c_{op} = \text{high} \) would result in an impact with value high and a probability of value low. We justify this approach by the circumstance that, the more complex an operation’s parameters, the more difficult it is for a malicious agent to craft a successful attack. On the other side, the more trivial the operation parameters, the easier it is to craft a successful attack, however, its impact may not be that serious.

For this consider the following example. On the one side, we may have an operation with a set of complex parameters (e.g., objects) that require, upon receiving, parsing by the application. On the other side, we may have an operation with only a few trivial parameters (e.g., numeric values). For the first case, i.e., the operation with a set of complex parameters, a malicious agent needs to know quite well, where (e.g., inside the structure of such a complex parameter) to put his malicious input, for that it causes it successfully exploits a vulnerability in the application, thus a lower probability that the attack succeeds. Contrary however, the impact is high, as if the attack is successful, a malicious agent can harm the application quite drastically. For the second case, i.e., the operation with only trivial parameters, a malicious agent may succeed with an attack quite easily, e.g., simply by overflowing any of the input parameters or trigger a subsequent computation to overflow. Yet, the impact maybe little, as such a trivial overflow attack normally does not allow to gain control over an application. The probability for succeeding however is high, as an attack does not require much more, than tampering around with input values, instead of manually crafting malicious inputs.

With the calculated impact and probability, the assess procedure, as a last step, calculates the overall risk level for the resulting risk. This is done by a simple table look-up, where the two input parameters, i.e., impact and probability, are used to derive the risk level (Figure 3.10). The assess procedure finally returns with a triplet which encapsulates (i) the potential impact of threat thrt’s underlying attack, (ii) the probability of succeeding with the attack and, (iii) the overall risk level for threat thrt.

The assess procedure, besides the already mentioned predicates of the risk analysis, further introduces the predicates probability/2 and risk_level/3, which
implement deduction of the respective values, as just discussed. The resulting values for each, i.e., impact, probability and risk level are values of a 5-point Likert-type scale with points \{very low, low, medium, high, very high\} (Figure 3.10). Remember that the application of a Likert-scale for doing risk assessment is only one possible solution, yet it provides a handy and human friendly tool for this task.

<table>
<thead>
<tr>
<th>Impact</th>
<th>VERY LOW</th>
<th>LOW</th>
<th>MEDIUM</th>
<th>HIGH</th>
<th>VERY HIGH</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERY LOW</td>
<td>VERY LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>HIGH</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>HIGH</td>
<td>HIGH</td>
<td>HIGH</td>
<td>VERY HIGH</td>
<td>HIGH</td>
</tr>
</tbody>
</table>

Figure 3.10: Look-up table for risk assessment. Based on given impact and probability values, the assess procedure calculates the risk level by consulting this look-up table.

The last procedure to be discussed as part of our security risk analysis is \texttt{comp} (Figure 3.11). As already discussed, it is called by the assess procedure for calculating an operation \texttt{op}'s complexity by its list of parameters, subsequently used for risk assessment. We motivate this approach, as the operation’s parameters define the attack vector, the means by which a malicious agent can intrude a software system and do further harm \cite{ZFB12a,ZFB13}. Equation 3.1 shows the underlying formula of our complexity calculation. The complexity factor \( c \) then reflects the complexity of an operation \texttt{op} by its signature and is denoted \( c(op) \). The more complex the set of input parameter types, the higher is the overall complexity \( c \). The complexity of type \( t_i \) considers whether a type is primitive or complex (i.e., has an internal structure) and is denoted \( c(t_i) \). The dependence factor between \( t_1, \ldots, t_n \) is denoted by \( d \). The overall complexity \( c \) of an operation \texttt{op} with the operation signature \( \texttt{op}(p_1 : t_1, \ldots, p_n : t_n) \) then is the sum of the input parameter type complexities \( c(t_i) \) and the dependence factor \( d(t_1, \ldots, t_n) \).

\[
c(op) = \sum_{i=1}^{n} c(t_i) + d(t_1, \ldots, t_n) g \tag{3.1}
\]

Figure 3.11 shows the algorithm used for calculating the operation’s complexity \( c(op) \). If the list of parameters is empty, the \texttt{comp} procedure simply returns with a complexity value of low. Contrary, if it contains any element, \texttt{comp} recursively descends until it reaches the empty list. It then returns (as just mentioned) with a complexity of low and subsequently, while returning for each recursive call, gradually calculates operation \texttt{op}'s complexity (line 33). For this, the operator \( \oplus \) infers the new complexity to be returned based on the actual parameter’s complexity \( \Delta_{\text{comp}} \) and the complexity value returned by the recursive call to \texttt{comp}. In our implementation, the operator \( \oplus \) is implemented by the \texttt{dcomp/3} predicate, which also uses the look-up table from Figure 3.10 for inferring the new complexity based on its two inputs,
as just discussed. After the last recursive call has returned, \texttt{comp} calculates the final operation complexity \(c(op)\), again in line 33, and returns with this value.

\begin{verbatim}
parameter: List of parameters \(\mathcal{L}\)
result : Operation complexity \(c\)

26: procedure \texttt{comp}(\mathcal{L})
27:   if \(\mathcal{L} = \emptyset\) then
28:     return \textit{low}
29:   else
30:     \texttt{param} \leftarrow \texttt{take}(\mathcal{L})
31:     \texttt{type} \leftarrow \texttt{type_of}(\texttt{param})
32:     \Delta_{\text{comp}} \leftarrow \texttt{comp}(\texttt{type})
33:     return \Delta_{\text{comp}} \oplus \texttt{comp}(\mathcal{L})
34:   end if
35: end procedure
\end{verbatim}

Figure 3.11: Operation complexity calculation procedure of the risk analysis algorithm. For each operation’s list of parameters, this procedure calculates the operation complexity by its attack vector, i.e., the list of parameters.

Our implementation of a security risk analysis mimics a human being in doing a security risk analysis. Existing knowledge is used to, based on an established set of rules (i.e., guidelines), infer new, learnable knowledge. Its “logical” nature allows for an efficient computational representation by logic programming and eradicates the need for a security expert except establishing the E\textit{DB}. The new knowledge manifests itself in the risk profile \(\mathcal{R}\mathcal{P}\), which contains valuable information regarding potential vulnerabilities in the assessed SUT by valued risks. Especially the assessment of risks is notably valuable, as it later allows to prioritize resulting test cases. Such a prioritization by, e.g., order of execution, advances non-functional security testing to a more structured testing process, away from its prevalent penetration testing-like style.

3.1.2.3 Test Data

As already mentioned, we use DCGs to both, codify the structure of and generate test data in our VKB. DCGs are a Prolog formalism, which allow to state CFGs and CSGs as normal logic programs. At this, DCGs work in both ways, i.e., they allow to verify that a sentence is correct w.r.t. a grammar, but also to generate sentences w.r.t. a grammar. By considering the syntax based nature of malicious input data in non-functional security testing, e.g., SQL\textit{I} input strings, DCGs then offer a powerful mechanism for generating elaborate malicious test data by stating grammars, which invalidate the proper syntax of the input, expected by a SUT.

Definition 3.3 introduces the language to be used by logic programs, which describe DCGs for test data generation in our VKB.
Definition 3.3 (Test Data Specification). The test data specification, $\mathcal{TDS}$, describes either a context-free or context-sensitive grammar by a logic program. It is a set of rules and facts with terms over the alphabet $\Sigma_{\mathcal{TDS}} = \langle \mathcal{F}, \mathcal{C}, \mathcal{X} \rangle$, with disjoint sets of symbols

$\mathcal{F} = \{\text{testdata}\}$ is the initial set of predicates,
$\mathcal{C} = \{c_1, \ldots, c_m\}$ is a set of constants, and
$\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables.

Further, $\mathcal{C} = \langle \mathcal{E}, \mathcal{A}, \mathcal{I}, \mathcal{T}, \mathcal{S} \rangle$, with disjoint sets of symbols, where

$\mathcal{E}$ is a set of exploits,
$\mathcal{A}$ is a set of attacks,
$\mathcal{I}$ is a set of production rule identifiers,
$\mathcal{T}$ is a set of types, and
$\mathcal{S}$ is a set of terminal symbols.

where sets $\mathcal{E}$, $\mathcal{A}$, and $\mathcal{T}$ are equal with their identically labeled counterparts from Definitions 3.1 and 3.2.

The predicate described by the initial set $\mathcal{F}$ is a function over $\mathcal{C}$ with

$\text{testdata} \subseteq \mathcal{E} \times \mathcal{A} \times \mathcal{T} \times \mathcal{I}$ to query for new data.

Contrary to $\Sigma_{SP}$ and $\Sigma_{EDB}$, the set $\mathcal{F}$ of predicates of alphabet $\Sigma_{\mathcal{TDS}}$ is not finite. Thus initially it only contains $\text{testdata}/4$ as the only predicate, which declares the entry point for test data generation, i.e., the predicate used to query our test data generator for new test data. Any further predicates of set $\mathcal{F}$ are purely dependent on the test data to be generated. Put another way, for $\Sigma_{\mathcal{TDS}}$, we only require the predicate $\text{testdata}/4$ to occur in a logic program, describing a DCG for test data generation. This allows to keep the test data generator as generic as possible by providing a single “interface” predicate to query for new data, but “hide” the concrete implementation. Thus, a security expert who declares the $\mathcal{EDB}$ in the same breath can declare the necessary DCGs for any formalized attack in the $\mathcal{EDB}$. Consider for example the case when generating test data for SQLI to evade the signature of a programmer’s intended query (as codified in the $\mathcal{EDB}$ from program $\Pi_{EDB}$ in Section 3.1.2.3). This would require a grammar that allows to produce input strings like, e.g., `' OR '1' = '1'--`. Program $\Pi_{\mathcal{TDS}}$ describes a

Again, variables are necessary for that production rules of the grammar (i.e., terms) can occur ungrounded.
DCG that allows for that, e.g., $\Pi_{TDS} =$

$$\text{testdata}(sql\_attack, \text{signature}\_\text{evasion}, \ldots, S_0, S) \leftarrow$$

$$\text{apostrophe}(S_0, S_1), \text{or}(S_1, S_2), \text{apostrophe}(S_2, S_3), \text{number}(S_3, S_4),$$

$$\text{apostrophe}(S_4, S_5), \text{equals}(S_5, S_6), \text{apostrophe}(S_6, S_7), \text{number}(S_7, S_8),$$

$$\text{apostrophe}(S_8, S_9), \text{comment}(S_9, S),$$

$$\text{apostrophe}(S_0, S), \text{for} (S_0, 1, S),$$

$$\text{equals}(S_0, S) \leftarrow \text{connects}(S_0, =, S),$$

$$\text{for} (S_0, 1, S),$$

$$\text{connects}(S_0, =, S),$$

$$\text{for} (S_0, \text{OR}, S),$$

$$\text{for} (S_0, \text{OR}, S),$$

$$\text{for} (S_0, \text{for} (S_0, ||, S).$$

Program $\Pi_{TDS}$

8, if then queried for new test data would return with the two solutions ' OR '1' = '1'-- and ' || '1' = '1'--. Obviously, this grammar does not allow to generate anything else. Yet, for the purpose of evading a programmer’s intended SQL query signature it would suffice, at least if done on a MsSQL, PostgresQL, or Oracle database server. However, on a MySQL database server the attack would fail, as MySQL expects a trailing white space after a comment (e.g., “–”). Thus, for that the data generated by program $\Pi_{TDS}$ works for at least the four just mentioned database vendors (i.e., MsSQL, MySQL, Oracle and PostgresQL), the set of rules would need to be refined and extended. It is exactly for this reason why set $\mathcal{F}$ of alphabet $\Sigma_{TDS}$ is not finite and solely provides \text{testdata}//4 as the only predicate. Thus, a security expert has his freedom in declaring DCGs for test data generation, thereby keeping them generic and extensible and only restricted to provide \text{testdata}//4 as the main goal predicate. Appendix \ref{app:dcg} shows further examples of DCGs for test data generation.

At first sight, the grammar described by program $\Pi_{TDS}$ does not appear to be very efficient by length and rule complexity (due to the difference lists). However, this is only as we have used our formal notation for logic programs. Using Prolog’s syntax for declaring DCGs results in a much more concise formulation of the grammar from program $\Pi_{TDS}$, as shown in Figure \ref{fig:logic_program}.

Obviously, extending this grammar to meet further requirements is straightforward, i.e., it just needs to be extended by the necessary predicates and terminal symbols. Consider for example the necessary extensions for program $\Pi_{TDS}$ to generate test data to work for all four mentioned database vendors. Figure \ref{fig:logic_program} shows the resulting logic program.

For program $\Pi_{TDS}$ from Figure \ref{fig:logic_program} set $\mathcal{C}$ of alphabet $\Sigma_{TDS}$ contains the dis-

\footnote{The underscore, i.e., “_”, indicates the use of a wildcard, i.e., during deduction of a solution, these parameters remain undbound, i.e., are “neglected”.
}
3.1 Model-based, Non-functional Security Testing of Web Applications

Figure 3.12: Program $\Pi_{\text{TDS}}$ reproduced in Prolog DCG notation. Prolog DCG notation more or less resembles the structure of BNF, a common notation for CFG and CSGs.

Figure 3.13: Extensions to program $\Pi_{\text{TDS}}$ to work for the four mentioned database vendors (MsSQL, MySQL, PostgreSQL and Oracle).

For the sake of clarity we also give the final set $\mathfrak{F}$ of predicates for program $\Pi_{\text{TDS}}$ (remember that $\mathfrak{F}$ as to Definition 3.3 initially only contained $\text{testdata}$), that is

$$\mathfrak{F} = \{\text{testdata}, apostrophe, number, connects, equals, comment, or\}.$$
such a VKB is established, i.e., filled with knowledge, but also how to maintain it, i.e., keep the knowledge up-to-date. As mentioned in the introduction, our method makes non-functional security testing feasible for “lays” in security, i.e., necessary security vulnerability knowledge is codified into our method. This implies however, that such knowledge at some point needs to be codified. This is where a security expert comes into play, i.e., by establishing the knowledge of the EDB and necessary test data definitions. Thus, our method still requires a security expert, however, not anymore for the tedious and labor-intensive task of test design and testing, but instead only for once codifying necessary security vulnerability knowledge and test data definitions in our VKB and further, keeping it up-to-date periodically. Observe that the tasks of establishment and maintainability only arise for the EDB and test data definitions but not for the IDB as it is fully provided. As long as the alphabets of the security problem SP, the EDB and the risk profile RP (Section 3.1.3) do not change, the IDB also must not be modified.

Codification of security vulnerability knowledge for the EDB follows directly from applying the predicates as introduced in definitions 3.2 and 3.3 from sections 3.1.2.1 and 3.1.2.3. For this, consider the case of codifying knowledge on SQLI and XSS vulnerabilities in web applications. In Section 2.1.4 we introduced different kinds of SQLI and XSS vulnerabilities that our method detects, viz. blind SQLI, SQLI by signature evasion and SQLI by incorrect type handling, as well as Type 1 and Type 2 XSS. Section 3.1.2.1 already showed the example of a “codified” SQLI by signature evasion in program EDB. Following we again discuss this idea of codifying knowledge by the example of Type 1 XSS. A more thorough discussion of the complete codified knowledge of the EDB for testing web applications follows in Chapter 4. The purpose of this discussion is to foster understanding of the subsequent sections.

Type 1 XSS vulnerabilities are severe in a sense that they (quite often) easily allow to gain control of numerous victims’ machines for different purposes. In order to do so, an attacker simply crafts a malicious piece of JavaScript code that subsequently is executed in a victim’s browser to, e.g., install a trojan or steal a cookie (Section 2.1.4). The only requirement an attacker has to a web application is some input field whose content (that was entered by the attacker) is not sanitized and reflected in a victim’s browser. Thus, the attack pattern could be formulated as to “search for any input field that accepts text input (i.e., JavaScript code)” as this opens up a potential attack vector for Type 1 XSS (observe that we cannot add a constraint for input sanitation as such knowledge initially is not known by an attacker; such is gained during attacking the application). With these few lines of thought and definitions 3.2 and 3.3 we now have everything at hand for codifying a Type 1 XSS vulnerability for our VKB. Similarly as in Section 3.1.2.1 we declare a small program EDB, that stores the just discussed idea of a Type 1 XSS
vulnerability, viz. $\Pi_{\mathcal{EDB}} =$

\begin{align*}
&\text{exploit}(\text{xss\_attack}), \\
&\text{attack}(\text{xss\_attack}, \text{reflected}, \text{xssap}, [\text{authentication}, \text{leakage}]), \\
&\text{vul\_type}(\text{xss\_attack}, \text{text}), \\
&\text{vul\_type}(\text{xss\_attack}, \text{hidden}), \\
&\text{attack\_pattern}(\text{xsslap}, \text{xss\_attack}, X_{\mathcal{DR}}, X_{\mathcal{D}}, X_{\mathcal{P}}) \leftarrow \text{module}(X_{\mathcal{DR}}), \\
&\text{operation}(X_{\mathcal{DR}}, X_{\mathcal{D}}, \ldots), \\
&\text{parameter}(X_{\mathcal{D}}, X_{\mathcal{P}}, X_{\mathcal{T}}), \\
&\text{vul\_type}(\text{xss\_attack}, X_{\mathcal{T}}).
\end{align*}

Program $\Pi_{\mathcal{EDB}}$ declares a typical Type 1 (or \textit{reflected}) XSS vulnerability that is, an attack that can lead to \textit{authentication} (install a trojan) or \textit{leakage} (steal a cookie). The attack vector is defined by the corresponding attack pattern ($\text{attack\_pattern}/5$) by considering input fields of type \textit{text} and \textit{hidden} (observe that hidden fields are HTML input fields that are just not rendered by the browser but present in the HTML code displayed in the browser, thus input can be specified for these fields prior to submitting a request to a web application). Similarly as then done in Section 3.1.2.3 for test data w.r.t. SQLI, we next define test data for XSS attacks\footnote{In the following we use the more concise and readable Prolog notation for test data.}. As stated earlier, XSS attacks use JavaScript to create the payload\footnote{For sure, JavaScript is not the only payload language, but the most common.}, i.e., the injection string. For example, to make the victim’s browser pop-up a window displaying the message “XSS” an injection string like

\begin{verbatim}
<script>alert("XSS");</script>
\end{verbatim}

suffices. Thus, we codify this “knowledge” as a DCG into our VKB which results in the listing from Figure 3.14. Observe that this DCG not only generates the starting tag \texttt{<script>} but also variations thereof, e.g., \texttt{<ScrIpt>} or \texttt{<script>ipt} to bypass potential input sanitation measures.

This example of establishing security vulnerability knowledge for the $\mathcal{EDB}$ shows that necessary knowledge can be codified in a straight-forward and, clear and concise manner following definitions 3.2 and 3.3. Further, the extensibility of the $\mathcal{TDS}$ allows for any kind of textual based malicious input data to be formulated by the use of DCGs and thus, does not limit a security expert in creating arbitrarily complex and malicious input data. The full $\mathcal{EDB}$ for testing web applications is shown in Appendix A and contains declarations for all five kinds of attacks discussed in Section 2.1.4 with symbolic names as listed in the following:

- SQLI by signature evasion as \textit{signature\_evasion},
- blind SQLI as \textit{blind},
3.1 Model-based, Non-functional Security Testing of Web Applications

Figure 3.14: Excerpt of the DCGs for generating XSS injection strings from Figure D.1 from Appendix D.

- SQLI by incorrect type handling as type_handling,
- Type 1 XSS as reflected, and
- Type 2 XSS as stored.

The full $\mathcal{TDS}$ for testing web applications w.r.t. the $\mathcal{EDB}$ from Appendix A is shown in Appendix D. The complete implementation of the $\mathcal{IDB}$ is shown in Appendix B.

In case of keeping the knowledge of the VKB up-to-date, various strategies are available, e.g., a security expert keeps a local copy of the VKB and corresponding test data definitions, updates them as necessary and distributes them to testers. Local copies of the VKB and test data definitions at the tester’s site are then replaced with the updated versions and testers can continue testing with updated knowledge. Automated distributing and updating of a VKB and corresponding test data definitions could be realized by, e.g., versioning [BR00]. Figure 3.15 shows such an automated update and merging scenario from the tester’s and security expert’s site. An initial VKB is provided by a security expert to a tester, i.e., VKB 1.0. After the security expert updates his local copy of the VKB to, e.g., version 1.1, he provides this updated version to the tester. At the tester’s site, the local version of the VKB, e.g., 1.0, is merged with the new version from the security expert, which then yields version 1.1 of the VKB. From a more technical perspective, the actual management and merging of different versions of a VKB (i.e., VKBs with differing knowledge) can be achieved by classic versioning and revision control systems like Subversion [CSFP04] or CVS [FB01] as of the text-based nature of
the contents of the VKB.

We provide implementations of the full $\mathcal{EDB}$ for testing web applications and the corresponding test data definitions in Appendices A and D. The complete implementation of the $\mathcal{IDB}$ is shown in Appendix B.

### 3.1.3 Risk Profile

The risk profile $\mathcal{RP}$ constitutes the output of the VKB, more precisely, its security risk analysis. It is a set of grounded instances of the $\text{risk}/7$ predicate, thus, another logic program, which contains only facts (like, e.g., the security problem $\mathcal{SP}$). Each of the grounded instances of the $\text{risk}/7$ predicate declares a potential risk, identified for a security problem $\mathcal{SP}$ during deduction of the risk profile $\mathcal{RP}$. At this, the terms formed with grounded instances of the $\text{risk}/7$ predicate range over elements of the sets of set $\mathcal{C}$ of alphabet $\Sigma_{\mathcal{RP}}$ as introduced by Definition 3.4, i.e., variables $X_i$ are grounded with elements of sets of set $\mathcal{C}$ of alphabet $\Sigma_{\mathcal{RP}}$.

**Definition 3.4 (Risk Profile).** The risk profile, $\mathcal{RP}$, describes a declarative risk model by a logic program. It is a set of facts with terms over the alphabet $\Sigma_{\mathcal{RP}} = ...$
\[\{\mathcal{F}, \mathcal{C}, \mathcal{X}\}, \text{ with disjoint sets of symbols}\]

\[\mathcal{F} = \{\text{risk}\} \text{ is a finite set of predicates},\]
\[\mathcal{C} = \{c_1, \ldots, c_m\} \text{ is a set of constants, and}\]
\[\mathcal{X} = \{X_1, \ldots, X_n\} \text{ is a set of variables}^{[1]}\]

Further, \(\mathcal{C} = \langle \mathcal{M}, \mathcal{O}, \mathcal{E}, \mathcal{A}, \mathcal{G}, \mathcal{P}, \mathcal{V} \rangle\), \text{ with disjoint sets of symbols, where}\n
\[\mathcal{M}\text{ is a set of modules,}\]
\[\mathcal{O}\text{ is a set of operations,}\]
\[\mathcal{E}\text{ is a set of exploits,}\]
\[\mathcal{A}\text{ is a set of attacks}\]
\[\mathcal{G}\text{ is a set of attack goals,}\]
\[\mathcal{P}\text{ is a set of parameters, and}\]
\[\mathcal{V}\text{ is a set of ordered risk assessment value triplets in the form of (impact, probability, risk level)}\]

where all of the just mentioned sets, except \(\mathcal{V}\), are equal with their identically labeled counterparts from Definitions 3.1 and 3.2.

The predicate described by set \(\mathcal{F}\) is a function over \(\mathcal{C}\) with

\[\text{risk} \subseteq \mathcal{M} \times \mathcal{O} \times \mathcal{E} \times \mathcal{A} \times \mathcal{G} \times \mathcal{P} \times \mathcal{V}\] to declare a risk.

As Definition 3.4 states, instances of the \text{risk/7} predicate in the risk profile \(\mathcal{R}\) are grounded with constants declared by both, the \(\mathcal{S}\) and the \(\mathcal{E}\) by applying deduction rules, declared in the \(\mathcal{I}\). Hence, it is “learned” by our VKB from the the \(\mathcal{S}\) and the \(\mathcal{E}\) and comprises new knowledge. For that this knowledge is sound and complete, we have defined the disjoint subsets of the sets of constants \(\mathcal{C}\) of alphabets \(\Sigma_{\mathcal{S}}, \Sigma_{\mathcal{E}}\) and \(\Sigma_{\mathcal{R}}\) from Definitions 3.1, 3.2, and 3.4 to be equal (Definition 3.4) to establish the domain of discourse.

Program \(\Pi_{\mathcal{R}}\) shows a risk profile \(\mathcal{R}\) as inferred by the security risk analysis using knowledge declared by programs \(\Pi_{\mathcal{S}}\) and \(\Pi_{\mathcal{E}}\) and the deduction procedure

\[^{[1]}\text{Again, variables are necessary for that terms can occur ungrounded.}\]
as described by program $\Pi_{IDB}$ (Figures 3.8-3.11), e.g., $\Pi_{RP} =$

\[
\text{risk(auth, login, sql\_attack, signature\_evasion,}
\]
\[
[\text{authentication, leakage, tampering}],
\]
\[
\text{uname, [high, high, high]}),
\]
\[
\text{risk(auth, login, sql\_attack, signature\_evasion,}
\]
\[
[\text{authentication, leakage, tampering}],
\]
\[
\text{pword, [high, high, high]}).
\]

Program $\Pi_{RP}$ describes a risk profile for the security problem $SP$ from program $\Pi_{SP}$. It contains two grounded instances of the risk/7 predicate, describing the risk of SQLI attack by signature\_evasion due to both parameters of operation login of module auth, i.e., uname and pword. Potential goals, among others, are authentication or leakage. For both risks, the impact, probability and risk level are high. This stems from the fact that they both entail the same exploit and attack, e.g., sql\_attack and signature\_evasion.

For program $\Pi_{RP}$, set $C$ of alphabet $\Sigma_{RP}$ contains the disjoint sets

- $M = \{\text{auth}\}$,
- $O = \{\text{login}\}$,
- $E = \{\text{sql\_attack}\}$,
- $A = \{\text{signature\_evasion}\}$,
- $G = \{\text{authentication, leakage, tampering}\}$,
- $P = \{\text{uname, pword}\}$, and
- $V = \{\text{high, high, high}\}$.

The deduced risk profile $RP$ does not explicitly subsume any negative requirements, but rather describes testing scenarios that may violate a negative requirement (Section 2.1.5.2) by a risk and its subsumed threat (Section 3.1.2.2). Thus, our VKB and its security risk analysis eventually target negative requirements although they are not explicitly established but instead expressed by the risks of the risk profile $RP$, i.e., scenarios that may violate a security property of the SUT, e.g., a login mechanism is bypassed by some kind of SQLI.

### 3.1.4 Test Generation

An advantage of our method is that we avoid additional effort for explicit generation of discrete, executable test cases in some target language (e.g., Java). This is due to our risk profile $RP$, a normal logic program, which describes an executable specification, i.e., we use the risk profile $RP$ as an executable test specification. This stems from that specifications, stated in Prolog, are executable \[\text{Den91}\]. When calculating the solution for a query, Prolog internally establishes a search tree which can be put
on a level with the control flow of a program. By tracing each “decision” Prolog takes during deduction, i.e., each currently selected goal, one can derive “execution traces”, a common source for test cases. In our case however, we cannot execute the risk profile $\mathcal{RP}$ by Prolog, as Prolog lacks the necessary facilities to invoke programs not written in Prolog, e.g., a web application. Instead, we use a custom interpreter written in Scala, which provides the necessary facilities to invoke external programs of different sources easily, to “execute” a risk profile $\mathcal{RP}$, which is discussed next.

### 3.1.5 Test Execution

Remember that in our method we do not explicitly generate executable test cases, but instead rely on the risk profile $\mathcal{RP}$ as an executable specification. Thus, we use the risk profile $\mathcal{RP}$ as a main input to our test engine, as shown in Figure 3.16. Our test engine then generates in-memory test cases, i.e., executable test objects, which are executed against a SUT. Necessary test data is provided by the inference engine of the VKB based on the test data definitions. After evaluating the result of each test case our test controller returns with a test log.

![Figure 3.16: Abstract overview of test execution. The risk profile $\mathcal{RP}$ is executed against the SUT. For each test outcome, a verdict is derived during test evaluation. Test execution returns with a test log. Necessary test data is generated by the inference engine of the VKB based on the test data definitions.](image)

For concrete test execution, our test engine requires two configuration options to be set, viz.

--- _priority_ for prioritizing test execution, i.e., to define the necessary _risk_level_ for a risk of the risk profile $\mathcal{RP}$ to be executed by a test case, and
**3.1 Model-based, Non-functional Security Testing of Web Applications**

-- *selection* to set the test data selection strategy based on what was retrieved from the Prolog solver\(^2\), which is either *iterative*, i.e., select data in the order retrieved from the Prolog solver, *random*, i.e., select data randomly, and *all*, i.e., to execute the test case for all retrieved data strings.

Figure 3.17 shows the complete algorithm of our test engine. After receiving a risk profile \(\mathcal{RP}\) as an input, our test engine loops through all risks \(r\) of the risk profile \(\mathcal{RP}\). For each risk \(r\), the algorithm then first checks, whether \(r\)'s *risk level* is equal to or higher than the user defined *threshold*, i.e., *priority*. If it is, risk \(r\) is further processed by testing, otherwise it is discarded. For each selected risk \(r\), the test engine then first queries for test data (line 4). Next, prior to executing risk \(r\) as a test case, the number of executions is determined based on the *selection* parameter in the conditional block in lines 5-9. After that, test execution starts \(n\) number of times in line 10. By calling *invoke*, an in-memory test case is generated for the current \((m, o, d)\) triplet. This triplet consists of a module \(m\) and operation \(o\) of the current risk \(r\), and the full test data set \(d\) as retrieved from the Prolog solver. At this, *invoke* also handles data selection as configured via the *selection* parameter, as well as logging the invocation of the operation and which test data string from the data set was used. After *invoke* has returned with a result (line 11), the test engine next evaluates this result by calling *evaluate* (see test evaluation, discussed in the following Section), which then returns with a verdict for the current test case w.r.t. its outcome, which subsequently is logged (line 13). For evaluating a test case, the *evaluate* procedure requires the result returned from the SUT, i.e., a test case’s outcome, the executed attack \((a)\), and potentially achievable attack goals \((g)\). The algorithm returns with the complete test log in line 17.

The *invoke* procedure plays an important role during test execution. It facilitates invoking programs of arbitrary sources by making use of abstractions (e.g., reflection, networking and OS APIs) as provided by the Scala programming language, which we used to implement our test engine. For testing web applications, what is actually going on under the hood of our Invoker is that, given a module (with its URI), an operation, the HTTP method type\(^3\), and parameters with corresponding values, it automatically generates a web request, either *GET* or *POST* and sends it to the SUT using a URL, constructed from the module’s base URI and the operation’s name (and, in case of a *GET* request, the parameters and their values). One of the problems we ran into with using this Invoker was that at first most tests failed. However, we fairly quickly realized that this is because of session IDs. As most web applications make use of session IDs to distinguish users from each other, we first needed to obtain a valid session prior to actually executing test

\(^2\)Remember that Prolog returns with every possible solution.

\(^3\)This information is not contained in the declarative version of the risk profile \(\mathcal{RP}\), as it is useless at this point. Instead, it is contained in the DSL-based version of the risk profile \(\mathcal{RP}\) as discussed in Section 3.2.
3.1 Model-based, Non-functional Security Testing of Web Applications

parameter: Risk Profile $\mathcal{RP}$
result : Test Log $\mathcal{L}$

1: **procedure** $\text{TEST}(\mathcal{RP})$
2: for each $r \in \mathcal{RP}$ do
3: if $r \rightarrow \text{risk}.\text{level} \geq \text{threshold}$ then
4: $d \leftarrow \text{TEST.DATA}(r \rightarrow a)$
5: if $\text{selection} = \text{all}$ then
6: $n \leftarrow d \rightarrow \text{size}$
7: else
8: $n \leftarrow \text{executions}$
9: end if
10: for $n$ do
11: $\text{result} \leftarrow \text{INVOKER}(r \rightarrow m, r \rightarrow o, d)$
12: $\text{verdict} \leftarrow \text{EVALUATE}(\text{result}, r \rightarrow a, r \rightarrow g)$
13: $\text{LOG}(\text{verdict})$
14: end for
15: end if
16: end for
17: return $\mathcal{L}$
18: **end procedure**

Figure 3.17: Test execution algorithm.

cases for that we retrieved any output. We solved this problem by just performing a simple GET request on the base URI of the SUT prior to executing any test cases which resulted in that we got a valid session for executing test cases.

Figure 3.18 shows an example of test execution by performing an SQLI against a web application, i.e., the SUT. Our test controller emits an HTTP request with generated test data as values for the respective parameters, i.e., $\text{uname} = 1' \text{ OR } '1'=1' --$ and $\text{pword} = \text{foo}$. Upon receiving of the request the web application processes it and, given that it is vulnerable to SQLI, returns with data as it leaks from the back-end database. The response from the web application then further is evaluated by the test controller (Section 3.1.6).

3.1.6 Test Evaluation

The purpose of test evaluation is, for any executed test case, to check, whether its result is as expected, i.e., meets a predefined test oracle and then return with the corresponding verdict. However, one of the problems in non-functional security testing is that prior to executing a test case, i.e., an attack, one does not clearly know what will be the output of the SUT, as this depends on how well an attack is crafted. For example, consider again the case of an SQLI by evading a programmer’s intended query signature. A test case that tests whether this kind of SQLI is possible
3.1 Model-based, Non-functional Security Testing of Web Applications

Figure 3.18: Exemplary visualization of test execution in case of a successful SQLI. Our test controller essentially generates HTTP requests that are submitted to the SUT, i.e., a web application. After processing the request, the response is sent back to the test controller for further evaluation.

could return with at least four possible outcomes, viz.

1. arbitrary data from the database, if the attack succeeded,

2. an exception, if the attack did not succeed in its current form, but still, is possible in another form (e.g., with a different injection string), as user input obviously is not sanitized,

3. an error message, indicating that the input was not processed by the application, or

4. nothing, i.e., NULL, which would make it apparently impossible to derive a verdict for the test case (e.g., due to that an SQLI is not possible or the system crashed).

Clearly, these possible outputs show that one cannot state an unambiguous oracle then.

In our non-functional security testing method, test evaluation is done by applying monitoring techniques, i.e., we analyze the application’s output, and based on that decide a verdict. We argue that the consequences of performing an attack against a SUT manifest themselves in the output of the SUT. Possible values for a verdict are

- PASS, if a test case succeeded, i.e., we were able to monitor an expected outcome,

- FAIL, if a test case did not succeed, i.e., we were not able to monitor any potentially expectable outcome, or an error occurred, and

- INCONCLUSIVE if it cannot be decided, whether a test case passed or failed.

To derive such a verdict we have designed our method to be capable of dealing with four major attack goals (Section 2.1.4) with distinct outcome characteristics, viz.
• **authentication violation**, if the attack, performed by a test case, attempts to bypass authentication mechanisms; for this, we, e.g., monitor the outcome to not be empty (i.e., it must contain data), but also to not contain any error or exception messages (which would indicate, that the attack failed),

• **leakage**, if the attack, performed by a test case, attempts to bypass authorization mechanisms and access protected data; for this, we, e.g., monitor the outcome to not be empty (i.e., it must contain data), but also to not contain any error or exception messages (which would indicate, that the attack failed),

• **tampering**, if the attack, performed by a test case, attempts to bypass authorization mechanisms to alter protected data; for this, we, e.g., monitor the outcome to not contain any error or exception messages (which would indicate, that the attack failed), and

• **dos**, if the attack, performed by a test case, attempts to make the target unavailable (i.e., denial-of-service); for this, we try to measure a timeout, i.e., if the target is not responsive within a certain time frame (e.g., default network timeout), we assume the attack succeeded.

For deriving a verdict, our method first decides whether an SQLI or XSS attack has been performed during testing. Following, two decision procedures are described which then allow to infer whether an SQLI (Figure 3.21) or XSS attack (Figure 3.24) was successful.

**Detecting Successful SQLI Attacks**  Figure 3.21 shows the state machine that infers a verdict for an executed SQLI attack. It is designed to detect three variations of SQLI that are considered in our method, i.e., SQLI by signature evasion or incorrect type handling and blind SQLI (see also Figure A.1 in Appendix A). Upon receiving a response $r$ from the the SUT after executing an SQLI attack, the state machine enters the state $q_{SQLI}$. Next, based on whether an SQLI by (i) signature evasion, (ii) blind SQLI or (iii) SQLI by incorrect type handling has been executed, the state machine transitions to the corresponding state, viz. (i) $q_{SE}$, (ii) $q_B$ or (iii) $q_{TH}$. From each of these states, the state machine then enters one of the final states, viz. $q_F$, $q_P$ or $q_I$ corresponding to the three verdicts FAIL, PASS and INCONCLUSIVE. Following the conditions for deriving one of the three verdicts from states $q_{SE}$, $q_B$ and $q_{TH}$ are discussed.

To detect whether an SQLI by signature evasion was successful, i.e., the verdict is PASS, condition $c_2$ needs to be fulfilled. Thus, the response $r$ must not be null (it must contain data, i.e., $r \neq \text{null}$), $r$ must not contain any error messages ($r \neq$...
3.1 Model-based, Non-functional Security Testing of Web Applications

The table shows the necessary conditions for that transitions in the state machine can be made.

<table>
<thead>
<tr>
<th>Label</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁</td>
<td>(\neg c₂ \land \neg c₃)</td>
</tr>
<tr>
<td>c₂</td>
<td>((r \neq \text{null}) \land (r \neq \text{error}) \land (r \neq o) \land (i \notin r))</td>
</tr>
<tr>
<td>c₃</td>
<td>((r = \text{null} \lor r = o))</td>
</tr>
<tr>
<td>c₄</td>
<td>(\neg c₅ \land \neg c₃)</td>
</tr>
<tr>
<td>c₅</td>
<td>(c₂ \lor (d \geq t))</td>
</tr>
</tbody>
</table>

Legend:
- \(d\) duration of test execution
- \(t\) timeout
- \(i\) input data
- \(o\) original page that received the request
- \(r\) submitted request

Figure 3.21: State machine for detecting successful SQLI attacks. The table shows the necessary conditions for that transitions in the state machine can be made.

error\(^{16}\), its contents must differ from those of the page \(o\) where the inject was originally submitted to \((r \neq o)\) and, finally, the inject must not be mirrored in the response \((i \notin r)\). This is in accordance with our declaration of SQLI by signature evasion in the \(\mathcal{EDB}\) from Figure A.1 in Appendix A. Potential goals of this attack are authorization, leakage or tampering. They all require (i) data (different to the original page) to be returned from the SUT and (ii) no thrown error. In case

\(^{16}\)We put a special focus on SQL specific error messages, e.g., “Incorrect syntax near” or “Unclosed quotation mark”.
the response \( r \) is null (\( r = \text{null} \)) or the same page is displayed (\( r = o \)), i.e., \( c_3 \) is satisfied, the verdict is \text{INCONCLUSIVE}. The fact that no error has been thrown, and further, the input apparently has been consumed, prohibits to derive \text{FAIL} as a verdict. However, if both \( c_2 \) and \( c_3 \) are unsatisfiable, the verdict is \text{FAIL} (\( c_1 \) is satisfied).

To detect a successful blind SQLI, more or less the same rationale as described for SQLI by signature evasion applies. The only difference is that for a successful attack, i.e., the verdict is \text{PASS}, we also consider whether the SUT does not respond within a given timeout, i.e., \( d \geq t \) (where \( d \) is the duration for the response and \( t \) a predefined timeout). This is due to our test data definitions for testing web applications (Figure D.1 in Appendix D) that also reflects injection strings that attempt to trigger a delay in the SUT. Hence, our tool also measures the execution time of test cases to then derive whether a blind SQLI attack was successful by provoking a denial of service (\text{dos}). To summarize, for that a blind SQLI is successful the same conditions as in case of \( c_2 \) must hold with the addition that we also consider time delays, i.e., \( c_5 \). As in case of SQLI by signature evasion, the verdict is \text{INCONCLUSIVE} if \( c_3 \) is satisfied. Finally, if \( c_4 \) is satisfied, the verdict is \text{FAIL}.

For SQLI by insecure type handling, the same rationale as for SQLI by signature evasion applies, i.e., if \( c_2 \) is satisfied, the attack succeeded, i.e., the verdict is \text{PASS}. If the response is null or equals the original page (\( c_3 \)) the verdict is \text{INCONCLUSIVE}. Finally, if either \( c_2 \) or \( c_3 \) are satisfied, the verdict is \text{FAIL}, i.e., \( c_1 \) is satisfied.

The reason why we check the response \( r \) not contain the injection string is by virtue that this would not be the case in a successful SQLI attack. Instead, we would expect different content as of data leaking from the database or accessing a protected area of the application, but not the reflected inject. Note that this also lets us clearly distinguish whether an operation is vulnerable to SQLI or XSS. If, for example, our tool executes an SQLI attack against an operation vulnerable to reflected XSS attacks (which happens due to the similarity of the attack patterns for SQLI and XSS; see the \text{EDB} from Figure A.1 in Appendix A) and the application immediately mirrors the injection string, this could mislead to infer that the operation is vulnerable to SQLI as of different content (due to the mirrored injection string). Searching the application’s response explicitly to not contain the injection string resolves this ambiguity.

In case that the operation is vulnerable to stored XSS attacks, our tool might also classify this vulnerability as an SQLI vulnerability. This is due to the above decision criteria for successful SQLI attacks For the most part, in case of a stored XSS vulnerability, the application does not immediately mirror the injection string. Thus, if our tool performs an SQLI attack against an operation vulnerable to stored XSS attacks, the application throws no error and further, the content has changed (but does not mirror the injection string), then our tool also identifies an SQLI vulnerability for the operation. This is sound due to that both, stored XSS (at least for placing the injection string) and SQLI attacks abuse databases. If stored XSS works, SQLI at the most does as well.
Detecting Successful XSS Attacks  Figure 3.24 shows the state machine that infers a verdict for an executed XSS attack. It is designed to detect two variations of XSS that are considered in our method, i.e., reflected and stored XSS (see also Figure A.3 in Appendix A). Upon receiving a response \( r \) from the the SUT after executing an XSS attack, the state machine enters the state \( q_{XSS} \). Then, based on whether (i) reflected XSS or (ii) stored XSS has been executed, the state machine transitions to the corresponding state, viz. (i) \( q_R \) or (ii) \( q_S \). From each of these states, the state machine then enters one of the final states, viz. \( q_F \), \( q_P \) or \( q_I \), again corresponding to the three verdicts FAIL, PASS and INCONCLUSIVE. Following the conditions for deriving one of the three verdicts from states \( q_R \) and \( q_S \) are discussed.

Detecting a successful reflected XSS attack is fairly simple. Given that the SUT returns data (\( r \neq \text{null} \)) that does not contain an error (\( r \neq \text{error} \)) and further, reflects the injection string (\( i \in r \)), i.e., \( c_2 \) is satisfied, then the verdict is PASS and a reflected XSS vulnerability has been detected. If \( c_2 \) however cannot be satisfied, the verdict is FAIL, i.e., \( c_1 \) is satisfied. Remember that in case of reflected XSS there is no verdict of type INCONCLUSIVE, as this cannot occur. Either the injected string is reflected in the response or not.

The situation is a great deal more complicated if about to detect a successful stored XSS attack. This is by virtue that code injected during a stored XSS attack sinks into a database table. Thus for verifying that an attack was successful, the source code of the application needs to be analyzed to identify calling locations w.r.t. the injected code, i.e., where it is loaded from the database and subsequently displayed in a victim’s browser. Apparently, our method and its tool implementation then are not capable to unambiguously decide on the success of a stored XSS attack. However, this is not true. Using an elaborate inference scheme we can, at least for the most part, decide whether a stored XSS attack was successful or not.

First of all, for inferring the verdict in case of a stored XSS, we need a second response \( (r_1) \) from the SUT, i.e., we generate another request using the same URL as used during testing (for sure, this time without parameter values). This happens in state \( q_S \). To then infer whether the stored XSS attack was successful, i.e., \( c_4 \) is satisfied and the verdict is PASS, the first response must not be null (\( r \neq \text{null} \)), contain no error message (\( r \neq \text{error} \)) and further, the second response must reflect the injection string (\( i \in r_1 \)). If this is the case, our method unambiguously detects a successful stored XSS attack. If, however, the injection string is not contained in the first and second response ((\( i \not\in r \) \&\& \( i \not\in r_1 \))) and the first response further does not contain an error (\( r \neq \text{error} \)), the verdict is INCONCLUSIVE, i.e., \( c_5 \) is satisfied.

One cannot exclude the possibility of an existing stored XSS vulnerability, just due to that the injection string was not reflected at all. It may be so at some other point in the application (e.g., loaded into a different page’s content). Observe that this is different to the case where only the response from testing, i.e., \( r \), reflects

\[17\] Remember that our tool implementation, like our method, has a black-box view of the SUT.
the injection string \(((i \in r) \land (i \notin r_1))\). This implies that our injection string was reflected immediately which indicates the presence of a reflected XSS vulnerability but not a stored. In such a case the verdict obviously is \textbf{FAIL}, as the right part of \(c_3\) is satisfied. Clearly, if \(c_4\) and \(c_5\) both cannot be satisfied, i.e., \(\neg c_4 \land \neg c_5\), the verdict also is \textbf{FAIL} as the left part of \(c_3\) is satisfied.

After test evaluation is done, our tool generates a test log and test feedback into the risk profile \(\mathcal{RP}\) (Section 3.2).
One might argue now that such an evaluation procedure may yield false positives and false negatives. However, to counterattack such false positives or negatives, we motivate not to execute a test case only once but rather multiple times, for that such false negatives and false positives can be levered out. To the best of our knowledge, such false positives and false negatives cannot be avoided completely, at least in automated test evaluation for non-functional security testing. Nevertheless, the evaluation of our method (Chapter 4) shows that our test evaluation procedure is effective by detecting successful attacks by monitoring the application’s response.

As for test execution, also for test evaluation we used Scala as an implementation language. This is due to necessary facilities to analyze outcomes of test cases w.r.t. certain characteristics which Prolog misses, e.g., efficient text processing and the like.

### 3.2 High–level Modeling Facilities

Our method described so far solely builds on logic programming (except for test execution and test evaluation). Consequently, our method relies on declarative models, i.e., logic programs. A common drawback in logic programming however is the rather outlandish syntax. Thus, the tool implementation of our method offers additional “high–level” modeling facilities by DSLs. This is motivated by the circumstance that the use of DSLs helps to develop readable and maintainable models with little effort. Further, as non-functional security testing of web application is a problem that occurs often enough, it is well worthwhile to express instances of the problem, i.e., system and testing artifacts, as sentences in a simple language [GHJV94].

Following we introduce the two DSLs of our tool, viz. the system DSL (SDL; Section 3.2.2) and the risk DSL (RDL; Section 3.2.3). Semantically, our DSLs express the same as their declarative counterparts, viz. the security problem $\mathcal{SP}$ and the risk profile $\mathcal{RP}$ as to Definitions 3.1 and 3.4. However, their syntax partly differs from their declarative counterparts. This is as to Scala’s rather clumsy syntax compared to Prolog. Albeit Scala provides quite a lot of “syntactical sugar”, it cannot beat the syntactical conciseness of a purely declarative language. We further present our web spider (Section 3.2.1) which helps in automatically establishing a security problem $\mathcal{SP}$. The web spider generates its own model of a SUT that is represented in XML. The choice to use XML as the fundamental representation of our system model is motivated by the fact that most model-based testing tools and approaches use XML (more precisely, XMI, a dialect of XML) as its storage and interchange format. Thus, our tool can easily be combined with features of model evolution [BBL10], or model-based regression testing [ZFKB12]. Finally, we discuss the necessary model translators which translate, (i) our XML based system model into a DSL-based system model (or security problem $\mathcal{SP}$) and, subsequently, its declarative counter part, and (ii) our declarative risk profile $\mathcal{RP}$ as it results from our security risk analysis into a DSL-based risk profile $\mathcal{RP}$. Figure 3.25 illustrates
3.2 High–level Modeling Facilities

this modeling infrastructure.

Figure 3.25: Overview of the modeling facilities of our tool implementation. A web spider establishes an XML-based model of the SUT, which then is translated into a declarative and DSL-based (linked to the XML-based model) security problem $SP$ by the SDL translator. The declarative security problem $SP$ then is used for a security risk analysis to generate a declarative risk profile $RP$. This declarative risk profile $RP$ finally is translated into a DSL-based representation by the RDL translator and linked to the DSL-based security problem $SP$.

As a final remark, for reasons of obviousness, we do not discuss each feature of the various model elements of our DSLs. For example, each element obviously has a name feature, however, we do not show them in our abstract syntax to keep it lucid. Further, we followed the convention to start names of model elements and (depicted) features with uppercase letters. In our concrete DSL examples however, we sometimes used lowercase letters instead of uppercase letters which is due to Scala’s syntax rules.

3.2.1 Web Spider

A web spider is a bot that performs automated web indexing, i.e., it systematically searches content of websites and indexes those very sites for later processing. For example, search engines make use of web spiders to create their index on which they perform a search. However, web spiders may also be used with a different intention, viz. to get a blueprint of a web application by user-accessible services (or operations). Put another way, we use a web spider to get a model of a web application by its user-accessible services and their parameters. While establishing such a model, the spider starts with the base URL of the application (i.e., its front page) to search for both, HTML input forms (this is where a user can access application functionality) and follow-up links. If the spider comes across an HTML input form, relevant information (e.g., module path, operation name, and the like) is extracted and stored inside the model of the SUT (see below; Section 3.2.2). If the spider however comes across a follow-up link, it first checks whether following the link would leave the domain of the application and, if not, add this link to the set of follow-up links.
3.2 High–level Modeling Facilities

...to process after processing of the current link and its contents is done. As soon as there are no more links to follow the spider returns with an XML-based model of the SUT describing its user-accessible functionality. We did not implement the spider by ourselves but instead used crawler4j [Gan13], a Java library which implements a web spider. What however was left to us in instantiating our custom web spider was to programatically set up its configuration by declaring (i) what to look for (e.g., the just mentioned HTML forms) and (ii) setting up the search strategy by what links to follow.

A common drawback in using such web spiders is that they only work for static applications, i.e., web applications which dynamically generate content and follow-up links cannot be crawled with such an automated spider. In such a case, one needs a proxy-based spider, i.e., a web spider that is used as proxy by the browser while the user is manually crawling the application. Although this imposes a quite laborious task on a tester, one still can retrieve an automatically generated model (which however is not automatically established).

At the time of this writing our tool’s web spider is capable of automatically establishing a model for applications written in PHP/SQL. However, extending it for applications implemented using other technologies, e.g., .NET or JSP, reduces to a case of how to configure the spider.

3.2.2 System Description Language

The System Description Language (SDL) formally declares syntactical modeling elements for describing a SUT (or security problem $SP$) in a textual manner. At this, our SDL is tailored to a black-box view of web applications. Hence, the SDL is rather small, as it must only provide elements that we can firmly instantiate with sound knowledge on the SUT, e.g., operation names and their parameters, or content folders of the web application (subsequently referred to as Module in our tool). Figure 3.26 shows the abstract syntax (or metamodel) of our SDL.

According to the abstract syntax from Figure 3.26, the SUT element is the root of a system model (or security problem $SP$). It declares a new SUT by a web application. Further, for each SUT, our model allows to define modules. The idea of a module is to represent a “content folder” of a web application. This is by virtue of that content folders are what software engineers refer to as a package. Packaging mechanisms allow for separation of concerns and modularization in applications written in high-level languages like Java just as do content folders in web applications. Thus our concept of a Module. Each Module then offers a variety of operations (or services, to use the more common term in web applications) which describe user-accessible functionality. These operations are declared using the Operation model element whereas each declared operation has a return type and an associated Method element. This element insofar is important as it specifies the type of HTTP method.

18Content folders may contain images or other resources, web sites or further content folders.
3.2 High–level Modeling Facilities

Figure 3.26: Abstract syntax of our system description language (SDL). Our language for describing a web application as a SUT is kept small and concise. This results from that in black-box testing we do not know much about the system that we can state with certainty.

used by an operation (e.g., GET or POST). Operations may also have associated zero or more parameters. The parameters insofar are of a high relevance as they describe the various attack vectors, an attacker may exploit in performing an attack. Also, each Parameter has associated a concrete type declared by a Type model element. Figure 3.27 shows the security problem $\mathcal{SP}$ from program $\Pi_{SP}$ from Section 3.1.2 using our SDL.

Using such a DSL for describing a system model for testing advances the overall testing process. However, it imposes the burden of establishing such a model on a tester or system developer. To avoid this burden and also ease the process of establishing a sound model of a SUT our tool slots a web spider (Section 3.2.1) ahead of actually starting the testing process (e.g., the security risk analysis and subsequent activities).

3.2.3 Risk Description Language

Contrary to the SDL that declared a rather small syntax, the Risk Description Language (RDL) for describing security risks w.r.t. a SUT is more comprehensive. This is due to that models of our RDL (e.g., a risk profile $\mathcal{RP}$) describe both, a test suite as well as an executable specification for non-functional security testing. Further, our risk profile $\mathcal{RP}$ (as mentioned earlier) also describes a negative specification that is necessary for non-functional security testing. Figure 3.28 shows the abstract syntax of our RDL.

Similarly to our SDL, also the RDL formally declares a set of syntactical model elements, yet for describing security risks. The central element of our RDL is the Risk concept. It declares a distinctive security risk the SUT potentially faces as of
object WebApp_SP {
  val root = new SUT(name = "WebApp",
    modules = List(
      new Module(name = "auth", uri = "www.victim.com",
        operations = List(
          new Operation(name = "login", method = "GET",
            parameters = List(
              new Parameter(name = "username", type_ = "TEXT"),
              new Parameter(name = "password", type_ = "TEXT"),
              new Parameter(name = "Submit1", type_ = "Submit")
            ))
      )
    )
  )
}

Figure 3.27: DSL-based notation of the security problem $\mathcal{SP}$ from program $\diamond_{\mathcal{SP}}$ from Section 3.1.1.

possible side-effect functionality. A risk itself is a composition of the four elements Attack, SUT, Test, and ThreatProfile. The Attack element describes the actual Exploit and its concrete Manifestation by which type of attack is used, e.g., signature evasion or blind in case of SQLI. Next, the SUT element encapsulates necessary information regarding the SUT, e.g., which Operation of which Module, and which IntrusionPoint, i.e., vulnerable parameter, potentially is affected. The Test element declares test execution specific features, viz. the number of Executions for a risk by testing and the Outcome element which in fact is a list that stores test feedback for each test run by tuples of the test data sent to the SUT and the corresponding outcome. Finally, the ThreatProfile contains further information regarding threats coming along with the risk, i.e., the Goals one can achieve with an attack, e.g., authorization or leakage. Further, the ThreatProfile contains vital information regarding test execution prioritization by its RiskLevel feature.

For the sake of completeness, the ThreatProfile element also provides features for storing the calculated Impact and Probability of the attack, encapsulated by the risk’s Attack element. Figure 3.29 shows the risk profile $\mathcal{RP}$ from program $\Pi_{\mathcal{RP}}$ from Section 3.1.3 using our RDL. As mentioned earlier (Section 3.1.6), after test evaluation is done, test feedback is generated into the risk profile $\mathcal{RP}$. This is accomplished by just loading the risk profile $\mathcal{RP}$ via reflection to then store tuples containing both, test data and the corresponding test verdicts inside the Outcome element of Test elements of Risks in the in-memory risk profile $\mathcal{RP}$. After feedback generation is done, the in-memory risk profile $\mathcal{RP}$ is serialized back into a physical risk profile $\mathcal{RP}$. Figure 3.30 shows an excerpt of the risk profile $\mathcal{RP}$ from program $\Pi_{\mathcal{RP}}$ with sample test feedback.
Both our DSL were implemented using the Scala programming language. They follow the design principle that testers neglect any details of the target platform and other implementation specifics during modeling [HK07].

### 3.2.4 Model Translators

The model translators somehow occupy the role of a middleware within our tool. They implement necessary facilities to translate our models into different representations. For our tool, this is necessary at two sites, (i) to translate the XML-based model of our web spider into its DSL-based representation and, further, its declarative representation, and (ii) to translate the declarative representation of the risk profile $\mathcal{RP}$ into its DSL-based representation for testing.

**SDL Translator** The task of the SDL translator is to translate the XML-based system model that results from the web spider into a system model (or security problem $\mathcal{SP}$) using notions of our SDL. Further, as our method internally works on declarative models, or logic programs, the SDL translator also fulfills the task of...
3.2 High–level Modeling Facilities

object WebApp_RP {
    val risk1 = new Risk(1,
        new Attack(exploit = "sql_attack",
            manifestation = "signature_evasion"),
        new ThreatProfile(goals = List("authentication","leakage",
            "tampering"), impact = "high", probability = "high",
            riskLevel = "high"),
        new SUT(module = "webapp", operation = "login",
            intrusionPoint= "uname"),
        new Test(executions = 10, outcomes = Nil))
    }

    val risk2 = new Risk(2,
        new Attack(exploit = "sql_attack",
            manifestation = "signature_evasion"),
        new ThreatProfile(goals = List("authentication","leakage",
            "tampering"), impact = "high", probability = "high",
            riskLevel = "high"),
        new SUT(module = "webapp", operation = "login",
            intrusionPoint= "pword"),
        new Test(executions = 10, outcomes = Nil))
}

Figure 3.29: DSL-based notation of the risk profile $\mathcal{RP}$ from program $\diamond_{\mathcal{RP}}$ from Section 3.1.3.

translating our DSL-based security problem $\mathcal{SP}$ into a declarative representation.

RDL Translator Contrary to the SDL translator, the RDL translator is far more limited in its functionality. This stems from that it only must translate a declarative risk profile $\mathcal{RP}$ as it yields from our security risk analysis into our DSL-based representation using notions of our RDL. This translation step insofar is vital, as our test engine expects a risk profile $\mathcal{RP}$ in our RDL. The key argument to use a DSL-based model for test execution and not our declarative representation was that a declarative model is far too bulky to work on, especially when generating test feedback into the model. Also, models which are based on DSLs traditionally are easier to understand and discuss for a human being as compared to a declarative model.

We have implemented our model translators by Scala’s parser combinators [MPO08]. Scala’s parser combinators offer a declarative syntax for easy development of language parsers. At this, our parsers implement the visitor pattern, i.e., each node of the input model is visited. Further, for each node type we have implemented “expansion templates”. These expansion templates predefine textual content of the output model that can be instantiated with values from the input model. Now, during processing of an input model, our parser, for each visited node then
3.3 Formal Foundations

Following we give the formal foundations of our method using set-based notation. We further show that our method describes a valid testing system w.r.t. Gourlay \cite{Gou83}.

We consider a SUT as a set \( P \) of programs \( p \), as stated in Definition 2.3. For the sake of completeness, we further introduce a definition of a program \( p \) as a function over a set \( \mathcal{O} \mathcal{P} \) of operations \( op \) as stated in Definitions 3.5 and 3.6.

**Definition 3.5 (Operation).** \( \mathcal{O} \mathcal{P} \) is a set of operations with elements \( op_i \), i.e.,

\[
\mathcal{O} \mathcal{P} = \{ op : op \in \mathcal{O} \}
\]

with set \( \mathcal{O} \) as by Definition 3.1. Each operation \( op_i \) represents a fundamental unit considered as testable by our method.

**Definition 3.6 (Program).** A program \( p \) denotes a function over a set \( \mathcal{O} \mathcal{P} \) of operations with

\[
p \subseteq \mathcal{O} \mathcal{P}^n
\]
with
\[ \mathcal{OP}^n = \mathcal{OP} \times \mathcal{OP} \times \cdots \times \mathcal{OP} = \{(op_1, \ldots, op_i) : op_i \in \mathcal{OP} \land 1 \leq i \leq n\}\]
and
\[ p \in \mathcal{M},\]
with set \( \mathcal{M} \) as by Definition 3.1.

More informally, by Definitions 3.5 and 3.6 we define operations \( op \) to be the building blocks of programs \( p \), which in turn are the building blocks of a SUT (Definition 2.3). Like for any operation \( op \) and set \( \mathcal{O} \), also each program \( p \) has an equally labeled counterpart in set \( \mathcal{M} \) as by Definition 3.1, i.e., we consider grounded instances of the \textit{module} predicate equally to programs \( p \) by grouping a set of operations \( op \). Our method thus considers programs \( p \) equal to modules \( m \), i.e., \( \mathcal{P} \equiv \mathcal{M} \) (with \( \mathcal{M} \) as from Definition 3.1).

The purpose of executing a software test is to verify that a requirement, either positive or negative, has been implemented (obviously either properly or erroneously), i.e., a software test has an \textit{objective}. Remember that negative requirements state a security property that must hold for the SUT, e.g., “the login mechanism cannot be bypassed”, which, for the most part, is not testable directly. Thus, our risk profile \( \mathcal{RP} \) does not explicitly state any negative requirements but rather describes testing scenarios that may violate a negative requirement by a risk and its subsumed threat (Section 3.1.2.2). Thus, we define test objectives upon grounded instances of the \textit{risk} predicate, which ultimately describe such testing scenarios.

\textbf{Definition 3.7} (Test Objective). \( TO \) is a set of test objectives with elements \( to_i \), where \( to_i \) describes a testing scenario that may lead to the violation of a negative requirement, i.e.,
\[ TO = \{to: to \in \epsilon\} \]
with
\[ \epsilon \subseteq \mathcal{E} \times \mathcal{A} \times \mathcal{O} \times \mathcal{P},\]
with sets \( \mathcal{O} \) and \( \mathcal{P} \) as by Definition 3.1, and sets \( \mathcal{A} \) and \( \mathcal{E} \) as by Definition 3.2.

Each \( to \) thus addresses a negative requirement \( r^- \in \mathcal{R}^- \) denoted to \( \alpha r^- \) (read \( to \) “addresses” \( r^- \)), i.e., each \( to \) evaluates whether a negative requirement is violated or not.

Each test case, to be executable, requires test data to be sent as input to the SUT. At this, especially in non-functional security testing, a test datum plays a key role by dictating the execution of the SUT to effectively stimulate side-effect behavior, i.e., exploit a vulnerability.

\textbf{Definition 3.8} (Test Data). \( TD \) is a set of test data with elements \( td_i \), i.e.,
\[ TD = \{td: td \in \tau\},\]
with
\[ \tau \subseteq \mathcal{T}O \times \Sigma_{TD}, \]
with \( \Sigma_{TD} \) being the set with all obtainable test data by solutions to logic programs over alphabet \( \Sigma_{TDS} \) as by Definition 3.3.

Our security risk analysis returns as a result a set of instances of the \textit{risk/7} predicate. Remember that we mentioned that the risk profile \( \mathcal{RP} \), which contains those very predicate instances, is used as an executable specification for testing. Thus, we define a test case w.r.t. a test objective \( to \) upon a grounded instance of the \textit{risk/7} predicate.

**Definition 3.9 (Test Case).** \( \mathcal{T} \) is the set of all test cases with elements \( t \), i.e.,
\[ \mathcal{T} = \{ t : t \in \phi \} \]
with
\[ \phi \subseteq \mathcal{T}O \times \mathcal{P} \times \mathcal{T}D. \]
A test case \( t \) then is defined as the triple \( t = \langle to, p, td \rangle \).

**Example 3.1.** Consider the risk profile \( \mathcal{RP} \) from program \( \Pi_{\mathcal{RP}} \). It contains two grounded instances of the \textit{risk/7} predicate. The first instance states that “there is a potential risk for an SQLI attack by signature evasion on operation login of module auth by abusing the parameter uname”. From this, we then first construct the test objective \( to \), i.e., \( to = (sql\_attack, signature\_evasion, login, uname) \). Further, program \( p \) gets assigned auth and td data retrieved from program \( \Pi_{TDS} \), e.g., \( ' OR '1' = '1'-- \). We then can define a test case \( t \) following Definition 3.9 as \( t = \langle to, p, td \rangle \), e.g.,
\[ t = \langle (sql\_attack, signature\_evasion, login, uname), auth, ' OR '1' = '1'-- \rangle \]

**Definition 3.10 (Test Suite).** \( \mathcal{TS} \) is a set of test suites with elements \( ts \), i.e.,
\[ \mathcal{TS} = \{ ts : ts \in \phi(\mathcal{T}) \setminus \emptyset \}. \]
A test suite \( ts \) then is an \( n \)-tuple, or a countable, finite set of test cases \( t \), which must not be empty.

The risk profile \( \mathcal{RP} \), which results from our security risk analysis, after Definitions 3.3 and 3.10 then also describes a test suite \( ts \).

For that one can use a test case as a tool for judging whether a SUT adheres to a test objective \( to \), it gets assigned a verdict \( v \) after execution. Further, such a set
of possible verdicts must be finite and enumerable to assure that a verdict $v$, i.e., the evaluation of a test case, is unique.

**Definition 3.11 (Verdict).** $\mathcal{V}$ is a finite, enumerable set of verdicts $v_i$, i.e.,

$$\mathcal{V} = \{v: v = \text{PASS} \lor v = \text{FAIL} \lor v = \text{INCONCLUSIVE}\}.$$

**Definition 3.12 (Verdict Function).** $\psi$ is the verdict function assigning each test case a distinct verdict $v \in \mathcal{V}$, i.e.,

$$\psi \subseteq \mathcal{T} \times \mathcal{V}.$$  

$\Psi$ is the verdict function applying $\psi$ to each test case $t$ of a test suite $ts \in \mathcal{TS}$, i.e.,

$$\Psi \subseteq ts.$$  

Software tests need to be executed for that their associated verdicts are useful in judging the adherence of a SUT against any set of test objectives $\mathcal{TO}$. A specific test run is characterized both, by its underlying test case $t$ and the resulting verdict $v$ by evaluating the outcome of the test case.

**Definition 3.13 (Test Run).** $\mathcal{TR}$ is the set of all test runs with elements $tr_i$, i.e.,

$$\mathcal{TR} = \{tr: tr \in \psi\},$$

Our method naturally provides a set of test oracles by its observable attack goals (Sections 2.1.4 and 3.1.6). This set internally is defined as part of the IDB (see Appendix B). We thus define the set $\mathcal{O}$ of test oracles as follows in Definition 3.14.

**Definition 3.14 (Oracle).** $\mathcal{O}$ is a finite, enumerable set of oracles $o_i$, i.e.,

$$\mathcal{O} = \{o: o = \text{AUTHENTICATION} \lor o = \text{DOS} \lor o = \text{LEAKAGE} \lor o = \text{TAMPERING}\}.$$  

**Definition 3.15 (Oracle Function).** $o$ is the oracle function assigning each pair of a program $p$ and test case $t$ as set of test oracles $o_i$, i.e.,

$$o \subseteq \mathcal{T} \times \mathcal{P}$$

with

$$o \in \varphi(\mathcal{O}).$$
In his 1983 paper Groulay establishes a mathematical framework for the investigation of testing \([\text{Gou83}]\) as given following in Definition 3.16.

**Definition 3.16** (Testing system according to Gourlay \([\text{Gou83}]\)). A testing system is a collection \(\langle \mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok} \rangle\), where

- \(\mathcal{P}\) is a set of programs,
- \(\mathcal{S}\) is a set of specifications,
- \(\mathcal{T}\) is a set of tests,
- \(\text{corr} \subseteq \mathcal{P} \times \mathcal{S}\), denoting the correctness of a program \(p\) w.r.t. a specification \(s\), i.e., \(p \text{corr} s\), and
- \(\text{ok} \subseteq \mathcal{P} \times \mathcal{S} \times \mathcal{T}\), expressing that test \(t\) performed on program \(p\) is judged successful by specification \(s\), i.e., \(p \text{ok}_t s\),

with \(s \in \mathcal{S}\), \(p \in \mathcal{P}\), \(t \in \mathcal{T}\) and \(\forall p \forall s \forall t p \text{corr} s \implies p \text{ok}_t s\).

The latter term claims that if a program \(p\) is correct w.r.t. a given specification \(s\), then no test \(t\) for program \(p\) will ever fail w.r.t. specification \(s\), thus legitimating Gourlay’s testing system \([\text{Gou83}]\). Observe that this does not mean that a test case \(t\) cannot fail. If a test case \(t\) fails due to that that program \(p\) is not correct w.r.t. specification \(s\), then \(\forall p \forall s \forall t p \text{corr} s \implies p \text{ok}_t s\) must not hold. Contrary, if a test case \(t\) is supposed to fail, \(\forall p \forall s \forall t p \text{corr} s \implies p \text{ok}_t s\) however holds, as the outcome of \(t\) is pass as the test case \(t\) must fail. As a side note, \(\text{ok}\) roughly relates to what today generally is named a test oracle (Definition 3.14).

For our testing method to represent a valid testing system w.r.t. Definition 3.16 we need to show two things, viz. (i) we need two show that the necessary sets, viz. \(\mathcal{P}\), \(\mathcal{S}\) and \(\mathcal{T}\) are founded and (ii) with the necessary sets founded, we have to show that \(\forall p \forall s \forall t p \text{corr} s \implies p \text{ok}_t s\) holds.

In case of (i), we, by Definitions 3.6 and 3.9 have just shown that our testing method is founded upon sets which are equivalent to sets \(\mathcal{P}\) and \(\mathcal{T}\) from Definition 3.16. This leaves us with showing that our method also considers a specification \(\mathcal{S}\). At first, one might argue that we can reuse set \(\mathcal{S}\) of specifications as introduced in Definition 2.2. However, using this approach would ultimately invalidate our testing method. This is by virtue of that w.r.t. Definition 3.16 which states that if a program \(p\) is correct w.r.t. a specification \(s\), then no test \(t\) for program \(p\) must fail w.r.t. specification \(s\). However, as our method is designed to detect SQLI and XSS vulnerabilities in web applications, i.e., show that a program \(p\) is not correct w.r.t. a specification \(s\), \(\forall p \forall s \forall t p \text{corr} s \implies p \text{ok}_t s\) would not hold anymore if \(\text{corr}\) does not hold, i.e., testing becomes insignificant. Fortunately, our method establishes another set \(\mathcal{S}\) of specifications, viz. the risk profile \(\mathcal{R}\). Eventually, we can consider a
risk \( r \) as a specification \( s \) that specifies defective behavior of the SUT, i.e., an attack that is possible due to a vulnerability. Thus, our risk profile \( \mathcal{R} \mathcal{P} \) ultimately can be considered a set \( \mathcal{S} \) of specifications \( s \) that is equal to set \( \mathcal{S} \) from Definition 3.16. Hence, our method establishes the sets required w.r.t. Definition 3.16.

This leaves us with showing that our method also satisfies \( \forall p \exists s \forall t \ p \text{corr } s \implies p ok_t s \) for that it describes a valid testing system w.r.t. Definition 3.16.

As stated by Definition 3.16, \( ok \) expresses “that test \( t \) performed on program \( p \) is judged successful by specification \( s \)”. This behavior is described by our oracle function \( o \), which assigns each pair of test case \( t \) and test objective \( to \) a distinct test oracle, i.e., it decides, whether the test objective \( to \) was satisfied by test case \( t \) w.r.t. a specification \( s \). Remember that a test objective \( to \) is formed upon a risk \( r \) that we ultimately consider a specification \( s \). If a program \( p \) does not contain exploitable side-effect functionality, i.e., a test objective \( to \) cannot be satisfied, then a test case \( t \) fails, otherwise it passes. Our method thus satisfies \( ok \).

The purpose of \( corr \) is to aid in showing that a testing system is valid, i.e., it verifies that specifications \( s \) are met by programs \( p \). In other words, for our testing method to be valid, whenever \( corr \) holds, then also \( ok \) holds. If a program satisfies a test objective \( to \), i.e., contains exploitable side-effect functionality, then our testing method reveals that side-effect functionality\(^19\). However, this cannot be done formally but rather only be shown by evidence. We thus argue and show by evidence (Chapter 4) that our testing method satisfies \( \forall p \exists s \forall t \ p \text{corr } s \implies p ok_t s \). Hence it describes a valid testing system w.r.t. Definition 3.16 which detects vulnerabilities in programs with side-effect behavior.

\(^{19}\)The case where \( corr \) does not hold is irrelevant, as from something wrong, anything we can imply, whether true or false, yields true.
Chapter 4

Experimental Evaluation

The true method of knowledge is experiment.

William Blake

We have evaluated our method and its tool implementation for testing web applications by Damn Vulnerable Web Application (DVWA) \cite{Ran14}, which is a, as its name already indicates, damn vulnerable web application. It has been developed and is maintained by RandomStorm \cite{Ltd14} with the goal to provide a legal environment for security experts to test their skills or tools. DVWA contains a number of vulnerabilities, among them SQLI, XSS, or file inclusion (for a complete list see Section 4.1.1).

Following (Section 4.1) we briefly describe DVWA (Section 4.1.1) and then discuss the lab setup for our case studies (Section 4.1.2) and concrete case study execution (Section 4.1.3). Finally, we present the results we got (Section 4.2) when testing DVWA with the tool implementation of our method at two different security levels, viz. low (Section 4.2.1) and medium (Section 4.2.2).

4.1 Case Studies

The goal of our case studies is to show that non-functional security testing of web applications is feasible with our method and its tool implementation. Further, w.r.t. the formal foundations given at the end of the previous chapter (Section 3.3), we show that our method and its tool implementation are a valid testing system by discovering existing vulnerabilities in web applications.

4.1.1 Damn Vulnerable Web Application

Damn Vulnerable Web Application (DVWA) \cite{Ran14} is a PHP/MySQL web application that is damn vulnerable. Its main goals are to be an aid for security pro-
professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and aid teachers/students to teach/learn web application security in a classroom environment [Ran14]. It is developed by RandomStorm, a UK-based network security, vulnerability management and compliance company, focused on providing enterprise-level, proactive security management tools and services [Ltd14].

DVWA is designed to implement a number of vulnerabilities w.r.t. OWASP’s 2013 top ten list [OWA13]. OWASP’s 2013 top ten list is based on eight datasets from seven firms that specialize in application security, including four consulting companies and three tool/SaaS (Software-as-a-Service) vendors (one static, one dynamic, and one with both). The data spans over 500,000 vulnerabilities across hundreds of organizations and thousands of applications from different domains, e.g., e-banking, e-Health, or online shopping, i.e., any domain that makes use of web applications. The top ten items are selected and prioritized according to this prevalence data, in combination with consensus estimates of exploitability, detectability, and impact estimates [OWA13]. We however will not discuss any of these top ten items, as (i) relevant vulnerabilities for our method, i.e., ones we test for, were already discussed (Section 2.1.4.1), and (ii) this would go beyond the scope of this thesis. The interested reader is advised to study OWASP’s 2013 top ten list [OWA13] which also provides a concise discussion of the various vulnerabilities and exploitation techniques. The vulnerabilities implemented by DVWA are as follows:

- Brute Force Password Cracking
- Remote Code/Command Execution
- Cross-site Request Forgery
- Insecure Captchas
- File Inclusion
- SQLI
- Blind SQLI
- File Upload
- Reflected XSS
- Stored XSS

To make training skills and tools more interesting, DVWA also provides three security levels, viz. low, medium, and high. Depending on which security level is configured for DVWA, none, few or various security mechanisms (e.g., input sanitation) are running to avert attacks.
4.1.2 Lab Setup

For our case studies we have installed DVWA on a dedicated web server running PHP 5.5.8 and Apache 2.4.6 in our lab alongside a MySQL database, version 5.3.35. Our testing tool was running on a different client machine in the same network as the server machine. Further, the network between the client and the server machine was a closed subnetwork, i.e., no connections from the outside were possible, to keep interference with other network traffic as low as possible. Figure 4.1 shows this setup.

![Lab Setup Diagram](image)

Figure 4.1: Lab setup for our case studies. A client machine is running our testing tool. During testing our testing tool issues requests to a web server that are forwarded to DVWA, which in turn creates the response. A MySQL database is running alongside the web server on the server machine.

We deactivated DVWA’s login mechanism to ease test execution. This is by virtue of that otherwise our tool would have needed to successfully login every time we start a test run for that we get a valid session and subsequently receive data from DVWA. Of course, this could be circumvented by equipping our tool with a login module, which would handle authenticating our tool prior to testing. However, as such a feature is not always necessary we decided to just deactivate the login mechanism for testing. Besides, deactivating the login mechanism does not distort or influence our results in any way.

4.1.3 Case Study Execution

As a first step in executing our case studies, we used the web spider of our tool to establish a security problem $SP$. Figure 4.2 shows the reduced security problem $SP$ for DVWA. We deliberately chose to reduce the initial security problem $SP$ as returned from our web spider due to (i) our tool would fail to detect vulnerabilities for the remaining operations (in total 6; Section 4.1.1), e.g.,
4.1 Case Studies

/vulnerabilities/brute/# which implements a brute force vulnerability and (ii) we want to keep the discussion clean and focused. Further, we also do not show the corresponding declarative counterpart of the security problem $SP$. This is due to that for our tool implementation the SDL translator replaces all the names (i.e., textual strings) that occur in a DSL-based representation of a security problem $SP$, e.g., operation or parameter names, with hashed IDs (thus rendering it unreadable for a human being). Using hashed IDs we again can represent operations, parameters and the like as symbols (by their IDs), thus, representing something Prolog can reason about. Using plain strings would result in that Prolog could not reason properly anymore and, occasionally, our VKB would fail in delivering a risk profile $RP$.

```java
object DVWA_SP {
    val root = new SUT(
        modules = List(
            new Module(name = "dvwa", uri = "http://10.4.4.16",
                operations = List(
                    new Operation(name = "/vulnerabilities/sqli/#",
                        method = "GET", parameters = List(
                            new Parameter(name = "id", type_ = "text"),
                            new Parameter(name = "Submit", type_ = "submit")
                        )),
                    new Operation(name = "/vulnerabilities/sqli_blind/#",
                        method = "GET", parameters = List(
                            new Parameter(name = "id", type_ = "text"),
                            new Parameter(name = "Submit", type_ = "submit")
                        )),
                    new Operation(name = "/vulnerabilities/xss_r/#",,
                        method = "GET", parameters = List(
                            new Parameter(name = "name", type_ = "text"),
                            new Parameter(name = ", type_ = "submit")
                        )),
                    new Operation(name = "/vulnerabilities/xss_s/#",,
                        method = "POST", parameters = List(
                            new Parameter(name = "txtName", type_ = "text"),
                            new Parameter(name = "btnSign", type_ = "submit"),
                            new Parameter(name = "mtxMessage", type_ = "text")
                        ))
            ))
        )
    }
```

Figure 4.2: Reduced security problem $SP$ as established by our web spider for the DVWA (clearly, the web spider did not reduce the model but we did manually).
The reduced security problem $\mathcal{SP}$ from Figure 4.2 states that DVWA offers four operations with potentially user-controllable parameters. At first, the names of the operations may seem a bit strange, however, this results from the structure and inner workings of web applications. Generally, web application developers enjoy the freedom of rejecting any design patterns or common standards (as, e.g., of the W3C) by just writing the application their way. This often results in that common implementation patterns, e.g., to use a question mark to indicate the start of parameters in a URL, are not used. Hence, we decided to store relative path names (from the base URI) up to the name of the requested action inside operation names (after all, this looks quite similar to the structure of OO applications with their packages). This way we ensure that requests generated by the Invoker always reach their proper destination. Note that DVWA uses the hash sign (#) as action names which triggers the form to submit to itself.

The remaining contents of the security problem from Figure 4.2 should be clear by now, each operation, besides its name, further declares its HTTP method type, i.e., GET or POST, as well as its list of parameters. Apparently none of the operations declares a return type as required by the abstract syntax of the SDL (Figure 3.26), yet this is not the case. PHP functions do not declare a return type, as they can return anything (even nothing). Thus, our SDL reflects this by setting an operation’s return type to None in which case it is not shown. Nevertheless, we decided to keep the return type in our SDL, as it is necessary again, when testing, e.g., applications written in Java or C#.

Chapter 3 introduced the VKB (Section 3.1.2), our expert system for performing a security risk analysis and risk assessment. There, we also already discussed small examples of knowledge for a VKB to detect SQLI by signature evasion (Section 3.1.2.1) and Type 1 XSS vulnerabilities (Section 3.1.2.4). For the purpose of our case studies we extended the codified knowledge of our $\mathcal{EDB}$. First, we defined the full set of attacks our methods should be capable of detecting as shown in Figure 4.3. Besides only containing an attack declaration for SQLI (sql\_attack) by signature\_evasion (as was the case for the $\mathcal{EDB}$ from program $\Pi_{\mathcal{EDB}}$) the $\mathcal{EDB}$ from Figure 4.3 now further contains declarations for blind SQLI and SQLI by incorrect type\_handling. Further, we declared a second exploit for XSS attacks, e.g., xss, and corresponding attacks, viz. reflected and stored XSS, respectively. Each of the attacks also specifies the relevant attack pattern, e.g., sqlap or xssap, and a list of potential attack goals (last parameter of each predicate).

For that our VKB can detect potential vulnerabilities in a security problem $\mathcal{SP}$ during a security risk analysis, necessary attack patterns need to be defined that are matched against operations of the security problem $\mathcal{SP}$. These patterns, viz. sqlap and xssap, which allow to identify potential SQLI vulnerabilities, al-
4.1 Case Studies

exploit(sql_attack).
exploit(xss).

attack(sql_attack,signature_evasion,sqlap,[authentication,leakage,tampering]).
attack(sql_attack,blind,sqlap,[authentication,leakage,tampering,dos]).
attack(sql_attack,type_handling,sqlap,[authentication,leakage,tampering]).
attack(xss,reflected,xssap,[leakage,authentication]).
attack(xss,stored,xssap,[leakage,authentication]).

Figure 4.3: Codified exploits and respective attacks for SQLI and XSS from our $\mathcal{EDB}$ for testing web applications.

ready were shown in Sections 3.1.2.1 and 3.1.2.4 as part of programs $\Pi_{\mathcal{EDB}}$ and $\Pi_{\mathcal{EDB}}$. Figure 4.4 again shows these attack patterns. The two attack patterns from

attack_pattern(sqlap,sql_attack,M,O,IP):-module(M),
operation(M,O,_,_),
parameter(M,O,IP,T),
vul_type(sql_attack,T).

attack_pattern(xssap,xss,M,O,IP):-module(M),
operation(M,O,_,_),
parameter(M,O,IP,T),
vul_type(xss,T).

Figure 4.4: Codified attack patterns to detect potential SQLI and XSS vulnerabilities from our $\mathcal{EDB}$ for testing web applications.

Figure 4.4 essentially describe the same logic, i.e., to search for operations that have user-controllable parameters of a certain type. The alert reader might now ask why we then did not use only one implementation of the attack_pattern/5 rule by replacing the second parameter, e.g., sqlap and xssap, respectively, with a variable. This however is not possible as otherwise Prolog’s solution strategy would match any attack on any operation where any attack pattern matches. Thus, we need to provide a concrete value for the second parameter of attack_pattern/5 to counter the problem of false positives that otherwise would be produced by Prolog, i.e., we need to restrict the search space.

As a last step of knowledge codification we declared parameter types, i.e., instances of the vul_type/2 predicate, commonly exploited by SQLI and XSS. Figure 4.5 shows these types. The listing from Figure 4.5 further shows an interesting aspect of Prolog, viz. the exclamation mark (!) in a rule’s body. The exclamation
mark essentially tells Prolog to stop searching for further solutions. For example, if Prolog’s inference engine is checking whether the type text is vulnerable to SQLI (sql_attack), the first occurrence of the vul_type/2 rule matches, thus Prolog’s inference engine would stop searching for possible further solutions. Obviously, if the first rule would not match, Prolog’s inference engine would continue to search for a valid solution or return with no solution if the rule cannot be satisfied at all.

Thus, the main reasons for using this cut operator (remember that it cuts off possible solutions from a search tree) is for reasons of efficiency. One however should be careful in using a cut operator, as it well could happen that it breaks any rule and no solutions at all can be found.

The complete EDB (with documentation) for testing web applications is shown in Figure A.1 in Appendix A.

We next executed our security risk analysis against the security problem SP from Figure 4.2. Figure 4.6 shows an excerpt of the risk profile RP for DVWA that resulted from our security risk analysis (the first three risks identified). The full risk profile (which spans four pages) is shown in Figure C.1 in Appendix C. It contains a total of 25 risks, where each of the risks describes either an SQLI or XSS attack (Attack) by trying to control a parameter of one of the four operations from the security problem SP as shown in Figure 1.2. Each risk further contains necessary information w.r.t. the SUT, and a ThreatProfile that contains required information w.r.t. test evaluation (Section 3.1.6) and the outcome of the risk assessment calculation. Observe that the risk assessment provides valuable information w.r.t. test prioritization (although we did no use it during our experiments, as for all risks, the same assessment was calculated; see below). Finally, each risk contains test execution related information (Test). To summarize this exemplarily for one risk, e.g., risk 1 (as shown in Figure 4.6) describes the potential threat of an SQLI attack by signature evasion against operation /vulnerabilities/SQLI/# by using parameter id as an intrusion point, i.e., parameter id opens up the attack vector. Further, the ThreatProfile specifies that an attack can have a high impact and probability, thus, also its overall risk level is high. Finally, risk 1 states that a resulting test case that evaluates the presence of the SQLI vulnerability shall be
executed ten times with test oracles as specified by the ThreatProfile’s goals. Analogously one can then interpret the knowledge stored in the remaining 24 risks.

```scala
object DVWA_RP {
  val risk1 = new Risk(1,
    new Attack(exploit = "sql_attack",
      manifestation = "signature_evasion"),
    new ThreatProfile(goals = List("authentication","leakage",
      "tampering"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation =
      "/vulnerabilities/sqli/#", intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  val risk2 = new Risk(2,
    new Attack(exploit = "sql_attack", manifestation = "blind"),
    new ThreatProfile(goals = List("authentication","leakage",
      "tampering","dos"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation =
      "/vulnerabilities/sqli/#", intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  val risk3 = new Risk(3,
    new Attack(exploit = "sql_attack", manifestation = "type_handling"),
    new ThreatProfile(goals = List("authentication","leakage",
      "tampering"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation =
      "/vulnerabilities/sqli/#", intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  ...
}
```

Figure 4.6: Excerpt of the risk profile $\mathcal{RP}$ for DVWA after executing our security risk analysis on the security problem $\mathcal{RP}$ from Figure 4.2.

Our risk analysis apparently successfully matches all attack patterns (sqlap and xssap) against any operation. This could lead to the conclusion that we just blindly match any attack pattern against any operation. Yet, this is not true. The reason for this apparent “blind matching” stems from the similarity in what SQLI and XSS exploits require. Consider again the attack patterns of as shown in Figure 4.4. Both attack patterns (sqlap and xssap) require a user-controllable parameter of at least one common type, e.g., text (Figure 4.5). Observe however further that none of the parameters of type submit (that occur in the security
problem $SP$ for DVWA in Figure 4.2 was detected as potentially vulnerable. This is by virtue that the type `submit` was not declared as one of the types vulnerable to SQLI or XSS. Hence, our security risk analysis does not perform a “blind matching”.

Intriguingly, the risk assessment as done for the risk profile $RP$ from Figure C.1 (and Figure 4.6) assigned the same assessment score for the risk level, as well as the same values for impact and probability, to each risk. Again, this also did not happen by choice or chance but by calculations during the security risk analysis, more precisely the risk assessment. Remember the `assess` procedure from Figure 3.9 that describes how the risk assessment is calculated. By now considering the calculation of the complexity value for an operation and the $IDB$’s assessment regarding the severity of the various attack goals, this similarity is easily explained. By virtue of the same complexity scores (due to identical operations) and partly identical attack goals, the scores for impact, probability, and risk level must not differ, as the calculations during risk assessment by definition achieve the same scores. As a side note, observe that the risk profile $RP$ contains a numeric value as risk level instead of any of low, medium or high. The use of numeric values simply allows to calculate more fine grained scores for the risk level, which is why we used them here.

### 4.2 Findings

Following we discuss the results when instrumenting our testing tool with the risk profile $RP$ from Figure C.1 and subsequently executing it against DVWA, first with the security level set to low (Section 4.2.1) and next with the security level set to medium (Section 4.2.2).

#### 4.2.1 Testing with Low Security Level

For our first case study the security level of DVWA was set to low and test data selection to iterative. A security level of low yields that DVWA has no protection mechanisms whatsoever, e.g., input sanitation, filter lists, and the like, in place. This results in that vulnerabilities are fairly easily detectable, and thus also exploitable. Hence, the demands put on the tool implementation of our method are quite high, i.e., to detect all those vulnerabilities in DVWA it knows about as to its $EDB$, viz. SQLI and XSS vulnerabilities. Figure 4.7 shows the results of executing the risk profile $RP$ from Figure C.1 against DVWA as installed and configured in our lab environment. Due to the contents of the risk profile $RP$, i.e., 25 risks with 10 executions each, a total number of 250 test cases were executed against the four operations from the security problem $SP$ for DVWA (Figure 4.2).

At first sight, the results shown in Figure 4.7 may be a little bit dizzying, as the verdict `FAIL` has been deduced for 124 test runs. Yet, this is desirable. Remember that our method is designed for non-functional security testing. Contrary
### 4.2 Findings

#### Risks

<table>
<thead>
<tr>
<th>Risk</th>
<th>Target</th>
<th>Executions</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk 1 - SQLI (signature evasion)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10 9 1 0</td>
<td></td>
</tr>
<tr>
<td>Risk 2 - SQLI (blind)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 3 - SQLI (type handling)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 4 - XSS (reflected)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 5 - XSS (stored)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10 0 4 6</td>
<td></td>
</tr>
<tr>
<td>Risk 6 - SQLI (signature evasion)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 7 - SQLI (blind)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 8 - SQLI (type handling)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 9 - XSS (reflected)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 10 - XSS (stored)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10 0 4 6</td>
<td></td>
</tr>
<tr>
<td>Risk 11 - SQLI (signature evasion)</td>
<td>/vulnerabilities/xss/# (name)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 12 - SQLI (blind)</td>
<td>/vulnerabilities/xss/# (name)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 13 - SQLI (type handling)</td>
<td>/vulnerabilities/xss/# (name)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 14 - XSS (reflected)</td>
<td>/vulnerabilities/xss/# (name)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 15 - XSS (stored)</td>
<td>/vulnerabilities/xss/# (name)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 16 - SQLI (signature evasion)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 17 - SQLI (blind)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 18 - SQLI (type handling)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 19 - SQLI (signature evasion)</td>
<td>/vulnerabilities/xss_p/# (mtxMessage)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 20 - SQLI (blind)</td>
<td>/vulnerabilities/xss_p/# (mtxMessage)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 21 - SQLI (blind)</td>
<td>/vulnerabilities/xss_p/# (mtxMessage)</td>
<td>10 0 10 0</td>
<td></td>
</tr>
<tr>
<td>Risk 22 - XSS (reflected)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 23 - XSS (stored)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 24 - XSS (reflected)</td>
<td>/vulnerabilities/xss_p/# (mtxMessage)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
<tr>
<td>Risk 25 - XSS (stored)</td>
<td>/vulnerabilities/xss_p/# (mtxMessage)</td>
<td>10 10 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.7: Results from test execution for our first case study with DVWA. The left side shows the risks with corresponding attacks and targets (with the exploited parameter), as identified during our security risk analysis, followed by the number of executions. The three columns on the right show the number of successful, i.e., PASS, unsuccessful, i.e., FAIL, and inconclusive, i.e., INCONCLUSIVE, test cases.

...to functional security testing, where a verdict of PASS is hoped-for to show that the SUT fulfills its specification, in non-functional security testing this is not the case. In non-functional security testing, the question is not anymore whether the SUT fulfills the specification but rather whether it contains exploitable side-effect functionality of some kind. Thus, if, in non-functional security testing, the verdict for a test run is FAIL, it means that the executed attack did not succeed, thus, the expected vulnerability (hopefully) does not exist (or, in the worse case, could not be detected). In case of PASS however, it means the SUT is vulnerable, as a vulnerability has been found an (already partly) exploited. This should be kept in mind during the discussion of the results of our case studies.

Risks 1 to 3 evaluated the possibility of an existing SQLI vulnerability (of all three types) for operation /vulnerabilities/sqli/#. From the 30 executed test cases, 29 successfully verified the existing vulnerability, yet one test case failed. After checking the test log, we found out that during this test run a single quote, i.e., “’” (remember, the most easiest SQLI exploit), was used as test data, which triggered an error in DVWA. Besides then knowing that the simplest exploit will not work, we however further know, that the application definitely has problems with injected quotes (which suggests the presence of an SQLI vulnerability). Fortunately however, the remaining 29 test runs successfully identified the existing vulnerability...
which does not leave any further room for speculations.

One could argue now that this is a false negative, however, an error/exception is not among the goals of an SQLI attack as to our EDB. Further, considering an error/exception as a successful or unsuccessful attack, would eventually increase the number of false positives and negatives. As an error can have many sources, i.e., failure in the network and the like, it could falsify our test results.

The failure of risk 4 for all ten executions is sound, as it describes a potential reflected XSS vulnerability for operation /vulnerabilities/sqli/# (remember the similarity of the attack patterns for SQLI and XSS attacks). Our tool successfully denied the existence of this vulnerability, which in fact, is the case, as it is not present in DVWA, and thus cannot be exploited.

Contrary, the test results for risk 5, which describes the threat of a potential stored XSS vulnerability for operation /vulnerabilities/sqli/#, are a bit more interesting. Our tool deduced FAIL four times as a verdict, and six times INCONCLUSIVE. This is because of that our tool cannot clearly decide on the outcome of a test if executing a stored XSS attack against some operation vulnerable to SQLI. Given that no error is thrown and the injected string is not contained in the response of subsequent requests, the verdict is INCONCLUSIVE. Thus, this is sound. The reason why four test runs failed is a result of the used XSS inject strings, which all contained single quotes at various positions. This leads to an error thrown by the application (as of the back end database). Thus, the results for risk 5 are also sound, as they indicate that no stored XSS vulnerability is present for operation /vulnerabilities/sqli/.

Executing risks 6 to 8, which describe potential SQLI vulnerabilities for operation /vulnerabilities/sqli_blind/# yielded exclusively PASS as a verdict. Our tool thus was able to detect the blind SQLI vulnerability in DVWA.

Risks 9 and 10 then evaluated whether operation /vulnerabilities/sqli_blind/# also contains a reflected or stored XSS vulnerability. In case of the reflected XSS attack (risk 9), all ten verdicts are FAIL, whereas in case of the stored XSS attack they are INCONCLUSIVE. For the reflected XSS attack, our tool deduced that the verdicts are FAIL due to that DVWA did not mirror any inject. In case of the stored XSS attack (risk 10), our tool hit its walls by not being able anymore to clearly state whether the stored XSS vulnerability exists or not. In such a case, further testing would be necessary.

The aim of risks 11 to 13 was to evaluate if operation /vulnerabilities/xss_r/# contains an SQLI vulnerability of any of the known types w.r.t. our tool’s EDB, viz. signature evasion, blind or due to incorrect type handling. An operation that is vulnerable to reflected XSS attacks immediately mirrors the inject. As this is the case for operation /vulnerabilities/xss_r/#, our tool thus could deny the presence of an SQLI vulnerability, as immediately mirroring the inject is among the criteria which exclude a successful SQLI attack.

Risk 14 then triggered execution of ten reflected XSS attacks against operation /vulnerabilities/xss_r/#. By our decision criteria for a successful reflected XSS
attack our tool could verify the presence of an existing reflected XSS vulnerability for operation /vulnerabilities/xss_r/# as the inject immediately is mirrored in the application’s answer. Risk 15 then triggered executing stored XSS attacks against operation /vulnerabilities/xss_r/#. Yet, all test runs failed, as to our decision criteria for a successful stored XSS attack, which requires the inject to be reflected in some subsequent request’s response. As this did not happen in case of operation /vulnerabilities/xss_r/# during testing (by virtue that it really only contains a reflected XSS vulnerability) our tool denied the presence of a stored XSS vulnerability.

For the next series of risks, viz. risks 16 to 21, our tool executed the known SQLI attacks against both parameters of operation /vulnerabilities/xss_s/#, viz. txtName and txtMessage. As operation /vulnerabilities/xss_s/# (despite containing a stored XSS vulnerability) immediately mirrors the inject, our tool again denied the existence of some SQLI vulnerability. Otherwise, given that the inject would not have been mirrored and further, no error would have been thrown by the SUT during testing, the verdict would have been PASS as to the decision criteria of our test evaluation.

The last four risks, viz. risks 22 to 25, then finally evaluated the possibility of reflected and/or stored XSS vulnerabilities for operation /vulnerabilities/xss_s/#, viz. risks 22 and 24 for reflected, and risks 23 and 25 for stored. Intriguingly, for all test runs we received the verdict PASS. This may sound disturbing in case of the reflected XSS attack, yet, as just mentioned, operation /vulnerabilities/xss_s/# immediately mirrors the inject, thus our tool also detects a reflected XSS vulnerability instead of only the stored XSS vulnerability. Nevertheless, those results are sound. Remember that for reflected XSS our tool checks whether the response contains the inject string. If this is the case, a reflected XSS vulnerability has been found. As operation /vulnerabilities/xss_s/# behaves this way, our tool strikes and also detects a reflected XSS vulnerability for operation /vulnerabilities/xss_s/#. In case of the stored XSS vulnerability, our tool also managed to detect it, as subsequent requests to operation /vulnerabilities/xss_s/# without parameters mirror the inject string from the test run.

The discussion of the test results for our first case study show that our method behaved as hoped and anticipated. It was able to detect all those vulnerabilities in DVWA it knows about, viz. XSS and SQLI vulnerabilities. Further, our tool also managed to, with a few exceptions, verify the non-existence of other, assumed (as to our security risk analysis) vulnerabilities for the various operations (e.g., SQLI for operation /vulnerabilities/xss_r/#). However, on the other side, the results also show that our tool soon hits its wall when it is about to detect stored XSS vulnerabilities. Yet, this is due to how XSS attacks work, e.g., to place an inject string somewhere in a database for that it is loaded later on. Thus, to truly decide on whether a stored XSS vulnerability exists, a tool must inspect the source code
of the SUT, i.e., check whether some input is mirrored to a victim at another point in the application. Without the availability of the SUT’s source code, one can only guess whether a stored XSS vulnerability exists. Our decision criteria as described during test evaluation reflect this guessing procedure. Although its results may not always be accurate, e.g., if the verdict is INCONCLUSIVE, they at least give us a good indicator, whether some operation should be tested further or not.

### 4.2.2 Testing with Medium Security Level

For our second case study the security level of DVWA was set to medium and test data selection again to iterative. Increasing the security level yields that DVWA has protection mechanisms running aiming at preventing attacks by mitigating potential vulnerabilities. To prevent SQLI attacks DVWA now, prior to submitting input to the database, sanitizes it using `mysql_real_escape_string`. This operation prepends a backslash to certain characters, viz. \x00, \n, \r, \, ', '" and \x1a to escape them. In case of XSS attacks, the prevention mechanisms are a little more versatile. First of all, the obligatory `<script>` tag, if present, is removed from any input string. Second, each input is processed by PHP’s `trim` function, which (obviously) removes leading and trailing whitespaces from the input. Third, the input is also processed by `htmlspecialchars` which transforms certain characters with special meaning in HTML, e.g., `<` or `>`, into their corresponding HTML entities, e.g., `<` and `>` in case of `<` or `>`. Finally, the input is also sanitized by `mysql_real_escape_string` prior to being processed by the application itself. With these set of prevention mechanisms, hacking DVWA immediately becomes catchier, as malicious input must not be in plaintext anymore but rather must be obfuscated to elude the protection mechanisms and achieve its goal.

Figure 4.8 shows the results for our second case study. Again, the risk profile $\mathcal{RP}$ from Figure C.1 was used, however (as mentioned earlier), this time the security level of DVWA was set to medium. We did not change the number of executions for any risk, thus again, a total number of 250 test cases were executed against the four operations from the security problem $\mathcal{SP}$ for DVWA (Figure 4.2). Following we again discuss the test outcomes, however, this time we focus on interesting test runs, viz. where the results differ from Figure 4.7. For the remaining results, the same rationale as discussed in case of the test results from Figure 4.7 applies.

As expected, the results for our second case study as shown in Figure 4.8 differ from those of the first case study, as shown in Figure 4.7.

Apparently, executing risk 1 resulted in the same test results as in case of our first case study. However, this is sound. Again, the one failing test case submitted a single quote, i.e., “’”, as input to the SUT. The SUT again triggered an error, although the input got sanitized, i.e., “\’”. Nevertheless, the resulting query still was not valid, as it just contained (although escaped) one single quote, which is not
valid SQL. As in case of our first case study, this indicates the presence of an SQLI vulnerability which then was verified by the remaining nine test cases.

The results we retrieved for executing risk $2$ are far more interesting. Contrary to our first case study, where the security level of DVWA was set to low, this time, only two of ten test runs were successful (remember that risk $2$ describes a potential blind SQLI vulnerability for operation /vulnerabilities/sqli/#). This is a result of the protection mechanisms, which now, at the most, sanitized the inject strings generated by our tool. Fortunately, it still could detect the vulnerability. What should be kept in mind at this point is that, from our point of view, there is no difference whether one or ten test runs succeeded. Each indicator for the potential presence of some vulnerability, despite how humble it may be, should be taken serious and pursued to assure that the vulnerability does not exist, or, as in case of DVWA, is mitigated properly.

<table>
<thead>
<tr>
<th>Risks</th>
<th>Target</th>
<th>Executions</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk 1 - SQL (signature evasion)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10</td>
<td>PASS 0 FAIL 9 INCONCLUSIVE 1</td>
</tr>
<tr>
<td>Risk 2 - SQL (blind)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10</td>
<td>2 8 0</td>
</tr>
<tr>
<td>Risk 3 - SQL (type handling)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10</td>
<td>10 0</td>
</tr>
<tr>
<td>Risk 4 - XSS (reflected)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 5 - XSS (stored)</td>
<td>/vulnerabilities/sqli/# (id)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 6 - SQL (signature evasion)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
<tr>
<td>Risk 7 - SQL (blind)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
<tr>
<td>Risk 8 - SQL (type handling)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
<tr>
<td>Risk 9 - XSS (reflected)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 10 - XSS (stored)</td>
<td>/vulnerabilities/sqli_blind/# (id)</td>
<td>10</td>
<td>0 0 10</td>
</tr>
<tr>
<td>Risk 11 - SQL (signature evasion)</td>
<td>/vulnerabilities/xss_r/# (name)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 12 - SQL (blind)</td>
<td>/vulnerabilities/xss_r/# (name)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 13 - SQL (type handling)</td>
<td>/vulnerabilities/xss_r/# (name)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 14 - XSS (reflected)</td>
<td>/vulnerabilities/xss_r/# (name)</td>
<td>10</td>
<td>6 4 0</td>
</tr>
<tr>
<td>Risk 15 - XSS (stored)</td>
<td>/vulnerabilities/xss_r/# (name)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 16 - SQL (signature evasion)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 17 - SQL (blind)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 18 - SQL (type handling)</td>
<td>/vulnerabilities/xss_p/# (txtName)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 19 - SQL (signature evasion)</td>
<td>/vulnerabilities/xss_e/# (txtMessage)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 20 - SQL (blind)</td>
<td>/vulnerabilities/xss_e/# (txtMessage)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 21 - SQL (blind)</td>
<td>/vulnerabilities/xss_e/# (txtMessage)</td>
<td>10</td>
<td>0 10 0</td>
</tr>
<tr>
<td>Risk 22 - XSS (reflected)</td>
<td>/vulnerabilities/xss_e/# (txtName)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
<tr>
<td>Risk 23 - XSS (stored)</td>
<td>/vulnerabilities/xss_e/# (txtName)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
<tr>
<td>Risk 24 - XSS (reflected)</td>
<td>/vulnerabilities/xss_e/# (txtMessage)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
<tr>
<td>Risk 25 - XSS (stored)</td>
<td>/vulnerabilities/xss_e/# (txtMessage)</td>
<td>10</td>
<td>10 0 0</td>
</tr>
</tbody>
</table>

Figure 4.8: Results from test execution for our second case study with DVWA. The left side shows the risks with corresponding attacks and targets (with the exploited parameter), as identified during our security risk analysis, followed by the number of executions. The three columns on the right show the number of successful, i.e., PASS, unsuccessful, i.e., FAIL, and inconclusive, i.e., INCONCLUSIVE, test cases.

In case of risk 5, this time, all test runs resulted in a verdict of FAIL (compared to our first case study where we also retrieved INCONCLUSIVE as a verdict). Interestingly, this results from the prevention mechanisms, especially `mysql_real_escape_string`, which sanitizes user input and thus triggers an error on executing the XSS injection string. Hence, if performing a stored XSS attack against operation /vulnerabilities/sqli/# our tool this time clearly verified that
the vulnerability is non-existent.

Regarding risk 14 we also retrieved different results compared to our first case study. Again, this due to the protection mechanisms which are now running in DVWA. Apparently, four test runs used input strings which were too trivial to elude any prevention mechanisms. Yet, our tool again still was able to detect the vulnerability as indicated by the six successful test runs.

For the remaining risks, as already mentioned earlier, the same rationale as for our first case study applies. For example, if considering risks 16 to 21, our tool again only deduced FAIL as a verdict. Yet, this is as expected, as the operation in question, viz. /vulnerabilities/xss_s/# still does no implement an SQLI vulnerability, thus it cannot be found.

The discussion of the results for our second case study again show the effectiveness of the tool implementation of our non-functional security testing method by logic programming. Our tool again was able to verify all existing vulnerabilities, yet, this time with a slightly lower success rate. However, this is as of DVWA’s increased security level which makes exploiting it more trickier. What is important is that our tool still could verify the vulnerabilities.

**False Positives and False Negatives** During our first case study, our method and its tool implementation produced a total of 16 false positives. Of these 16, 6 occurred when executing risk 5 and 10 occurred when executing risk 10. In both cases, these are the result from that our tool implementation did not clearly reject (remember the verdict, i.e., INCONCLUSIVE) the existence of a stored XSS vulnerability in operations /vulnerabilities/sqli/# and /vulnerabilities/sqli_blind/#, respectively. This is by virtue of our test evaluation that, in some cases, i.e., if executing a stored XSS attack against some operation vulnerable to SQLI, derives INCONCLUSIVE as verdict. Yet, in the event of risk 6, 4 test cases still were able to reject the existence of a stored XSS vulnerability, which suggests that it really does not exist (nevertheless, further investigation would be advised).

During our second case study, our method and its tool implementation produced a total of 12 false negatives and 10 false positives. The first occurrence of false negatives was for risk 2 with a total number of 8, i.e., the existence of an SQLI vulnerability for operation /vulnerabilities/sqli/# was rejected although it is existent. This is due to that our injection strings for blind SQLI were too weak to pass the protection mechanism of DVWA. Yet, as 2 of the 10 test cases verified the existence of an SQLI using techniques of blind SQLI, further investigation would be advised. The other 4 false negatives occurred when executing risk 14, i.e., a reflected XSS attack against operation /vulnerabilities/xss_r/#. Again, this is due to too weak injection strings that could not bypass the protection mechanisms that are in place in DVWA if setting the security level to medium. However, as 6 of the 10 test executions passed, our tool still could verify with noteworthy certainty the existence of a reflected XSS for operation /vulnerabilities/xss_r/#. 
4.2 Findings

<table>
<thead>
<tr>
<th>Case Study Nr</th>
<th>False Positives</th>
<th>False Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 4.9: Summary of false positives and false negatives for both our case studies with DVWA.

In case of the 10 false positives that occurred during execution of our second case study, as they occurred again when executing risk 10, i.e., a stored XSS attack against an operation vulnerable to SQLI, viz. /vulnerabilities/sqli_blind/#, the same rationale as for our first case study applies.

Figure 4.9 summarizes the number of false positives and false negatives that occurred during the two case studies with DVWA.
Chapter 5
Discussion

The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" but "That's funny..."

Isaac Asimov

On the grounds of the results from our case studies presented in the preceding chapter, in this chapter we give a thorough and evaluative discussion of our method. First, we interpret the findings from our two case studies and relate them to the research challenges raised in Chapter 1 (Section 5.1), followed by positioning our work w.r.t. state-of-the-art as discussed in Chapter 2 (Section 5.2). We conclude this chapter with an elicitation of the implications of our work (Section 5.3).

The research paradigm adopted in this dissertation is design science research [PTRC07, vAMPR04]. According to van Alan [vAMPR04], the main purpose of design science research “is achieving knowledge and understanding of the problem domain by building an application of a design artifact”. This essentially motivates the development of a prototype implementation of a novel method and its evaluation w.r.t. usability, effectiveness, and efficiency. We have evaluated our method and its tool implementation by two case studies using a vulnerable web application as discussed in the previous chapter.

Design-science is technology oriented with a two-fold purpose, viz. (i) to address specific research activities, i.e., building, evaluating, theorizing about, and justifying new methods and (ii) produce output by effective artifacts, i.e., representational constructs, models, methods, and instantiations [MS95]. We have developed a novel model-based testing method for non-functional security testing of web applications with the purpose to address current deficiencies of non-functional security testing, viz. its lack for structured and reproducible testing as well as the incorporation of necessary security vulnerability knowledge on the grounds of (i) knowledge engineering, (ii) logic programming and (iii) MBT. We further have
implemented our method by a tool that has been presented at an international information security conference [ZFKB14]. Additionally, the work presented in this thesis also resulted in a research article submitted to an international software engineering journal [ZFB14].

The principle-based nature of design theory requires at least two components for the definition of a design theory, (i) a set of user requirements, and (ii) the principles governing the development process [MMG02]. In our case, these components are provided by (i) the research challenges, i.e., the set of user requirements, and (ii) the contributions, i.e., the principles governing the development process. On the grounds of the outcomes of our case studies from the previous chapter, in the following (Section 5.1) we evaluate our contributions as outlined in Section 1.2 w.r.t. to the research challenges as defined in Section 1.1. Upon this, we further infer the implications of our work.

5.1 Interpretation of Findings

The results from our case studies clearly show that the application of logic programming and knowledge engineering for non-functional security testing is valuable in finding existing vulnerabilities in web applications by reasoning on potential vulnerabilities in a declarative model of a web application. Our VKB plays a primary role during this reasoning process by both, codifying the necessary security vulnerability knowledge and implementing the security risk analysis, i.e., the reasoning rules. Further, the search-based nature intrinsic to the calculation of a solution in logic programming achieves a high coverage by the resulting number of test cases for a SUT. This is undermined by the circumstance that during our case studies, all vulnerabilities in DVWA that our EDB contained, i.e., XSS and SQLI, were found and verified. Further, the risk assessment which is done as part of our security risk analysis provides valuable information regarding the severity of potential vulnerabilities and is auxiliary in doing actual testing by concentrating on serious vulnerabilities. We thus successfully accomplished research challenges RC1 and RC2. Research challenge RC1 addressed the codification of security vulnerability knowledge for non-functional security testing which is in line with research challenge RC2 that addressed an automated security risk analysis and risk assessment. This is achieved by contributions C1, i.e., our VKB, C2, i.e., our security risk analysis, C4, i.e., sets of logical predicates for declaring a security problem SP and a risk profile RP and C7, i.e., our tool implementation for model-based non-functional security testing of web applications.

The tool implementation of our method has shown to be valuable and effective as it detected all vulnerabilities it “knew” about by virtue of the EDB, i.e., XSS and SQLI. This suggests that non-functional security testing offers great potential
to be automated. Indeed, given that certain requirements are fulfilled, this is the case, as discussed in the following. First of all, necessary security vulnerability knowledge needs to be available by an expert system and further also accessible for testing. Second, a model of the SUT, i.e., a web application, needs to be available in a declarative format for that the just mentioned knowledge can be applied on it. Third, automated reasoning rules need to be implemented to efficiently derive test cases from the model of the SUT on the grounds of codified security vulnerability knowledge using search-based techniques. Fourth, a test engine needs to be implemented that allows to execute the just mentioned test cases against a SUT. Fifth, and finally, efficient mechanisms for detecting successful test cases, i.e., attacks, need to be designed. As the results of our case studies for our method and its tool implementation show, if these requirements are fulfilled then non-functional security testing for web applications can be automated in a highly-efficient, structured and reproducible manner. We thus successfully accomplished research challenge RC3 and RC4. Research challenge RC3 addressed automated, non-functional security testing of web applications to detect SQLI and XSS vulnerabilities and research challenge RC4 addressed automated generation, execution and evaluation of negative security test cases. We achieved accomplishment of research challenge RC3 by contributions C3, i.e., automatic generation of a test suite, C4, i.e., sets of logical predicates for declaring a security problem $SP$ and a risk profile $RP$, C5, i.e., a posteriori evaluation of test runs by test oracles, C6, i.e., successfully detection of SQLI and XSS attacks and C7, i.e., our tool implementation for model-based non-functional security testing of web applications. The accomplishment of research challenge RC4 is by virtue of contributions C1, i.e., our VKB, C2, i.e., our security risk analysis, C3, i.e., automatic generation of a test suite, C5, i.e., a posteriori evaluation of test runs by test oracles and C7, i.e., our tool implementation for model-based non-functional security testing of web applications.

During our case studies a number of false positives (13) and false negatives (16) occurred. At first sight, this might be irritating as it indicates that our method is partially inefficient. However, further consideration of the results from our case studies (Figures 4.7 and 4.8) reveals that these false positives and false negatives were always levered out with certain confidence by other test cases. Put another way, although not every test case succeeded in detecting or rejecting the presence of a vulnerability, the overall number of successful test cases compared to unsuccessful test cases (i.e., false positives and false negatives), that is 234 vs. 16 for our first case study and 228 vs. 22 for our second case study, indicates the effectiveness of our method. This further undermines our statement that we have successfully accomplished research challenges RC2 and RC3.

Another interesting aspect of our method, besides successfully detecting vulnerabilities in web applications, is the circumstance that it, as part of its test data generation and test execution, also delivers working exploits, i.e., ad-hoc
proofs-of-concept, for vulnerabilities in a SUT, i.e., a web application. By using DCGs our VKB generates efficient test data strings (or malicious injection strings) that are both, valuable in testing but also in showing that an exploit actually really works. This further undermines our conclusion that we have successfully accomplished research challenges RC1 and RC4.

Figure 5.1 summarizes our findings by picturing the relation of each research challenge to its respective, accomplishing contributions.

<table>
<thead>
<tr>
<th>Research Challenge</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC1</td>
<td>C1, C2, C4, C7</td>
</tr>
<tr>
<td>RC2</td>
<td>C2, C4, C7</td>
</tr>
<tr>
<td>RC3</td>
<td>C3, C4, C5, C6, C7</td>
</tr>
<tr>
<td>RC4</td>
<td>C1, C2, C3, C5, C7</td>
</tr>
</tbody>
</table>

Figure 5.1: Relation of research challenges and respective, accomplishing contributions.

Unfortunately, there is one constraint we have to accept w.r.t. our method, i.e., testing against stored XSS attacks. We already mentioned this in the previous chapter when discussing the false positives and false negatives that occurred during our case studies. For the sake of completeness, we will recap this constraint here. The tool implementation of our method hit its walls as soon as it executed a stored XSS attack against some operation vulnerable to SQLI. Given that no error is thrown and the response of the web application is identical to the page the request was submitted to, our tool cannot decide whether an attack was successful or not (Figure 3.24). This is by virtue of that it implements a black-box view of the system and thus cannot decide whether the inject is stored somewhere in the application or not. Thorough testing against stored XSS attacks requires a white-box testing approach, i.e., a view of all the application’s internals. Yet, this should not be seen as a disadvantage but rather a result from research challenge RC3, indicating that testing against stored XSS cannot always be automated due to the trickiness of this attack.

5.2 Positioning w.r.t. State-of-the-Art

Logic Programming As discussed in Section 2.2.1, logic programming has found its application in testing for two purposes, viz. test case generation [BM95, CGRSP10, Den91, GZAP10, GKMSP90, LP00, VK95] and test data generation [GBR98, JBW+94, Meu01]. This discussion however also shows that so far, logic programming has not been applied for the concrete case of security testing. In the context of security, logic programming has been applied for statically verifying (for the most part) cryptographic protocols [AB05, ACG+06, Bla01, Bla03, CAM01]. From our point of view, this does however not qualify as testing, as no test cases,
5.2 Positioning w.r.t. State-of-the-Art

either abstract or executable, are generated. Thus, our work is a valuable contribution in logic programming as it introduces a novel application of logic programming for security testing. In the event of test data generation, our method shows a further application of logic programming for this purpose, viz. using DCGs. Existing work in this area so far only applied constraint solving.

Knowledge Engineering in Security Engineering As in case of logic programming, also knowledge engineering, as discussed in Section 2.2.2, so far has not been applied in security testing. Yet, it has found various applications in security engineering in general. At this, knowledge engineering for the most part has been used for codifying vulnerability knowledge by different means, e.g., by attack trees or graphs [JNO05, LIS+06, RCSM09] and vulnerability signatures [AGI12]. The resulting codified knowledge then has been used for vulnerability analysis of networks or software to identify potential vulnerabilities in either. However, the results of those analyses have not further been exploited for testing. Besides those approaches there also exist a few expert systems for security engineering that are similar to our VKB [EZ01, OGA05, SEH13]. The purpose of these systems again is to identify potential flaws in networks, firewalls or software systems, yet using purely knowledge engineering and reasoning techniques. However, the outcomes are not used for testing. Our work thus is a valuable contribution in knowledge engineering as it introduces a novel application of knowledge engineering for non-functional security testing. Contrary to existing work, our expert system not only infers new knowledge for then discussing the security of a software system or network. Instead, it derives knowledge that is further used for evaluating a system by testing.

Penetration Testing In face of existing penetration testing techniques w.r.t. [BYCJ11] our method falls into the second category, i.e., application penetration testing whose goal is to target the logical structure of a software system, i.e., its programs and components [BYCJ11]. As common in penetration testing, the number of “real” testing approaches, i.e., approaches that follow some methodical design, for application penetration testing is rather humble [DZG08, HCO09, MSS12, May11]. These approaches either slot some kind of analysis, e.g., static or vulnerability analysis prior to testing [DZG08, HCO09] or provide canned plugins that implement some attack, i.e., prior to testing, tedious development work is necessary for that testing can be done after all [MSS12, May11]. What however is common to all these approaches is the circumstance that security expert knowledge is not considered for actual testing. Obviously, this is where our method lifts itself from current penetration testing techniques in that it actively integrates such security expert knowledge into the testing process by its VKB. This results in that our tool is also a valuable contribution in penetration testing. Further, extending our method for other types of software systems, i.e., object-oriented or embedded, does not require writing new plugins as in case of, e.g., [MSS12, May11], but instead only the contribution of a security expert to extend the knowledge of
the VKB (i.e., the $\epsilon DB$ and the DCGs).

**Model-based Security Testing** MBST is a very young research field [SGS12], thus, at the time of this writing there does not exist much work in this area. The main branch of existing approaches in MBST works by applying functional security models that are used for test case generation. These test cases then are executed against an underlying SUT. Existing work addresses security testing of Java applications [BBNC01], security policies [MFBLT08], firewalls [WJ02] or the CEPS [J¨08]. These approaches fall under the category of functional security testing. In the event of non-functional security testing, current research however is rather sparse. Existing work either focuses on integrating system and threat models for test generation [WWX07] or a combination of system, vulnerability (resulting from mutating a system model) and attack models [SKR07]. Although these approaches already provide valuable contributions in model-based non-functional security testing, in particular also for web applications as for [SKR07], they still require a security expert or a modeling expert. This is by virtue of necessary models, i.e., threat and attack models, or models of the SUT that need to be developed manually. Our introduced method differs from existing work in that it (i) uses a combination of a system and a risk model, i.e., a formal description of potential exploit scenarios, and (ii) does not require the presence of a security expert for establishing the security model, i.e., the risk model. Thus, our method also comprises a valuable contribution in MBST by thoroughly exploiting its support for automation not only for test execution and evaluation but also for generation of a security model.

**Web Application Security Testing** Contrary to previous topics, relevant related work in web application security testing is broad. Existing work can be classified into active vulnerability testing, i.e., directly exploiting some vulnerability by searching for it, and passive vulnerability testing, i.e., invalidating the behavior or a property of an application and thereby revealing some vulnerability. For the former, i.e., active vulnerability testing, various approaches exist that make use of different techniques, viz. (i) taint analysis [AC10, AC11], (ii) mutation of some input model combined with model checking [BOP12a, BOP12b], (iii) entry point enumeration combined with fuzzing [SXP11, XP10, XSP09] and (iv) dynamic analysis combined with fuzzing [PK08]. In the latter case, i.e., passive vulnerability testing, existing approaches on the one side apply bypass testing, i.e., trying to avert user input checks [OWDH04, TBM+05] and, on the other side, verify or invalidate predefined security properties on the basis of model checking [ACC+10]. In face of web application security testing, our method falls into the category of active vulnerability testing, i.e., our method tries to verify potentially existing vulnerabilities by exploiting them during testing. The most akin work discussed is by Xiong and Peyton [XP10] which, similarly to our work, uses security knowledge provided by databases that are maintained by security experts. However, compared to our work, they do not apply a security risk analysis for identifying potentially
5.2 Positioning w.r.t. State-of-the-Art

vulnerable spots in the SUT but rather use entry point enumeration to then match those results with the database. Further, contrary to our tool, they use predefined fuzz vectors whereas our method dynamically generates test data during testing based on the outcomes of the security risk analysis. In general, one common problem of most existing approaches for web application security testing we see, is that for the most part security vulnerability knowledge is not incorporated. Although the idea of mutating a specification and subsequently generating test traces from it sounds fruitful \cite{BOP12a, BOP12b}, from our point of view, it does not meet the necessary requirements for sound non-functional security testing. This is by virtue, that not always a programming flaw yields a vulnerability but rather the trickiness of an attacker, who manages to circumvent prevention mechanisms. Further, opposed to existing work our tool also addresses the problem of establishing necessary models for testing, viz. an input model of the SUT and a respective test model. As of its high degree of automation, our method and its tool implementation are valuable contributions in web application security testing by delivering an easy to use tool-chain for non-functional security testing of web applications that can be applied by non-security expert testers.

Existing Tools One might argue that our method is implemented in various tools, e.g., Burp Suite \cite{Por14}, WebScarab \cite{OWA14}, Metasploit \cite{May11}, SQL-Ninja \cite{Ice14} or Fortify \cite{HP14}, yet, this is not the case. For example, Burp Suite and WebScarab both are tools for evaluating the security of web applications by performing different exploits. However, contrary to our method, these tools are not automated, i.e., testers need to manually establish specifications of web applications and further, for testing (i.e., discovering vulnerabilities) thorough knowledge of both, web application vulnerabilities and usage of the tool is necessary. Thus, using such tools generally comes along with a shallow learning curve. Metasploit, as already mentioned earlier, requires a not negligible amount of work for that testing can be done (e.g., implementing an exploit as a plugin, configuration of the exploit, and the like). In case of SQLNinja, the problem is that it only targets databases, i.e., it does not consider the application and thus, already requires that a user knows some vulnerable parameter or operation of an application for that the back-end database can be attacked. Thus, a user of SQLNinja requires knowledge that is established by our method. Fortify, in the end, is a static analyzer, i.e., it analyzes the source code of some program/application to search for known programming flaws that lead to potential vulnerabilities. However, like all static analyzers, it (i) does not execute the software, i.e., fails to deliver a working exploit that verifies the vulnerability and (ii), at most due to (i), is prone to deliver false positives.

The above relation of our method to existing related work clearly points out that it is novel in at least two important aspects, viz. (i) the embedding of codified,
5.3 Implications

i.e., formalized, security vulnerability knowledge into a testing process and, (ii) its high degree of automation, viz. the automated security risk analysis and risk assessment, test case and test data generation, and test execution and evaluation. This clearly sets our method apart from existing research in and industrial tools for non-functional security testing of web applications as discussed above. Further, the fundamental idea that is implemented in our method, i.e., the combination of knowledge engineering and logic programming with MBT, can also be applied for non-functional security testing of other types of software systems like embedded or object-oriented systems. By adapting the knowledge that is stored in the VKB, more precisely, the $EDB$ and the test data definitions (observe that the rules of the $IDB$ remain unchanged) non-functional security testing of other types of software systems is easily realizable. Thus, this fundamental idea also sets apart our method from known approaches and techniques in non-functional security testing in general.

W.r.t. design science research that aims for usable, efficient, and effective novel tools, our experimental evaluation has shown that our tool qualifies as a successful implementation of our proposed method. In view of usability, the high degree of automation of our tool implementation clearly speaks for itself. Moreover, our method delivers an out-of-the-box usable tool for non-security expert testers to do effective non-functional security testing. Our tool is efficient in that it (i) eradicates the need for a domain expert to establish a model of a SUT, (ii) eradicates the need for a security expert to design high-quality test cases for non-functional security testing, and (iii) accelerates the overall testing process (again) on the grounds of its high degree of automation. Our tool actually only requires the presence of an “ordinary” tester (whereas ordinary in this case means a tester without special knowledge in, e.g., security of web applications) for doing successful non-functional security testing of web applications. Finally, our tool also is effective in that it, with a low rate of false positives and negatives, detected all existing vulnerabilities during our case studies, thus producing the desired result. Notwithstanding, our tool’s efficiency and effectiveness further undermine its usability.

5.3 Implications

We believe that our work has major implications in non-functional security testing of web applications, but also in security testing and security engineering in general. In the following, we outline these implications.

Penetration testing can be done in a structured, reproducible and efficient manner. Our work has shown that penetration testing of web applications can be advanced to a structured, reproducible and more efficient method. In our case, this is by virtue of the VKB and its security risk analysis, whose outcomes are

1To clarify their different meanings, “efficiency is doing things right, while effectiveness is doing the right things”.

5.3 Implications

deterministic, i.e., given the same security problem $SP$, the resulting risk profile $RP$ will always be the same. This is a crucial results, as it makes testing reproducible. Further, the inherent structure of the risk profile $RP$ makes the overall testing process more structured, and again, reproducible. Finally, testing is more efficient as the risk profile $RP$ contains high-quality descriptions of attack scenarios that focus on potentially existing vulnerabilities in a SUT, i.e., testing is not anymore randomly but instead systematically.

The application of knowledge engineering and automated reasoning is valuable in security testing. The application of knowledge engineering and automated reasoning is valuable in non-functional security testing as it allows to eradicate the need for a necessary security expert for actual testing (remember that one is still needed for establishing and maintaining the knowledge of the VKB). This results in lower costs for testing but at the same time does not decrease the quality of the testing process. Instead, the use of codified knowledge and reasoning rules results in that the quality of testing cannot decrease, as it is not anymore prone to human errors. In addition to that, if merging the knowledge of various security experts in a VKB this results in improved testing quality over time. Further, non-functional security testing becomes feasible for non-security expert testers.

Testing against common vulnerabilities, e.g., SQLI and XSS can be effectively automated. In this dissertation we have designed and implemented a method for non-functional security testing of web applications for automatically detecting SQLI and XSS vulnerabilities. Given the circumstance that these two attacks are among the most common, annoying, persistent and severe, our method becomes even more compelling. Further, by considering the low number of false positives and false negatives (Section 4.2) such an automated detection and verification method proves valuable in eradicating these vulnerabilities in web applications.

Efficient testing tools for making web applications more secure. Our method and in particular its tool implementation yields efficient tools for making web applications more secure. This is by virtue of the implemented tool-chain that (i) automatically establishes a model of the SUT, (ii) generates an efficient test suite using an expert system, (iii) automatically executes this test suite against the SUT and (iv) evaluates the outcomes of the testing process, i.e., detects successful attacks. The application of such a tool-chain during the development stage of the life-cycle of a web application results in more secure web applications that are resistant against common attacks.
Chapter 6

Conclusion

Let us not look back in anger, nor forward in fear, but around us in awareness.

James Thurber

This chapter concludes this dissertation by first summarizing our work (Section 6.1), followed by again listing the key findings and implications of our work (Section 6.2) and an outlook on future work (Section 6.3).

6.1 Summary

The key idea of this dissertation was to show the successful application of logic programming and knowledge engineering for non-functional security testing of web applications. Using foundations of MBT as an underlying testing method, we successfully advanced non-functional security testing to a structured and reproducible testing process. The resulting tool-chain is then usable by non-security expert testers for successfully assessing web applications regarding their security. The resulting security assessment by testing is useful in improving an application’s security prior to deploying it to a production environment.

The tool implementation of our method starts by automatically establishing a model of the SUT, i.e., the security problem $\mathcal{SP}$ by a web spider. We consider the system model a “security problem” as it describes a vulnerable web application, thus it comprises a security problem. This security problem $\mathcal{SP}$ then is investigated by a security risk analysis that yields the risk profile $\mathcal{RP}$, a model describing potential attack scenarios against the SUT. The security risk analysis is implemented as part of our VKB by logical reasoning rules that use both, codified security vulnerability knowledge of the $\mathcal{EDB}$ and knowledge on the SUT that is provided by the security problem $\mathcal{SP}$. The risk profile $\mathcal{RP}$ then is used as an executable specification for testing, i.e., our test engine processes this model by executing the contained attack
scenarios (or test cases) against the SUT. Predefined oracles in the form of potential attack goals are used for evaluating the outcome of executed test cases to infer whether a test case passed, failed or if neither pass nor fail can be stated, i.e., the outcome is inconclusive. Our tool returns with a test log and test feedback in the risk profile $R_P$.

The formal foundations of our testing method as given in Section 3.3 and, further, the experimental evaluation as done in Chapter 4 and discussed in Chapter 5, both show that our testing method describes a valid testing system w.r.t. Gourlay [Gou83], i.e., if some program violates some specification the tool implementation of our method detects this violation.

6.2 Key Findings and Implications

As outlined in Chapter 5, we here recap the main findings on the grounds of our case studies and the implications of our work.

Key Findings

- Research challenge RC1 that addressed the codification of security vulnerability knowledge for non-functional security testing was successfully fulfilled by contributions C1, i.e., our VKB, C2, i.e., our security risk analysis, C4, i.e., sets of logical predicates for declaring a security problem $S_P$ and a risk profile $R_P$ and C7, i.e., our tool implementation for model-based non-functional security testing of web applications.

- Research challenge RC2 that addressed an automated security risk analysis and risk assessment was successfully fulfilled by contributions C2, i.e., our security risk analysis, C4, i.e., sets of logical predicates for declaring a security problem $S_P$ and a risk profile $R_P$ and C7, i.e., our tool implementation for model-based non-functional security testing of web applications.

- Research challenge RC3 that addressed automated, non-functional security testing of web applications to detect SQLI and XSS vulnerabilities was successfully fulfilled by contributions C3, i.e., automatic generation of a test suite, C4, i.e., sets of logical predicates for declaring a security problem $S_P$ and a risk profile $R_P$, C5, i.e., a posteriori evaluation of test runs by test oracles, C6, i.e., successfully detection of SQLI and XSS attacks and C7, i.e., our tool implementation for model-based non-functional security testing of web applications.

- Research challenge RC4 that addressed automated generation, execution and evaluation of negative security test cases was successfully fulfilled by contributions C1, i.e., our VKB, C2, i.e., our security risk analysis, C3, i.e., automatic
generation of a test suite, $C_5$, i.e., a posteriori evaluation of test runs by test oracles and $C_7$, i.e., our tool implementation for model-based non-functional security testing of web applications.

- Automated black-box testing against stored XSS vulnerabilities is difficult in terms of detecting a successful attack.

Implications

- Penetration testing can be done in a structured, reproducible and effective manner.
- The application of knowledge engineering and automated reasoning is valuable in security testing to make non-functional security testing feasible for non-security expert testers.
- Testing against common vulnerabilities, i.e., SQLI and XSS, can be efficiently automated.
- Efficient testing tools for making web applications more secure are feasible.

6.3 Future Work

In the course of our case studies we have identified relevant areas of future work as discussed in the following.

Taking care of System and Implementation Specifics  Currently, our $\mathcal{EDB}$ and $\mathcal{IDB}$ do not take into consideration system and implementation specifics during the security risk analysis. More precisely, our method currently is not able to reason on complex, user-defined types. Further, we do not take into consideration the respective programming language that was used to implement the application but rather declare an $\mathcal{EDB}$ for a specific system type. Overcoming this issue by refining the $\mathcal{EDB}$ would result in a more generic $\mathcal{EDB}$ that can be used for different system types. This for sure also requires rewriting the rules of the $\mathcal{IDB}$.

Autonomous Learning of the $\mathcal{EDB}$  Currently, our method requires a security expert to declare an $\mathcal{EDB}$. In light of existing machine learning techniques, an idea would be to develop learning strategies which let our method learn its respective $\mathcal{EDB}$ in an autonomous way. At this, resources like the CVE (Common Vulnerabilities and Exposures) database [MIT14] or NVD (National Vulnerability Database) [NIS14] would provide the inputs from which our method then would learn attacks and vulnerabilities to then automatically generate an $\mathcal{EDB}$. 
Autonomous Maintenance of the $\mathcal{EDB}$ Currently, not only for the establishment but also for maintenance of the knowledge of the $\mathcal{EDB}$ a security expert is required. In light of software versioning however, maintenance of the VKB could be improved by providing merging facilities \cite{Men02} that automatically integrate new knowledge in an existing $\mathcal{EDB}$. Further, by again learning such knowledge autonomously (see previous issue), the need for a security expert could be completely eradicated.

Improving the $\mathcal{EDB}$ by Learning from Test Feedback Our model-based tool implementation automatically generates test feedback into the risk profile $\mathcal{RP}$ which then is used by developers to improve the SUT. However, this knowledge not only can aid in improving the SUT, it also can be of great relevance for improving our tool and its method by learning from this very test feedback. Using techniques like, e.g., reinforcement learning \cite{Bar98}, verdicts of test runs, i.e., \textit{PASS} (the reward) and \textit{FAIL} (the punishment), can be used for improving the knowledge of the $\mathcal{EDB}$ and altering the decisions the $\mathcal{IDB}$ takes during risk analysis.

Test Evaluation Our case studies have shown that our tool in case of evaluating stored XSS attacks soon hits its wall. However, this is by nature of the trickiness of stored XSS attacks. Thus, we also see a need for improvement of the test evaluation for stored XSS attacks. Currently, our method and its tool implementation rely on a guessing strategy in deducing the verdict for a test run. Although our case studies have shown that such strategies work, under certain circumstances, they do not work proper and yield false positives or negatives (e.g., executing a stored XSS attack against some operation vulnerable to SQLI). However, to improve this, our tool must move away from its black-box view and rather use, at least, a gray-box view of the SUT to analyze, if necessary, its source code to identify potentially vulnerable sinks in the application for stored XSS attacks.

Automated Generation of DCGs by Mutation Besides the $\mathcal{EDB}$, also necessary DCGs for our method and its tool implementation currently need to be defined manually by a security expert. We consider to also automate this process by (i) algorithmically translate existing BNFs of, e.g., SQL or Javascript, into a corresponding Prolog-based DCG notation, and (ii) define relevant mutation operators which introduce necessary rules into the DCG that allow to then generate malicious input strings.

Extending Our Method’s “Skills” As of the knowledge codified in the $\mathcal{EDB}$, our method currently is capable of detecting SQLI and XSS vulnerabilities in web applications. Although this is already a good starting point, for becoming a more mature non-functional security testing method, our method’s “skills” however require improvement by extending the set of potential vulnerabilities it can detect. Immediate candidates for such an extension are, e.g., format string or remote code
execution vulnerabilities. Such can be achieved pragamatically by following the de-
liberations according to Section 3.1.2.4, i.e., to codify a security expert’s knowledge
on a vulnerability using definitions 3.2 and 3.3.
Bibliography


BIBLIOGRAPHY


[OMG12b] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1*, 2012.


RandomStorm Ltd. Damn Vulnerable Web Application (DVWA), 2014.\texttt{http://www.dvwa.co.uk}.


Appendix A

Extensional Database

Following we show the full EDB (Figure A.1) for testing web applications.

1 \% exploit(exploitID) :-
2 \% declares a known exploit
3 exploit(sql_attack).
4 exploit(xss).
5
6 \% attack(exploitID, attackVariation, attackPattern, list_of_goals) :-
7 \% declares a specific attack under some known exploit
8 attack(sql_attack,signature_evasion,sqlap,[authentication,leakage,tampering]).
9 attack(sql_attack,blind,sqlap,[authentication,leakage,tampering,dos]).
10 attack(sql_attack,type_handling,sqlap,[authentication,leakage,tampering]).
11 attack(xss,reflected,xssap,[leakage,authentication]).
12 attack(xss,stored,xssap,[leakage,authentication]).
13
14 \% vul_type(exploitID, type) :-
15 \% declares some known type to be vulnerable against a known exploit
16 vul_type(sql_attack,text) :- !.
17 vul_type(sql_attack,password) :- !.
18 vul_type(sql_attack,number) :- !.
19 vul_type(sql_attack,hidden) :- !.
20 vul_type(xss,hidden) :- !.
21 vul_type(xss,text) :- !.
22
23 \% attack_pattern(attackPattern, attack, module, operation, intrusionPoint) :-
24 \% matches a specific attack pattern against an operation to check
25 \% whether the given operation is potentially vulnerable to a given, known exploit
26 attack_pattern(sqlap,sql_attack,M,O,IP):-module(M),
27 operation(M,O,___),
28 parameter(M,O,IP,T),
29 vul_type(sql_attack,T).
attack_pattern(xssap,xss,M,O,IP):-module(M),
    operation(M,0,_,_),
    parameter(M,O,IP,T),
    vul_type(xss,T).

Figure A.1: $\mathcal{EDB}$ for detecting SQLI and XSS vulnerabilities in web applications.
Appendix B

Intensional Database

Following we show the $TDB$ (Figure B.1) which implements the rules for our security risk analysis as introduced in Chapter 3.

```
% blacklist( + module, + operation, - exploit, - intrusionPoint, - attackPattern) :-
%     blacklists an operation if one of the available attack patterns can
%     be matched against it (see attack_pattern/5 in EDB)

blacklist(M,O,E,IP,AP) :- module(M),
                     operation(M,O,L),
                     attack_pattern(AP,E,M,O,IP).

% threat( + module, + operation, + attackPattern, + intrusionPoint, + exploit,
%         - exploitManifestation, - goals) :-
%     searches and returns all potential threats for a blacklisted operation
%     according to the identified exploit (see blacklist/5)

threat(M,O,AP,IP,E,EM,G) :- blacklist(M,O,E,IP,AP),
                         attack(E,EM,AP,G).

% risk( + module, + operation, + attackPattern, + intrusionPoint, + exploit, + exploitManifestation,
%       + goals, - riskAssessment) :-
%     collects all threat related information and encapsulates it as a risk;
%     additionally, risk assessment information is added

risk(M,O,AP,E,EM,E,M,G,L) :- threat(M,O,AP,E,EM,G),
                         operation(M,O,L),
                         num_params(L,N),
                         comp(M,O,L,N,C),
                         attack_severity(E,EM,G,S),
                         impact(C,I1),
                         dcomp(I1,S,1),
                         probability(C,I,P),
                         risk_level(I,P,RL).
```
% comp(+module,+operation,+parameters,-numOfParams, -complexity) :-
% calculates the complexity of an operation on its number of parameters;
% this complexity is used as a basis for later risk assessment

comp(M,O,L,N,C) :- N > 0,
  operation(M,O,L,_),
  element_at(L,N,P),
  parameter(M,O,P,T),
  compt(T,C1),
  N1 is N - 1,
  comp(M,O,L,N1,C2),!
  dcomp(C2,C1,C).

% num_params(+list,-numOfParams) :-
% calculates the number of parameters in an operation's parameter list
num_params([_|T],N) :- num_params(T,N1),
  N is N1 + 1.
num_params([],0).

element_at(+list,+index,-element) :-
% returns an element from a list at a given index
element_at([_|T],K,X) :- K > 1,
  K1 is K - 1,
  element_at(T,K1,X).
element_at([H|_],1,H).

compt(+type,-complexity) :-
% returns the complexity for a given type
compt(T,low) :- type(T,primitive).
compt(T,high) :- type(T,complex).

% attack_severity(+exploit,+attackManifestation,+goals,-severity) :-
% calculates the severity of a known attack by its potentially achievable goals
attack_severity(E,EM,[_|T],S) :- severity(H,S1),
  attack_severity(E,EM,T,S2),
  dcomp(S1,S2,S).

% dcomp(+c1,+c2,-c) :-
% calculates a new, adapted complexity c based on c1 and c2
dcomp(very_low,very_low,very_low) :- !.
dcomp(very_low,low,low) :- !.
dcomp(very_low,medium,low) :- !.
dcomp(very_low,medium,medium) :- !.
dcomp(very_low,high,low) :- !.
dcomp(very_low,very_high,medium) :- !.
dcomp(low,very_low,low) :- !.
dcomp(low,low,low) :- !.
dcomp(low,medium,medium) :- !.
dcomp(low,very_high,medium) :- !.
dcomp(medium,very_low,low) :- !.
dcomp(medium,medium,medium) :- !.
dcomp(medium,high,high) :- !.
dcomp(medium,very_high,high) :- !.
dcomp(high,very_low,low) :- !.
dcomp(high,low,medium) :- !.
dcomp(high,medium,high) :- !.
dcomp(high,high,high) :- !.
dcomp(high,very_high,high) :- !.
dcomp(very_high,very_low,medium) :- !.
dcomp(very_high,low,high) :- !.
dcomp(very_high,medium,high) :- !.
dcomp(very_high,high,medium) :- !.
dcomp(very_high,very_high,very_high).

% impact(+complexity,-impact) :-
% calculates a threats potential impact based on the operation complexity
% using a direct relation

impact(very_low,very_low) :- !.
impact(low,low) :- !.
impact(medium,medium) :- !.
impact(high,high) :- !.
impact(very_high,very_high).

% probability(+complexity,+impact,-probability) :-
% calculates the probability for a threat to become real based on the
% operation complexity and the threat's potential impact

probability(very_low,very_low,very_low) :- !.
probability(low,very_low,very_low) :- !.
probability(medium,very_low,very_low) :- !.
probability(high,very_low,very_low) :- !.
probability(very_high,very_low,very_low) :- !.
probability(very_low,low,very_low) :- !.
probability(low,low,very_low) :- !.
probability(medium,low,very_low) :- !.
probability(high,low,very_low) :- !.
probability(very_high,low,very_low) :- !.
probability(very_low,medium,high) :- !.
probability(low,medium,medium) :- !.
probability(medium,medium,low) :- !.
probability(very_high,medium,low) :- !.
probability(very_low,high,very_high) :- !.
probability(low,high,very_high) :- !.
probability(medium,high,high) :- !.
probability(high,high,high) :- !.
probability(very_high,high,medium) :- !.
probability(very_low,very_high,very_high) :- !.
probability(low,very_high,very_high) :- !.
probability(medium,very_high,very_high) :- !.
probability(high,very_high,high) :- !.
probability(very_high,very_high,medium).

risk_level(I,P,RL) :- value(I,IV),
value(P,PV),
RL is PV * IV.

% risk_level(+impact,+probability,-risk_level) :-
% calculates the risk level based on a threat's potential impact and its chances (probability) to become real
% risk_level = impact * probability

severity(given_goal,severity) :-
% declares the severity of a known goal in terms of LOW, MEDIUM, HIGH

language(+languageID).
% declares some known implementation language

value(+valueID,+numeric) :-
% declares values used by the risk analysis and its according numeric representation

% value(very_low,1).
% value(low,2).
% value(medium,3).
% value(high,4).
% value(very_high,5).
% `complexity(complexity) :-`
% declares a complexity for a known type
complexity(primitive).
complexity(complex).

Figure B.1: Implementation of our security risk analysis by the **IDB** as part of our VKB.
Appendix C

Risk Profile

Following we show the full risk profile $\mathcal{RP}$ of DVWA (Figure C.1) as used during the evaluation of our method and its tool implementation.

```scala
object DVWA_RP {
  val risk1 = new Risk(1,
    new Attack(exploit = "sql_attack", manifestation = "signature_evasion"),
    new ThreatProfile(goals = List("authentication","leakage","tampering"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/sqli/#",
      intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  val risk2 = new Risk(2,
    new Attack(exploit = "sql_attack", manifestation = "blind"),
    new ThreatProfile(goals = List("authentication","leakage","tampering","dos"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/sqli/#",
      intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  val risk3 = new Risk(3,
    new Attack(exploit = "sql_attack", manifestation = "type_handling"),
    new ThreatProfile(goals = List("authentication","leakage","tampering"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/sqli/#",
      intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  val risk4 = new Risk(4,
    new Attack(exploit = "xss", manifestation = "reflected"),
    new ThreatProfile(goals = List("leakage","authentication"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/sqli/#",
      intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))

  val risk5 = new Risk(5,
    new Attack(exploit = "xss", manifestation = "stored"),
    new ThreatProfile(goals = List("leakage","authentication"),
      impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/sqli/#",
      intrusionPoint= "id"),
    new Test(executions = 10, outcomes = Nil))
}
```
intrusionPoint= "id"),
new Test(executions = 10, outcomes = Nil))
val risk6 = new Risk(6,
new Attack(exploit = "sql_attack", manifestation = "signature_evasion"),
new ThreatProfile(goals = List("authentication","leakage","tampering"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/sqli_blind/#",
intrusionPoint= "id"),
new Test(executions = 10, outcomes = Nil))
val risk7 = new Risk(7,
new Attack(exploit = "sql_attack", manifestation = "blind"),
new ThreatProfile(goals = List("authentication","leakage","tampering","dos"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/sqli_blind/#",
intrusionPoint= "id"),
new Test(executions = 10, outcomes = Nil))
val risk8 = new Risk(8,
new Attack(exploit = "sql_attack", manifestation = "type_handling"),
new ThreatProfile(goals = List("authentication","leakage","tampering"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/sqli_blind/#",
intrusionPoint= "id"),
new Test(executions = 10, outcomes = Nil))
val risk9 = new Risk(9,
new Attack(exploit = "xss", manifestation = "reflected"),
new ThreatProfile(goals = List("leakage","authentication"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/sqli_blind/#",
intrusionPoint= "id"),
new Test(executions = 10, outcomes = Nil))
val risk10 = new Risk(10,
new Attack(exploit = "xss", manifestation = "stored"),
new ThreatProfile(goals = List("leakage","authentication"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/sqli_blind/#",
intrusionPoint= "id"),
new Test(executions = 10, outcomes = Nil))
val risk11 = new Risk(11,
new Attack(exploit = "sql_attack", manifestation = "signature_evasion"),
new ThreatProfile(goals = List("authentication","leakage","tampering"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/xss_r/#",
intrusionPoint= "name"),
new Test(executions = 10, outcomes = Nil))
val risk12 = new Risk(12,
new Attack(exploit = "sql_attack", manifestation = "blind"),
new ThreatProfile(goals = List("authentication","leakage","tampering","dos"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/xss_r/#",
intrusionPoint= "name"),
new Test(executions = 10, outcomes = Nil))
val risk13 = new Risk(13,
new Attack(exploit = "sql_attack", manifestation = "type_handling"),
new ThreatProfile(goals = List("authentication","leakage","tampering"),
impact = "high", probability = "high", riskLevel = 16),
new SUT(module = "dvwa", operation = "/vulnerabilities/xss_r/#",
    intrusionPoint= "name"),
new Test(executions = 10, outcomes = Nil))
val risk14 = new Risk(14,
    new Attack(exploit = "xss", manifestation = "reflected"),
    new ThreatProfile(goals = List("leakage","authentication"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_r/#",
    intrusionPoint= "name"),
    new Test(executions = 10, outcomes = Nil))
val risk15 = new Risk(15,
    new Attack(exploit = "xss", manifestation = "stored"),
    new ThreatProfile(goals = List("leakage","authentication"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_r/#",
    intrusionPoint= "name"),
    new Test(executions = 10, outcomes = Nil))
val risk16 = new Risk(16,
    new Attack(exploit = "sql_attack", manifestation = "signature_evasion"),
    new ThreatProfile(goals = List("authentication","leakage","tampering"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
    new Test(executions = 10, outcomes = Nil))
val risk17 = new Risk(17,
    new Attack(exploit = "sql_attack", manifestation = "blind"),
    new ThreatProfile(goals = List("authentication","leakage","tampering","dos"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
    new Test(executions = 10, outcomes = Nil))
val risk18 = new Risk(18,
    new Attack(exploit = "sql_attack", manifestation = "type_handling"),
    new ThreatProfile(goals = List("authentication","leakage","tampering"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
    new Test(executions = 10, outcomes = Nil))
val risk19 = new Risk(19,
    new Attack(exploit = "sql_attack", manifestation = "signature_evasion"),
    new ThreatProfile(goals = List("authentication","leakage","tampering"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "mtxMessage"),
    new Test(executions = 10, outcomes = Nil))
val risk20 = new Risk(20,
    new Attack(exploit = "sql_attack", manifestation = "blind"),
    new ThreatProfile(goals = List("authentication","leakage","tampering","dos"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "mtxMessage"),
    new Test(executions = 10, outcomes = Nil))
val risk21 = new Risk(21,
    new Attack(exploit = "sql_attack", manifestation = "type_handling"),
    new ThreatProfile(goals = List("authentication","leakage","tampering"),
    impact = "high", probability = "high", riskLevel = 16),
    new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "mtxMessage"),
    new Test(executions = 10, outcomes = Nil))
val risk22 = new Risk(22,
  new Attack(exploit = "xss", manifestation = "reflected"),
  new ThreatProfile(goals = List("leakage"), "authentication"),
  impact = "high", probability = "high", riskLevel = 16),
  new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
  new Test(executions = 10, outcomes = Nil))
val risk23 = new Risk(23,
  new Attack(exploit = "xss", manifestation = "stored"),
  new ThreatProfile(goals = List("leakage"), "authentication"),
  impact = "high", probability = "high", riskLevel = 16),
  new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
  new Test(executions = 10, outcomes = Nil))
val risk24 = new Risk(24,
  new Attack(exploit = "xss", manifestation = "reflected"),
  new ThreatProfile(goals = List("leakage"), "authentication"),
  impact = "high", probability = "high", riskLevel = 16),
  new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
  new Test(executions = 10, outcomes = Nil))
val risk25 = new Risk(25,
  new Attack(exploit = "xss", manifestation = "stored"),
  new ThreatProfile(goals = List("leakage"), "authentication"),
  impact = "high", probability = "high", riskLevel = 16),
  new SUT(module = "dvwa", operation = "/vulnerabilities/xss_s/#",
    intrusionPoint= "txtName"),
  new Test(executions = 10, outcomes = Nil))
}

Figure C.1: Risk profile $\mathcal{RP}$ for DVWA [Ran14] as used during evaluation of our method and its tool implementation.
Appendix D

Definite-clause Grammars

Following we show the full DCG (D.1) for testing web applications.

1 % testData(*exploit,*manifestation,*type,*dbvendor) -->
2  % clause to query for test data under some known exploit and its manifestation
3 testData(sqlAttack,sigEvasion,_,DB) -->
4   sql_apostrophe ;
5   ((sql_apostrophe,((or ; b_or),sql_space,(one,equals,one ; two,equals,two) ;
6      (and ; b_and),sql_space,one,equals,two ;
7      (or ; b_or),sql_space,ab(DB),equals,ab(DB)));
8   (admin ;
9      number,((or ; b_or),sql_space,one,equals,one ;
10     (and ; b_and),sql_space,one,equals,two ;
11     sql_apostrophe,((or ; b_or),sql_space,one,equals,one ;
12     (and ; b_and),sql_space,one,equals,two)));
13   ((isMySQL(DB)),mysql_comment ;
14    (isMSSQL(DB)),mssql_comment ;
15    (isOracle(DB)),oracle_comment ;
16    (isPostgres(DB)).postgres_comment).
17 testData(sqlAttack,blind,_,DB) -->
18   number,sql_apostrophe,or,((isMySQL(DB)),mysql_delay ;
19    (isMSSQL(DB)),mssql_delay ;
20    (isOracle(DB)),oracle_delay ;
21    (isPostgres(DB)).postgres_delay).
22 testData(sqlAttack,typeHandling,number,DB) -->
23   foo,or,sql_space,a,equals,a,((isMySQL(DB)),mysql_comment ;
24    (isMSSQL(DB)),mssql_comment ;
25    (isOracle(DB)),oracle_comment ;
26    (isPostgres(DB)).postgres_comment).
27 testData(sqlAttack,typeHandling,text,DB) -->
28   number,sql_apostrophe,or,sql_space,one,equals,one,((isMySQL(DB)),mysql_comment ;
29    (isMSSQL(DB)),mssql_comment ;
30    (isOracle(DB)),oracle_comment ;
31    (isPostgres(DB)).postgres_comment).
32 testData(xss,reflected,_,_) --> xss_inject.
33 testData(xss,stored,_,_) --> xss_inject.
34 xss_inject -->
35    xss_r_angle_bracket,xss_quote,xss_r_angle_bracket,start_script,xss_alert,end_script
36    start_img,xss_js_alert,end_img.
Figure D.1: DCG for detecting SQLI and XSS vulnerabilities in web applications.