Int J Softw Tools Technol Transfer (2019) 21:221–246
https://doi.org/10.1007/s10009-017-0472-3

REGULAR PAPER

Knowledge-based security testing of web applications by logic programming

Philipp Zech[1] · **Michael Felderer**[1] · **Ruth Breu**[1]

Published online: 13 September 2017

Abstract This article introduces a new method for knowledge-based security testing by logic programming and the related tool implementation for model-based non-functional security testing of web applications. Our method helps to overcome the current prevalent focus on functional instead of non-functional (or negative) requirements as well as the required high level of security knowledge when performing non-functional security testing. It addresses issues like considering non-functional requirements for testing, managing the virtually infinite amount of negative security test cases, advancing non-functional security testing away from its prevalent penetration testing-like style, and making non-functional security testing feasible for testers that are not experts in security via a security knowledge base. The method and its model-based tool implementation are evaluated in two studies, which show the method's effectiveness in detecting vulnerabilities in web applications and thus, also its value in making software system more secure.

Keywords Security testing · Model-based testing · Risk-based testing · Test data generation · Logic programming · Web applications · Knowledge management

1 Introduction

In recent years, web applications have become a central part of our lives, both commercially and privately. These applications 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 [1]. Security hence plays an important role for web applications.

Among the possible attacks against web applications, two classes of attacks stand out as of their potential damage and prevalence, viz. SQL injection (SQLI) and Cross-site scripting (XSS) attacks [1,2]. In SQLI, an attacker avails himself the circumstance that user input is not properly sanitized by the application; thus, it can carry malicious database statements which, when executed, exploit the back-end database of the application. In XSS, an attacker also avails himself 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.

Recently, model-based testing (MBT) [3], the variant of testing that relies on explicit models encoding information on the system under test (SUT) and/or its environment, has been extended for the purpose of security testing [4]. Resulting model-based security testing (MBST) approaches [5] thus immediately benefit from MBT 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 [6]. A crucial role in MBST is occupied by various kinds of security models, i.e., threat, fault and risk models, and weakness and vulnerability models. Whereas the former, i.e., threat, fault and risk models, specify what can go wrong, the latter, i.e., weakness and vulnerability models, aim at describing a weakness or a vulnerability itself. A common drawback, however, of these *security models* is the necessity of a security expert with the necessary domain knowledge to establish them for that efficient test cases been derived for a SUT. This necessity fortunately can be eradicated by combining MBST

✉ Michael Felderer
Michael.Felderer@uibk.ac.at

[1] Institute for Computer Science, University of Innsbruck, Innsbruck, Austria

with knowledge engineering and logic programming, i.e., by codifying domain-specific security vulnerability knowledge by *facts* and further, using reasoning *rules* to automatically infer a security model of the SUT. Further, by incorporating codified security vulnerability knowledge, inevitable negative requirements, i.e., requirements that state what a system shall not do compared to positive requirements that denote intended functionality, can be considered in testing at no additional cost [7].

In this article, we introduce *knowledge-based security testing of web applications founded on logic programming and model-based testing*, a novel method for detecting existing SQLI and XSS vulnerabilities in web applications. Contrary to previous approaches, our method works on a model of the application (its specification), creates actual inputs that exploit identified and existing vulnerabilities, evaluates the application before it is deployed, incurs no additional overhead, and establishes negative requirements by logic programming that are vital for testing. This stems from the relation that a negative requirement specifies everything outside the system, i.e., what it is *not supposed* to do thus providing the base for non-functional security testing. Observe that non-functional and negative requirements thus relate to the same kind of information in our context, yet at different levels of abstraction. Whereas negative requirements denote a specification, a resulting non-functional test case tests against the specified behavior from such a negative requirement. Such an automated establishment of negative requirements drastically lowers the level of expertise usually required for non-functional security testing. Further, using techniques from model-based testing (MBT), e.g., test generation and selection algorithms, yields that our method provides a structural approach for non-functional security testing. We have implemented our method in a tool that, starting from a software model of the application, automatically generates SQLI and XSS attacks for web applications written in PHP/SQL. Our method essentially describes a model-based, black-box testing view on the system under test as it does not rely on an application's source code but instead on its specification by a formal model. It is thus designed to detect vulnerabilities in a web application prior to its deployment. This is by virtue of employing formal software models for testing, which allow detection of design flaws already during early design of a software system. Consequently, existing vulnerabilities can be mitigated before the application reaches the end users, resulting in more secure web applications.

Overview of Proposed Method Our method as sketched in Fig. 1 and its tool implementation take four steps in security testing a web application:

(1) Establishing a declarative system model of the system under test (SUT) at the interface level. In our method, such a model is called a security problem $\mathcal{SP}$. This is by virtue of that we consider the SUT a security problem as it is vulnerable to attacks. We discuss the security problem $\mathcal{SP}$ in Sect. 3.1.

(2) Using foundations of logic programming, i.e., knowledge representation and reasoning, our method then establishes a risk profile $\mathcal{RP}$ by executing a security risk analysis on the security problem $\mathcal{SP}$ from step 1. This risk profile $\mathcal{RP}$ then describes security risks, i.e., vulnerabilities, the SUT potentially is exposed to. We discuss the security risk analysis and its resulting risk profile $\mathcal{RP}$ in Sects. 3.2 and 3.3.

(3) The risk profile $\mathcal{RP}$, as of also representing an executable specification, then is executed against the SUT. The subtasks in this step are

  (3.1) for each risk of the risk profile $\mathcal{RP}$ an in-memory test case tc is generated to be executed against the SUT,
  (3.2) for each generated test case test data is generated,
  (3.3) each test case then is executed against the SUT.

  We discuss test data generation, test execution and test evaluation in Sects. 3.2.3, 3.4, and 3.5.

(4) Finally, the outcome of each test case is evaluated by an oracle.

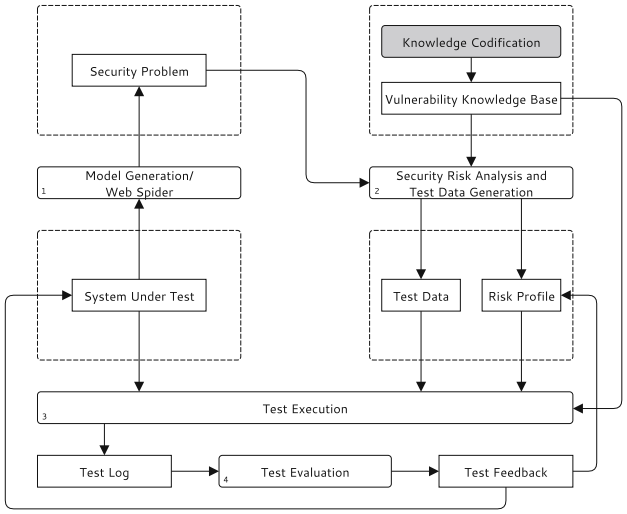


Fig. 1 Abstract overview of our proposed method: a web spider automatically establishes a declarative model of the SUT (a web application), i.e., the security problem $\mathcal{SP}$, which is then subjected to a security risk analysis, i.e., test generation. The resulting risk profile $\mathcal{RP}$ then is used as an executable specification for non-functional security testing of the SUT. Our tool returns with a test log. Observe that *Knowledge Codification* is the only manual task in our framework that actually requires security expert knowledge

Results We have evaluated our method and its model-based tool implementation in two experiments (Sect. 5) where we used it to detect SQLI and XSS vulnerabilities in Damn Vulnerable Web Application (DVWA) [8], a

PHP/MySQL web application that is highly vulnerable providing a legal environment for security professionals to test their skills and tools. Our experimental evaluation shows that our method is capable of detecting SQLI and XSS vulnerabilities under real-world conditions. Further, the generated exploits are, to a certain degree, apt to circumvent common prevention mechanisms. We thus argue that our work is a valuable contributions in non-functional security testing of web applications.

1.1 Contributions

The results presented in this article contribute to secure software engineering and more specifically to security testing in the following respects:

(1) A method and model-based tool implementation for non-functional security testing of web applications by logic programming. Our method generates effective test cases by an executable specification as to the risk profile $\mathcal{RP}$ and according test data. Our method returns with a verdict, which is either *pass*, *fail*, or *inconclusive*, depending on whether a test succeeds, fails, or none of pass or fail can be stated [9–11].
(2) An expert system for security testing in Prolog, which comprises both a security risk analysis and a grammar-based test data generator. Our expert system contributes (i) predicates for codifying security vulnerability knowledge, (ii) rules to infer new knowledge, i.e., a risk profile $\mathcal{RP}$, from a security problem $\mathcal{SP}$ by on the grounds of the codified security vulnerability knowledge, as well as (iii) rules and grammars for generating malicious test data [12,13].
(3) Two domain-specific languages (DSLs) for abstracting both declarative models of web applications, i.e., security problems $\mathcal{SP}$, and their corresponding risk profiles $\mathcal{RP}$. Their purpose is to provide a human-readable layer of abstraction onto the declarative models our method works on.
(4) An evaluation where we used our method and its model-based tool implementation to detect SQLI and XSS vulnerabilities in a vulnerable web application written in PHP/SQL. Our tool verified all existing vulnerabilities. Thus, in our experiments, our tool has shown to be valuable for non-functional security testing of web applications. Our current evaluation extends our previous work [11] in that we use a real-world application for evaluation instead of a simple toy example.

A key feature of our method thus is that it reduces the need for a security expert for testing except for codifying security vulnerability knowledge for our expert system. However, this is a one-time task, as the codified knowledge is reusable. Our method hence ultimately renders non-functional security testing of web applications feasible for laypersons in information security as of its *internalization* of security vulnerability knowledge. Further, as of its high degree of automation our tool implementation does not require any specific domain knowledge, either for system or test modeling as otherwise required by most existing work (Sect. 2).

The work presented in this article extends our previous work (as cited above) in various aspects, viz. (i) it introduces a model-based tool implementation of our method, (ii) it extends previous work by automated test data generation, and (iii) we provide a carefully designed empirical evaluation of our method and its tool implementation by two case studies.

1.2 Article organization

The remainder of our article is structured as follows. Section 2 discusses relevant related work. In Sect. 3 we then introducing our method in. Next, Sect. 4 discusses our experiments followed by a discussion of our results 5. We conclude in Sect. 6 with a summary and outlook on future work.

2 Related work

Following we discuss related work to the method presented in this article.

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 techniques [14] together with either symbolic execution [15] or feasible path analysis [16] have been harnessed.

In case of test case generation, existing approaches, which use logic programming, mainly build on constraint solving techniques. Vemuri et al. [17] 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 [18] suggests a method for generating test cases from Prolog-based specifications. A custom meta-interpreter monitors and controls the execution of programs using specified paths. This then allows the generation of tests for specification-based testing. Bieker and Marwedel [19] 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. [20] 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 some given coverage criterion (e.g., path or statement coverage). Lötzebeyer and Pretschner [21] 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. some predefined constraint to produce meaningful test sequences for specification-based testing. The work of Caballero et al. [22], 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 et al. [23] describes a complete testing methodology 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).

The work presented in this article describes a novel contribution in logic programming for testing as applies logic programming for the purpose of security testing of software systems. Further, we apply mechanics of logic programming for automated test data generation for security testing which so far has not been done.

2.2 Security testing

In the face of existing security testing techniques, our approach falls into the category of negative testing techniques, i.e., to focus on *negative* requirements specifying hidden security vulnerabilities [24]. Whereas for its counterpart, functional (or positive) security testing well-established techniques exist [25], for negative security testing, those are scarce. In practice, negative security testing simulates attacks as performed by hackers, which is called penetration testing. The main goal in this kind of testing is to compromise the security of a system [26,27]. Another approach to negative security testing is fuzzing [28,29], which initially was designed for testing protocol implementations on possible security flaws due to improper input handling. One of the main disadvantages of those techniques is their lack for supporting a systematic procedure when testing regarding the order of execution of test cases. Thus, negative testing is a tedious and time-consuming task. A solution to this lack of a systematic procedure is to incorporate risks for testing [30–32]. Risk-based testing consider risks to solve design, selection and prioritization of test cases [30]. Stallbaum et al. [33] propose RiteDAP, a tool to automatically generate system tests from activity diagrams which considers risks for test case prioritization. They also investigated different prioritization strategies w.r.t. the resulting fault detection rate.

Contrary to existing research, the method introduced in this article presents a systematic procedure for non-functional security testing. This stems for the deterministic nature of both logic programming and model-based testing which make testing in the context of our method fully reproducible. Further, the use of model-based testing allows incorporating a formal definition of risks as testing artifacts to solve design, selection and prioritization of test cases.

2.3 Model-based security testing

Blackburn et al. [34] describe 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. Mou-elhi et al. [35] 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. Jür-jens and Wimmel investigated the automated generation of test sequences from models in Focus.[1] They have applied their approach to both model-based testing of firewalls [36] and reactive systems [37]. Jürjens further extended UML by UMLsec, an extension to UML for secure systems development [38], which he applied in model-based testing of the Common Electronic Purse Specification (CEPS) [39]. 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. [40] 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. Maarback et al. [41] present a method where threat trees are built manually on the grounds of manually constructed data flow diagrams for an application. Subsequently, these threat trees are used to generate test cases by searching for test sequences

[1] The Focus language is a mathematical framework for the specification, refinement, and verification of distributed, reactive systems.

in threat tress, and corresponding test data. The latter further requires specifying relevant input parameters. In their 2012 paper, Xu et al. [42] extend their work [43] to further generate executable test code on the grounds of model implementation mappings that relate elements of threat models to their implemented counterparts.

In light of current research in model-based security testing, our main contribution is its novel application to non-functional security testing of web applications to detect SQLI and XSS vulnerabilities.

2.4 Web application security testing

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 penetration testing-like assessment methods whatsoever.

Avancini and Ceccato [44] 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. [45,46] motivates the use of a secure model formulated in ASLan++.[2] This model then is mutated to introduce typical vulnerabilities in web applications, and subsequently passed to a model-checker which yields attack traces. These 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. [47] 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. [48] 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. [49] 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 [50] propose a model-based framework for penetration testing of web applications. Their framework is linked to reference databases that are maintained by security experts. Based on the contents of such databases, their framework tests for known vulnerabilities by generating test cases using specified fuzz vectors. Chen et al. [51] describe a method for security testing web applications by their page flow (i.e., which page is presented next to a user based on the input). By such page flow descriptions, they next partition the application in logical components which are then coupled to generate test traces. These test traces then are translated into test cases which are executed against the SUT during security testing. Song et al. [52] motivate a more model-related testing approach which especially focuses on the server side of web applications. Client- as well as server-side behavior of the application are described using finite state machines (FSM), which then are coupled. The resulting composition models then are used to generate test cases by depth-first traversal from the initial state of the composition FSM. The resulting test cases then are executed against the SUT. Xu et al. [43] propose the application of aspect-oriented petri-nets for security verification of software systems. Aspect-oriented petri-nets are used for constructing both a system net, i.e., a threat-driven security model that describes system (or security) goals, intended functions, security threats, and threat mitigations, and threat models (or nets) built from vulnerable transitions in the system net w.r.t. a specific threat. These threat nets further are annotated with mitigation aspects. Verification then boils down to searching for threat paths in threat nets and checking whether these paths are also possible in the corresponding system net. If so, the modeled system is vulnerable. Kiezun et al. [53] address automatic generation of XSS and SQLI attacks for web applications grounded on symbolic execution and mutation. More precisely, their tool dynamically generates inputs which are symbolically tainted through execution. Finally, these inputs are mutated to synthesize concrete exploits.

Obviously, work similar to ours exists. HTTPUnit as presented by Tappenden et al. [47] also executes HTTP requests against a SUT. Their approach, however, requires the introduction of test layers in the SUT to evaluate security. Apart from that, HTTPUnit further requires manual coding of test cases. Our method avoids this burden by relying on a semi-automatically established $\mathcal{SP}$ to then automatically generate test cases, i.e., attacks, for a web application as running in a production environment. Like ours, also the work of Kiezun et al. [53] evaluates web applications for XSS and SQLI vulnerabilities. However, in order to properly detect attacks, their tool implementation requires full access to the application's source code and its back-end database. Using techniques from model-based testing ocularly dissolves this requirement. Similar to our work, Offutt et al. [48] perform bypass testing of web applications by analyzing web pages for input elements (i.e., operations) that accept user input. Subsequently, malicious inputs are generated for these

[2] ASLan++ is a specification language for model-checkers targeted to security analysis.

input elements and submitted to the web application. Our method generally follows the same idea, however, as of our risk analysis where we match specific attack patterns against operations, resulting test cases are tailored to specific operations of the SUT instead of representing a set of penetration test cases for a web application. Our method thus allows a tester to a have a more *subjective* view on the actual vulnerabilities of a specific web application instead of a rather objective view on web application vulnerabilities in general as it would result from executing a fixed set of penetration test cases. Apart from that, using such a fixed set of penetration test cases clearly results in missing vulnerabilities that may be specific to just some application. The most similar work discussed is by Xiong and Peyton [50] which, similarly like our tool, uses security knowledge provided by databases which are maintained by security experts. However, contrary to our work, they do not apply a security risk analysis for identifying potentially vulnerable spots in the SUT but rather use input enumeration to then match those results with the database. Further, contrary to our tool, they use predefined fuzz vectors, whereas our work dynamically generates tailored attacks and test data during testing based on the outcomes of the security risk analysis.

3 Method

In the following, we discuss our method in detail, except for the web spider and the domain-specific languages (DSLs) that our tool uses, which is postponed until Sect. 3.6. This is due to that those components, viz. the web spider and the DSLs actually are only necessary for our tool, yet, our method is designed to work without them.

3.1 Security problem

The security problem ($\mathcal{SP}$) comprises a declarative system model of a web application, i.e., the SUT, at an interface level. Its name is as we consider the SUT a vulnerable piece of software; thus, it represents a "security problem". It is the main input to our testing method and its security risk analysis.

A software model can be seen as a formal abstraction of some real-world entity or process. Thus, a model can be declared as a finite, enumerable set of facts with different properties. Hence, models formalize some real-world knowledge (e.g., on some entity or process).

Logic programs are sets of rules in the form of $A \leftarrow L_1, \ldots, L_m$, where rules may have an empty body, i.e., $A \leftarrow$, subsequently called a fact. Hence, supporting both rules and facts, the semantics of logic programs obviously support dynamic and static modeling of software (e.g., entity or process modeling). Thus, using semantics of logic programs allows to define a *domain of discourse*, i.e., describing a software model with a declarative syntax.

Consider for example the following grounded, normal logic program $\Pi_{\mathcal{SP}}$. It models a potentially vulnerable PHP application exploiting an SQLI vulnerability (see Fig. 2) , e.g., $\Pi_{\mathcal{SP}} =$

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

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

Definition 1 (*Security Problem*) $\mathcal{SP}$, the security problem, describes a declarative system model, or specification, of a SUT by a logic program. It is a set of facts with terms over the alphabet $\Sigma_{\mathcal{SP}} = \langle \mathfrak{F}, \mathfrak{C}, \mathfrak{X} \rangle$, with disjoint sets of symbols

$\mathfrak{F} = \{module, uri, operation, parameter\}$
is a finite set of predicates,
$\mathfrak{C} = \{c_1, \ldots, c_m\}$ is a set of constants, and
$\mathfrak{X} = \{X_1, \ldots, X_n\}$ is a set of variables.[3]

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

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

The predicates described by set $\mathfrak{F}$ are functions over $\mathfrak{C}$ with

$module \subseteq \mathfrak{M}$ to declare a new software module of a SUT,
$uri \subseteq \mathfrak{M} \times \mathfrak{U}$ to declare the URI of the module,
$operation \subseteq \mathfrak{M} \times \mathfrak{O} \times \wp(\mathfrak{P}) \times \mathfrak{T}$ to declare some operation
of a module with its parameters and its return type, where
$\wp(\mathfrak{P})$ is the powerset over all parameters, and
$parameter \subseteq \mathfrak{M} \times \mathfrak{O} \times \mathfrak{P} \times \mathfrak{T}$ to declare a parameter of some operation with its type.[4]

[3] The alphabet of the security problem $\mathcal{SP}$ needs to support variables, for that terms formed over it, can occur ungrounded.

[4] Observe that this predicate "transforms" a parameter from a plain symbol into something we can reason about.

```
1 if (isset($_GET['login'])) {
2     $uname = $_GET['uname'];
3     $pword = $_GET['pword'];
4
5     $query = "SELECT * FROM user WHERE username
          = '$uname' AND password = '$pword'";
6     $result = mysql_query($query);
7     $num_rows = mysql_num_rows($result);
8     ...
```

Fig. 2 PHP code vulnerable to SQLI. Using a properly crafted input string, an attacker can successfully evade a programmer's intended SQL query and perform an SQLI

In a more literal sense, by Definition 1, program $\Pi_{\mathcal{SP}}$ describes a software system, e.g., a web application, which realizes a module `auth` that offers an operation `login` with two parameters, viz. `uname` of type `text`, and `pword` of type `password`. As the parameters are controllable by the user, and thus open up a vulnerability, program $\Pi_{\mathcal{SP}}$ describes a security problem $\mathcal{SP}$, i.e., the knowledge of the real-world to be formalized, or *domain of discourse*, for later security risk analysis.

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

$$\mathfrak{M} = \{auth\},$$
$$\mathfrak{O} = \{login\},$$
$$\mathfrak{P} = \{uname, pword\},$$
$$\mathfrak{T} = \{text, password\}, \text{ and}$$
$$\mathfrak{U} = \{\text{“}http://www.victim.com\text{”}\}.$$

3.2 Vulnerability knowledge base

The vulnerability knowledge base (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 extensional database) for generating test data. Figure 3 shows its internal architecture.

The extensional database or knowledge base (Sect. 3.2.1) 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 [54]) during the security risk analysis and the risk assessment to satisfy the rules, i.e., logical predicates, of the intensional database (Sect. 3.2.2) to establish a risk profile on the grounds of a provided security problem. Finally, we use definite-clause grammars to codify the structure of test data definitions w.r.t. known exploits (Sect. 3.2.3). These test data definitions are necessary to later generate malicious inputs during testing to verify existing vulnerabilities in the SUT.

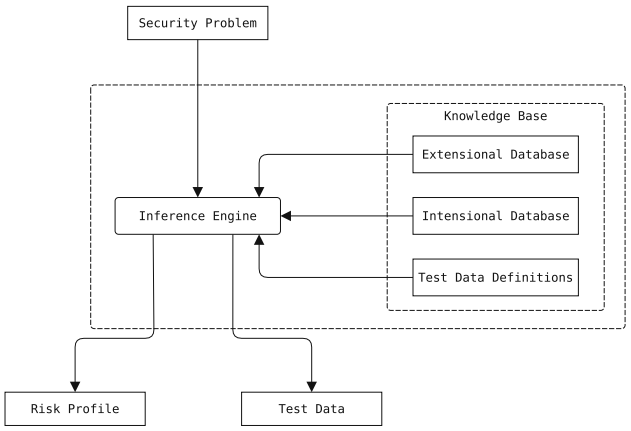


Fig. 3 Internal architecture of the VKB. The $\mathcal{EDB}$ provides the necessary knowledge for the $\mathcal{IDB}$. By test data definitions, we define the structure of test data w.r.t. known exploits in the $\mathcal{EDB}$. Observe that the inference engine conducts the security risk analysis and test data generation as indicated in Fig. 1 under step (2)

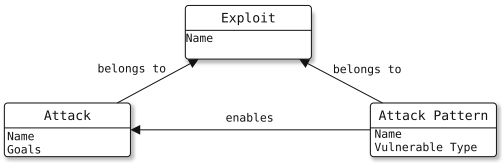


Fig. 4 Relationship of exploit, attack and attack pattern as codified in the $\mathcal{EDB}$. Each exploit has associated different attacks (i.e., variations thereof) and attack patterns

3.2.1 Extensional database

The extensional database ($\mathcal{EDB}$) (also called extensional knowledge base) forms the memory of our VKB. It stores security vulnerability knowledge, i.e., exploits, attacks and attack patterns as shown in Fig. 4. Exploits have a name and associated a set of attacks, as well as attack patterns. Each attack declares a name on its own and a set of potential attack goals 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.

The knowledge of the $\mathcal{EDB}$ is declared by logic programs over an alphabet $\Sigma_{\mathcal{EDB}}$ (Definition 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 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 $\Sigma_{\mathcal{EDB}} = \langle \mathfrak{F}, \mathfrak{C}, \mathfrak{X} \rangle$, with disjoint sets of symbols

$\mathfrak{F} = \{exploit, attack, vul\_type, attack\_pattern\}$ is a finite set of predicates,
$\mathfrak{C} = \{c_1, \ldots, c_m\}$ is a set of constants, and
$\mathfrak{X} = \{X_1, \ldots, X_n\}$ is a set of variables.[5]

Further, $\mathfrak{C} = \langle \mathfrak{E}, \mathfrak{A}, \mathfrak{AP}, \mathfrak{G}, \mathfrak{T} \rangle$, with disjoint sets of symbols, where

$\mathfrak{E}$ is a set of exploits,
$\mathfrak{A}$ is a set of attacks
$\mathfrak{AP}$ is a set of attack patterns,
$\mathfrak{G}$ is a set of attack goals, and
$\mathfrak{T}$ is a set of types. [6]

The predicates described by set $\mathfrak{F}$ are functions over $\mathfrak{C}$ with

$exploit \subseteq \mathfrak{E}$ to declare an exploit,
$attack \subseteq \mathfrak{E} \times \mathfrak{A} \times \mathfrak{AP} \times \wp(\mathfrak{G})$ to declare an attack under an exploit, its attack pattern and goals
$vul\_type \subseteq \mathfrak{A} \times \mathfrak{T}$ to declare a type, exploitable by an attack, and
$attack\_pattern \subseteq \mathfrak{AP} \times \mathfrak{A} \times \mathfrak{M} \times \mathfrak{O} \times \mathfrak{P}$ to declare an attack pattern for an attack.

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

$$
\begin{aligned}
&exploit(sql\_attack),\\
&attack(sql\_attack, signature\_evasion, sqlap,\\
&\quad [authentication, leakage, tampering]),\\
&vul\_type(sql\_attack, text),\\
&vul\_type(sql\_attack, password),\\
&attack\_pattern(sqlap, sql\_attack, X_{\mathfrak{M}}, X_{\mathfrak{O}}, X_{\mathfrak{P}}) \leftarrow\\
&module(X_{\mathfrak{M}}),\\
&\quad operation(X_{\mathfrak{M}}, X_{\mathfrak{O}}, \_, \_),\\
&\quad parameter(X_{\mathfrak{O}}, X_{\mathfrak{P}}, X_{\mathfrak{T}}),\\
&\quad vul\_type(sql\_attack, X_{\mathfrak{T}}).
\end{aligned}
$$

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

From the predicates of the $\mathcal{EDB}$, `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 $\mathcal{EDB}$, as, contrary to the rules of the $\mathcal{IDB}$, `attack_pattern/5` cannot be declared "generic". For this, see again program $\Pi_{\mathcal{EDB}}$. 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., $X_{\mathfrak{M}}$, $X_{\mathfrak{O}}$, $X_{\mathfrak{P}}$ can be bound, a security problem $\mathcal{SP}$ needs to be present.

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

$\mathfrak{E} = \{sql\_attack\}$,
$\mathfrak{A} = \{signature\_evasion\}$,
$\mathfrak{AP} = \{sqlap\}$,
$\mathfrak{G} = \{authentication, leakage, tampering\}$, and
$\mathfrak{T} = \{text, password\}$.

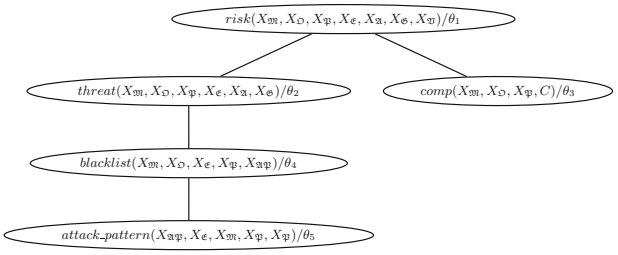
3.2.2 Intensional database

The intensional database ($\mathcal{IDB}$) (also called intensional knowledge base) 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 $\mathcal{EDB}$ against a security problem $\mathcal{SP}$ to deduce a risk profile $\mathcal{RP}$ that is, to infer *new* knowledge. Figure 5 illustrates this deduction procedure for one concrete risk based on the security problem $\mathcal{SP}$ from program $\Pi_{\mathcal{SP}}$ (Sect. 3.1), the $\mathcal{EDB}$ from program $\Pi_{\mathcal{EDB}}$ (Sect. 3.2.1) and substitutions $\theta_1, \theta_2, \theta_3, \theta_4$ and θ_5. Further, Fig. 5 introduces the main predicates of our risk analysis, viz. `blacklist/6`, `threat/6` and `comp/4`.

For deducing a risk profile $\mathcal{RP}$ (Sect. 3.3), our security risk analysis returns with multiple grounded instances of the `risk/7` predicate, i.e., potential risks in a security problem $\mathcal{SP}$ (see top of Fig. 5 for one such grounded instance with substitution θ_1). First, a solver (GNU Prolog [54]) unifies facts declared by a security problem $\mathcal{SP}$ with attack patterns from the $\mathcal{EDB}$, i.e., free variables of the `attack_pattern/5` predicate are bound to constants of the security problem $\mathcal{SP}$. 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 $\mathcal{SP}$. The `blacklist/5` predicate unifies information regarding a vulnerable operation ($X_{\mathfrak{O}}$) and the corresponding parameter ($X_{\mathfrak{P}}$) of a software module ($X_{\mathfrak{M}}$) with the matching attack pattern ($X_{\mathfrak{AP}}$) and its corresponding exploit ($X_{\mathfrak{E}}$).

[5] Again, variables are necessary for that terms can occur ungrounded (in particular for `attack_pattern/5`).

[6] Observe that this set is equal to set $\mathfrak{T}$ from Definition 1.



$\theta_1 = \{X_{\mathfrak{M}} \mapsto auth, X_{\mathfrak{O}} \mapsto login, X_{\mathfrak{P}} \mapsto uname, X_{\mathfrak{E}} \mapsto sql\_attack, X_{\mathfrak{A}} \mapsto signature\_evasion, X_{\mathfrak{G}} \mapsto [authentication, leakage, tampering], X_{\mathfrak{V}} \mapsto (high, low, medium)\}$

$\theta_2 = \{X_{\mathfrak{M}} \mapsto auth, X_{\mathfrak{O}} \mapsto login, X_{\mathfrak{P}} \mapsto uname, X_{\mathfrak{E}} \mapsto sql\_attack, X_{\mathfrak{A}} \mapsto signature\_evasion, X_{\mathfrak{G}} \mapsto [authentication, leakage, tampering]\}$

$\theta_3 = \{X_{\mathfrak{M}} \mapsto auth, X_{\mathfrak{O}} \mapsto login, X_{\mathfrak{P}} \mapsto \{uname, pword\}, X_{\mathfrak{E}} \mapsto sql\_attack, C \mapsto high\}$

$\theta_4 = \{X_{\mathfrak{M}} \mapsto auth, X_{\mathfrak{O}} \mapsto login, X_{\mathfrak{P}} \mapsto uname, X_{\mathfrak{E}} \mapsto sql\_attack, X_{\mathfrak{AP}} \mapsto sqlap\}$

$\theta_5 = \{X_{\mathfrak{AP}} \mapsto sqlap, X_{\mathfrak{E}} \mapsto sql\_attack, X_{\mathfrak{M}} \mapsto auth, X_{\mathfrak{O}} \mapsto login, X_{\mathfrak{P}} \mapsto uname\}$

Fig. 5 Deduction of a risk as part of inferring a risk profile $\mathcal{RP}$ by substitutions $\theta_1, \theta_2, \theta_3, \theta_4$, and θ_5. The key predicates are `attack_pattern/5`, `blacklist/5`, `threat/6`, and `comp/4`. The resulting risk profile $\mathcal{RP}$ is a set of grounded instances of the `risk/7` predicate

parameter: Security problem $\mathcal{SP}$, attacks $\mathfrak{A}$, attack patterns $\mathfrak{AP}$
result : Risk profile $\mathcal{RP}$

```
1:  procedure RISK_ANALYSIS(SP, A, AP)
2:      OP' ← ∅
3:      RP ← ∅
4:      for each op ∈ SP do
5:          if ∃ap ∈ AP: ap μ op then
6:              OP' ← OP' ∪ op
7:          end if
8:      end for
9:      for each op ∈ OP' do
10:         A_op ← {a : ap μ op ∧ ap σ a : ap ∈ AP, a ∈ A}
11:         for each a ∈ A_op do
12:             thrt ← (op, a)
13:             risk ← (thrt, ASSESS(thrt))
14:             RP ← RP ∪ risk
15:         end for
16:     end for
17:     return RP
18: end procedure
```

Fig. 6 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 $\mathcal{RP}$

For each blacklisted operation, the `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 `threat/6` predicate unifies all the information necessary to later generate a risk, i.e., the software module's ($X_{\mathfrak{M}}$), operation's ($X_{\mathfrak{O}}$) and parameter's name ($X_{\mathfrak{P}}$), as well as the exploit ($X_{\mathfrak{E}}$), a concrete attack ($X_{\mathfrak{A}}$) and potential attack goals ($X_{\mathfrak{G}}$). A concrete attack is inferred from the knowledge of the $\mathcal{EDB}$ that "knows" about various attacks w.r.t. an exploit. As of Prolog's solution strategy, each concrete attack, i.e., instance of the `attack/4` predicate, will be unified with instances of the `threat/6` predicate. For example, for the security problem $\mathcal{SP}$ from program $\Pi_{\mathcal{SP}}$ (Sect. 3.1) and the $\mathcal{EDB}$ from program $\Pi_{\mathcal{EDB}}$ (Sect. 3.2.1), this would yield a total of two instances of the `threat/6` predicate due to the contents of the security problem $\mathcal{SP}$, viz. one operation (`login`) with two parameters (`uname` and `pword`), and one declared attack in the $\mathcal{EDB}$, 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. `comp/4` calculates the operation complexity *C* by recursively iterating over the list of parameters ($X_{\mathfrak{P}}$) of an operation ($X_{\mathfrak{O}}$) declared by a software module ($X_{\mathfrak{M}}$). Finally, if `threat/6` and `comp/4` are fully grounded, the `risk/7` predicate (Sect. 3.3), 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 6, 7 and 8 give an algorithmic description of this deduction process, in future referred to as program $\Pi_{\mathcal{IDB}}$.

Figure 6 lists the main procedure, i.e., `risk_analysis`, of the risk analysis algorithm. Its inputs are (i) the security problem $\mathcal{SP}$, (ii) the set of known attacks $\mathfrak{A}$, and (iii) the set of related attack patterns $\mathfrak{AP}$. For each operation $op \in \mathcal{SP}$ it evaluates, whether there exists an attack pattern $ap \in \mathfrak{AP}$ that matches the attack vector, described by the operation's input parameters, i.e., $ap\ \mu\ op$ (read "ap matches op" with $\mu \subseteq \mathfrak{AP} \times \mathfrak{O}$). If a matching attack pattern is found, operation *op* is added to the set $\mathcal{OP}'$ of blacklisted operations. Next, in line 9, for each of the blacklisted operations of set $\mathcal{OP}'$, all applicable attacks *a* are collected in the set $\mathcal{A}_{op}$, i.e., $ap\ \mu\ op \wedge ap\ \sigma\ a$ (read "ap matches op and ap subsumes a" with $\sigma \subseteq \mathfrak{AP} \times \mathfrak{A}$). Then, for each operation-attack pair, a threat *thrt* is created which then is evaluated by calling the `assess` procedure (Fig. 7). Its return value is used to create a new `risk`, which finally is added to the set $\mathcal{RP}$, which describes the result of the risk analysis, the inferred 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 $\mathcal{EDB}$. Thus, our risk analysis may only detect what

```
parameter: Threat thrt
result   : Risk assessment asmnt
19: procedure ASSESS(thrt)
20:     imp ← COMP(thrt → op → p) ⊕ SEVERITY(thrt → a)
21:     prob ← PROBABILITY(impact)
22:     rl ← RISK_LEVEL(impact, probability)
23:     asmnt ← (imp, prob, rl)
24:     return asmnt
25: end procedure
```

Fig. 7 Risk assessment procedure of the risk analysis algorithm. It deduces for each operation and attack pair a risk assessment by the operation complexity

```
parameter: List of parameters ℒ
result   : Operation complexity c
26: procedure COMP(ℒ)
27:     if ℒ = ∅ then
28:         return low
29:     else
30:         param ← TAKE(ℒ)
31:         type ← TYPE_OF(param)
32:         Δcomp ← COMP(type)
33:         return Δcomp ⊕ COMP(ℒ)
34:     end if
35: end procedure
```

Fig. 8 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

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 $\mathcal{EDB}$).

As just discussed, the `risk_analysis` procedure calls the `assess` procedure (Fig. 7) to evaluate a threat $thrt$. For this, it receives a threat $thrt$ as input parameter. The `assess` procedure then first calls the `comp` procedure to calculate the complexity of the threat's subjected operation (Fig. 8). Based on the returned complexity value c_{op}, the `assess` procedure next calculates the potential impact of threat $thrt$'s underlying attack, the probability of succeeding with the attack and, based on those two former values, the overall risk level for threat $thrt$.

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 of *high*, i.e., $c_{op} = high$ 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, as example, on the one side, an operation with a set of complex parameters (e.g., objects) which require parsing by the application and, on the other side, an operation with only a few trivial parameters (e.g., a numeric values). For the first case, i.e., an operation with a set of complex parameters, a malicious agent needs to know quite well, where to put his malicious input, for that it causes its impact, thus a lower probability. 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., an operation with a set of trivial parameters, a malicious agent may succeed with an attack quite easily, e.g., simply by overflowing any input parameters or trigger a subsequent computation to overflow. Yet, the impact maybe little, as such an attack 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 (Table 1). 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 $\{very\_low, low, medium, high, very\_high\}$ (Table 1). 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.

The last procedure to be discussed as part of our security risk analysis is `comp` (Fig. 8). As already discussed, it is called by the `assess` procedure for calculating an operation 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 [12,13]. Equation 1 shows the underlying formula of our complexity calculation. The complexity factor c then reflects the complexity of an operation op by its signature and is denoted $c(op)$. The more complex the set of input parameter types, the higher is the overall complex-

Table 1 Look-up table for risk assessment

Impact		Very low	Low	Medium	High	Very high
Probability	Very low	Very low	Low	Low	Low	Medium
	Low	Low	Low	Medium	Medium	Medium
	Medium	Low	Medium	Medium	High	High
	High	Low	Medium	High	High	High
	Very high	Medium	High	High	High	Very high

Based on given impact and probability values, the `assess` procedure calculates the risk level by consulting this look-up table

ity 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 op with the operation signature $op(p_1 : t_1, \ldots, p_n : t_n)$ then is the sum of the input parameter type complexities $c(t_i)$ and the dependence factor $d(t_1, \ldots, t_n)$.

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

Figure 8 shows the algorithm used for calculating the operation's complexity $c(op)$. If the list of parameters is empty, the `comp` procedure simply returns with a complexity value of low. Contrary, if it contains any element, `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 op's complexity (line 33). For this, the operator $\oplus$ infers the new complexity to be returned based on the actual parameter's complexity Δ_{comp} and the complexity value returned by the recursive call to `comp`. In our implementation, the operator $\oplus$ is implemented by the `dcomp/3` predicate, which also uses the look-up table from Table 1 for inferring the new complexity based on its two inputs, as just discussed. After the last recursive call has returned, `comp` calculates the final operation complexity $c(op)$, again in line 33, and returns with this value.

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 $\mathcal{EDB}$. The new knowledge manifests itself in the risk profile $\mathcal{RP}$, 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.

3.2.3 Test data

We use definite-clause grammars (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. 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., SQLI 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 introduces the language to be used by logic programs, which describe DCGs for test data generation in our VKB.

Definition 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 $\Sigma_{\mathcal{TDS}} = \langle \mathfrak{F}, \mathfrak{C}, \mathfrak{X} \rangle$, with disjoint sets of symbols.

$\mathfrak{F} = \{testdata\}$ is the initial set of predicates,
$\mathfrak{C} = \{c_1, \ldots, c_m\}$ is a set of constants, and
$\mathfrak{X} = \{X_1, \ldots, X_n\}$ is a set of variables. [7]

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

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

[7] Again, variables are necessary for that production rules of the grammar (i.e., terms) can occur ungrounded.

where sets $\mathfrak{E}, \mathfrak{A}$, and $\mathfrak{T}$ are equal with their identically labeled counterparts from Definitions 1 and 2.
The predicate described by the initial set $\mathfrak{F}$ is a function over $\mathfrak{C}$ with

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

Contrary to $\Sigma_{\mathcal{SP}}$ and $\Sigma_{\mathcal{EDB}}$, the set $\mathfrak{F}$ of predicates of alphabet $\Sigma_{\mathcal{TDS}}$ is not finite. Thus, initially, it only contains `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 $\mathfrak{F}$ are purely dependent on the test data to be generated. Put another way, for $\Sigma_{\mathcal{TDS}}$, we only require the predicate `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 $\Pi_{\mathcal{EDB}}$ in Sect. 3.2.3). This would require a grammar that allows to produce input strings like, e.g., `' OR '1' = '1'–`. Program $\Pi_{\mathcal{TDS}}$ describes a DCG that allows for that, e.g., $\Pi_{\mathcal{TDS}} =$

$$\begin{aligned}
&testdata(sql_attack, signature_evasion, _, _, S0, S) \leftarrow \\
&\quad apostrophe(S0, S1), or(S1, S2), apostrophe(S2, S3), \\
&\quad number(S3, S4), apostrophe(S4, S5), equals(S5, S6), \\
&\quad apostrophe(S6, S7), number(S7, S8), \\
&\quad apostrophe(S8, S9), comment(S9, S), \\
&\quad apostrophe(S0, S) \leftarrow connects(S0, ', S), \\
&\quad number(S0, S) \leftarrow connects(S0, 1, S), \\
&\quad equals(S0, S) \leftarrow connects(S0, =, S), \\
&\quad comment(S0, S) \leftarrow connects(S0, --, S), \\
&\quad or(S0, S) \leftarrow connects(S0, OR, S), \\
&\quad or(S0, S) \leftarrow connects(S0, ||, S).
\end{aligned}$$

Program $\Pi_{\mathcal{TDS}}$,[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 $\Pi_{\mathcal{TDS}}$ works for at least the four mentioned database vendors, the set of rules would need to be refined and extended. It is exactly for this reason why set $\mathfrak{F}$ of alphabet $\Sigma_{\mathcal{TDS}}$ is not finite and solely provides `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 `testdata//4` as the main goal predicate.

[8] The underscore, i.e., "_", indicates the use of a wildcard, i.e., during deduction of a solution, these parameters remain unbounded, i.e., are "neglected".

```
1 testdata(sql_attack,signature_evasion,_,_) -->
      apostrophe,or,apostrophe,number,apostrophe,
      equals,apostrophe,number,apostrophe,comment
      .
2 apostrophe --> "'".
3 equals --> "=".
4 number --> "1".
5 or --> "OR" ; "||".
6 comment --> "--".
```

Fig. 9 Program $\Pi_{\mathcal{TDS}}$ reproduced in Prolog DCG notation. Prolog DCG notation more or less resembles the structure of BNF, a common notation for CFG and CSGs

```
1 testdata(sql_attack,signature_evasion,_,X1) -->
      {isMySQL(X1)},sql_evade,space ; {not(
      isMySQL(X1))},sql_evade.
2 sql_evade --> apostrophe,or,apostrophe,number,
      apostrophe,equals,apostrophe,number,
      apostrophe,comment.
3 isMySQL(mysql).
4 apostrophe --> "'".
5 equals --> "=".
6 number --> "1".
7 or --> "OR" ; "||".
8 comment --> "--".
9 space --> " ".
```

Fig. 10 Extensions to program $\Pi_{\mathcal{TDS}}$ to work for the four mentioned database vendors (MsSQL, MySQL, PostgreSQL and Oracle)

At first sight, the grammar described by program $\Pi_{\mathcal{TDS}}$ 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 $\Pi_{\mathcal{TDS}}$, as shown in Fig. 9.

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 $\Pi_{\mathcal{TDS}}$ to generate test data to work for all four mentioned database vendors. Figure 10 shows the resulting logic program.

For program $\Pi_{\mathcal{TDS}}$ from Fig. 10, set $\mathfrak{C}$ of alphabet $\Sigma_{\mathcal{TDS}}$ contains the disjoint sets

$$\begin{aligned}
\mathfrak{E} &= \{sql_attack\}, \\
\mathfrak{A} &= \{signature_evasion\}, \\
\mathfrak{I} &= \{testdata, sql_evade\}, \\
\mathfrak{T} &= \emptyset,
\end{aligned}$$

$$\mathfrak{TS} = \{1, OR, ||, =, --, ', \quad\}.$$

For the sake of clarity, we also give the final set $\mathfrak{F}$ of predicates for program $\Pi_{\mathcal{TDS}}$ (remember that $\mathfrak{F}$ as to Definition 3 initially only contained $testdata$) that is

$$\mathfrak{F} = \{testdata, apostrophe, number, connects, equals, comment, or\}.$$

Clearly, our VKB requires some effort to be established and maintained, especially in case of the $\mathcal{EDB}$. This is where a security expert comes into play, i.e., by establishing the knowledge of the $\mathcal{EDB}$ and necessary test data definitions. However, once established it can be easily distributed to various tester's sites and maintained by one or more security experts by drawing back on versioning techniques [55].

3.3 Risk profile

The risk profile ($\mathcal{RP}$) constitutes the output of the security risk analysis. It is a set of grounded instances of the `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 `risk/7` predicate declares a potential risk, identified for a security problem $\mathcal{SP}$ during deduction of the risk profile $\mathcal{RP}$. The terms formed with grounded instances of the `risk/7` predicate range over elements of the sets of set $\mathfrak{C}$ of alphabet $\Sigma_{\mathcal{RP}}$ as introduced by Definition 4, i.e., variables X_i are grounded with elements of sets of set $\mathfrak{C}$ of alphabet $\Sigma_{\mathcal{RP}}$.

Definition 4 (*Risk Profile*) $\mathcal{RP}$, the risk profile, describes a declarative risk model by a logic program. It is a set of facts with terms over the alphabet $\Sigma_{\mathcal{RP}} = \langle \mathfrak{F}, \mathfrak{C}, \mathfrak{X} \rangle$, with disjoint sets of symbols

$\mathfrak{F} = \{risk\}$ is a finite set of predicates,
$\mathfrak{C} = \{c_1, \ldots, c_m\}$ is a set of constants, and
$\mathfrak{X} = \{X_1, \ldots, X_n\}$ is a set of variables. [9]

Further, $\mathfrak{C} = \langle \mathfrak{M}, \mathfrak{O}, \mathfrak{E}, \mathfrak{A}, \mathfrak{G}, \mathfrak{P}, \mathfrak{T}, \mathfrak{V} \rangle$, with disjoint sets of symbols, where

$\mathfrak{M}$ is a set of modules,
$\mathfrak{O}$ is a set of operations,
$\mathfrak{E}$ is a set of exploits,
$\mathfrak{A}$ is a set of attacks
$\mathfrak{G}$ is a set of attack goals,
$\mathfrak{P}$ is a set of parameters, and
$\mathfrak{V}$ 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 $\mathfrak{V}$, are equal with their identically labeled counterparts from Definitions 1 and 2.
The predicates described by set $\mathfrak{F}$ are functions over $\mathfrak{C}$ with

$$risk \subseteq \mathfrak{M} \times \mathfrak{O} \times \mathfrak{E} \times \mathfrak{A} \times \mathfrak{G} \times \mathfrak{P} \times \mathfrak{V} \text{ to declare a risk.}$$

As Definition 4 states, instances of the `risk/7` predicate in the risk profile $\mathcal{RP}$ are grounded with constants declared by both the $\mathcal{SP}$ and the $\mathcal{EDB}$ by applying deduction rules, declared in the $\mathcal{IDB}$. Hence, it is "learned" by our VKB from the $\mathcal{SP}$ and the $\mathcal{EDB}$ and comprises new knowledge. For that this knowledge is sound and complete, we have defined the disjoint subsets of the sets of constants $\mathfrak{C}$ of alphabets $\Sigma_{\mathcal{SP}}$, $\Sigma_{\mathcal{EDB}}$ and $\Sigma_{\mathcal{RP}}$ from Definitions 1, 2, and 4 to be equal (Definition 4) to establish the common domain of discourse.

Program $\Pi_{\mathcal{RP}}$ shows a risk profile $\mathcal{RP}$ as inferred by the security risk analysis using knowledge declared by programs $\Pi_{\mathcal{SP}}$ and $\Pi_{\mathcal{EDB}}$ and the deduction procedure as described by program $\Pi_{\mathcal{IDB}}$ (Figs. 6, 7, 8), e.g., $\Pi_{\mathcal{RP}} =$

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

Program $\Pi_{\mathcal{RP}}$ describes a risk profile for the security problem $\mathcal{SP}$ from program $\Pi_{\mathcal{SP}}$. 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 that they both entail the same exploit and attack, e.g., `sql_attack` and `signature_evasion`.

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

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

[9] The alphabet of the risk profile $\mathcal{RP}$ needs to support variables, for that terms formed over it, can occur ungrounded.

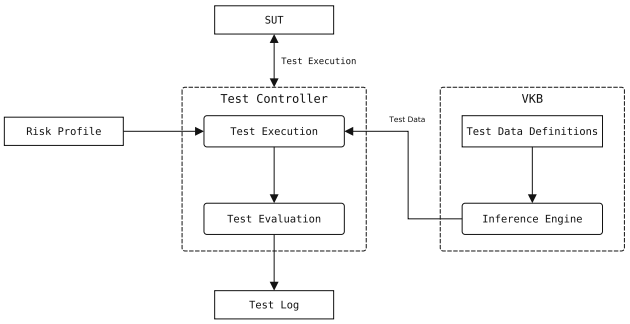


Fig. 11 Abstract overview of test execution. The test controller takes as an input a risk profile $\mathcal{RP}$ to execute it against the SUT. For each test outcome, a verdict is derived during test evaluation. Test execution returns with a test log

The deduced risk profile $\mathcal{RP}$ subsumes a necessary *negative* specification. This is due to its content that describes potential side-effect functionality, i.e., vulnerabilities, in the SUT by *negative* requirements. Remember that these negative requirements are established as part of our security risk analysis.

3.4 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 Fig. 11. Our test engine then generates in-memory test cases, i.e., executable test objects, which are executed against a SUT.

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
- `selection` to set the test data selection strategy based on what was retrieved from the Prolog solver (i.e., the inference engine)[10], 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 12 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 risk profile $\mathcal{RP}$. For each risk r, the algorithm then above all 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

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

```
1:  procedure TEST(RP)
2:      for each r ∈ RP do
3:          if r → risk_level ≥ threshold then
4:              d ← TEST_DATA(r → a)
5:              if selection = all then
6:                  n ← d → size
7:              else
8:                  n ← executions
9:              end if
10:             for n do
11:                 result ← INVOKE(r → m, r → o, d)
12:                 verdict ← EVALUATE(result, r → a, r → g)
13:                 LOG(verdict)
14:             end for
15:         end if
16:     end for
17:     return L
18: end procedure
```

Fig. 12 Test execution algorithm

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 (see also Sect. 3.6). 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. The `invoke` procedure 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. Thus, our tool implementation, as later discussed in detail in Sect. 3.6, can be applied on different software system due to this generic adaption mechanism. However, we will not go into more detail in discussing the `invoke` procedure and its implementation, as we already applied and discussed similar techniques earlier [56].

[10] Remember that Prolog returns with every possible solution.

3.5 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 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 with distinct outcome characteristics, viz.

- `authentication violation`,[11] 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 (Fig. 13) or XSS attack (Fig. 14) was successful.

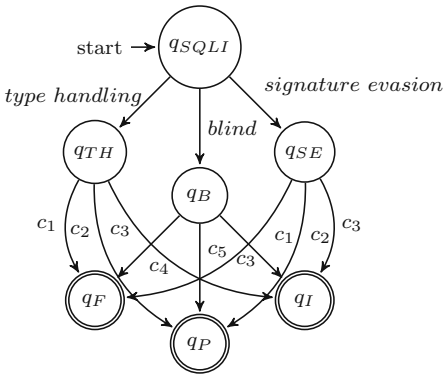
3.5.1 Detecting successful SQLI attacks

Figure 13 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., (i) SQLI by signature evasion or (ii) incorrect type handling and (iii) blind SQLI. Upon receiving a response r from the SUT after executing an SQLI attack, the state machine enters the state q_{SQLI}. Next, based on the type of SQLI attack that has been executed, the state machine transitions to the corresponding state, viz. (i) q_{SE}, (ii) q_B or (iii) q_{TH}. Then, given certain conditions as discussed below, the final verdict is inferred by entering one of the final states q_F, q_P or q_I corresponding to the three verdicts `FAIL`, `PASS` and `INCONCLUSIVE`.

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., $r \neq null$), r must not contain any error messages ($r \neq error$),[12] 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 ($i \notin r$). In case the response r is `null` ($r = null$) or the original page remains displayed ($r = o$), i.e., c_3 is satisfied, the verdict is `INCONCLUSIVE`. The fact that no error has

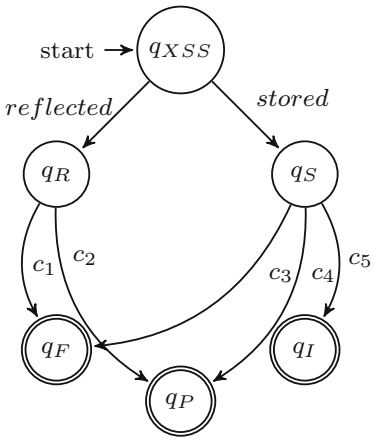
[11] For reasons of brevity, this goal is abbreviated with just `authentication`.

[12] We put a special focus on SQL specific error messages, e.g., "`Incorrect syntax near`" or "`Unclosed quotation mark`".



Label	*Condition*
c_1	$\neg c_2 \wedge \neg c_3$
c_2	$(r \neq null) \wedge (r \neq error) \wedge (r \neq o) \wedge (i \notin r)$
c_3	$(r = null \vee r = o)$
c_4	$\neg c_5 \wedge \neg c_3$
c_5	$c_2 \vee (d \geq t)$

Fig. 13 State machine for detecting successful SQLI attacks (Obviously we could have optimized this state machine by merging states q_{TH} and q_{SE}. Yet, for the sake of readability, we did not do so). The table shows the necessary conditions for that transitions in the state machine can be made



Label	*Condition*
c_1	$\neg c_2$
c_2	$(r \neq null) \wedge (r \neq error) \wedge (i \in r)$
c_3	$(\neg c_4 \wedge \neg c_5) \vee ((i \in r) \wedge (i \notin r_1))$
c_4	$(r \neq null) \wedge (r \neq error) \wedge (i \in r_1)$
c_5	$(r \neq error) \wedge (i \notin r) \wedge (i \notin r_1)$

Fig. 14 State machine for detecting successful XSS attacks. The table shows the necessary conditions for that transitions in the state machine can be made

been thrown, and further, the input apparently has been consumed, prohibits to derive `FAIL` as a verdict. However, if both c_2 and c_3 are unsatisfiable, the verdict is `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 `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 that also reflect injection strings that attempt to trigger a delay in the SUT. 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 `INCONCLUSIVE` if c_3 is satisfied. Finally, if c_4 is satisfied, the verdict is `FAIL`.

For SQLI by incorrect 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 `PASS`. If the response is null or equals the original page (c_3) the verdict is `INCONCLUSIVE`. Finally, if either c_2 or c_3 are satisfied, the verdict is `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.

3.5.2 Detecting successful XSS attacks

Figure 14 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., (i) reflected and (ii) stored XSS. Upon receiving a response r from the SUT after executing an XSS attack, the state machine enters the state q_{XSS}. Next, based on the type of XSS attack that has been executed, the state machine transitions to the corresponding state, viz. (i) q_R, (ii) q_S. Then, given certain conditions as discussed below, the final verdict is inferred by entering one of the final states q_F, q_P or q_I corresponding to the three verdicts `FAIL`, `PASS` and `INCONCLUSIVE`.

Detecting a successful reflected XSS attack is fairly simple. Given that the SUT returns data ($r \neq null$) that does not contain an error ($r \neq error$) 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.

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 ($r \neq null$), contain no error message ($r \neq error$) and further, the second response must reflect the injection string ($i \in r_1$). If this is the case, our method detects a successful stored XSS attack. If, however, the injection string is not contained in the first and second response ($(i \notin r) \wedge (i \notin r_1)$) and the first response further does not contain an error ($r \neq error$), 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 the injection string ($(i \in r) \wedge (i \notin r_1)$). 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 `FAIL`, as the right part of c_3 is satisfied. Clearly, if c_4 and c_5 both cannot be satisfied, i.e., $\neg c_4 \wedge \neg c_5$, the verdict also is `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 $\mathcal{RP}$ (Sect. 3.6).

One could argue now that such an evaluation procedure may yield false positives and false negatives. This may be true, however, 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 (Sect. 5) shows that our test evaluation procedure is effective by detecting successful attacks by monitoring the application's response. As a final note, we want to point out that the state-machines are defined in a SUT agnostic manner to provide as much genericity as possible.

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.

3.6 Tool implementation

In the following, we introduce the tool specific extension for our method.[13] These comprise a set of modeling facilities, viz. two DSLs providing human-readable abstractions to our declarative models, e.g., the security problem $\mathcal{SP}$ and the risk profile $\mathcal{RP}$ and necessary model translators.

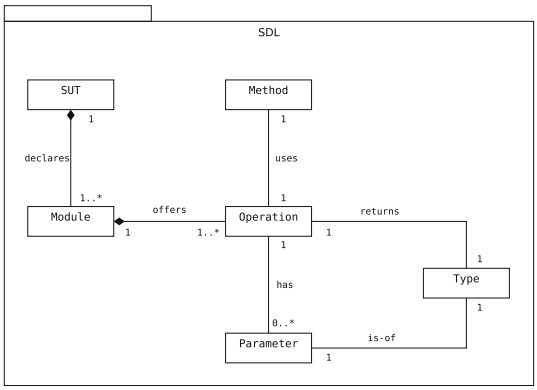


Fig. 15 Abstract syntax of our system description language (SDL). Our language for describing a web application as a SUT is kept small and concise. This is as in black-box testing we do not know much about the system that we can state with certainty

3.6.1 System DSL

The system DSL (or SDL) formally declares syntactical modeling elements for describing a SUT (or security problem $\mathcal{SP}$) in a textual manner. 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[14] of the web application (subsequently referred to as `Module` in our tool). Figure 15 shows the abstract syntax (or metamodel) of our SDL.

Observe that there exists a one-to-one correspondence between elements of the SDL and elements of the $\mathcal{SP}$ from Definition 1. We thus skip any further discussion of the SDL as its syntax and semantics should be self-explanatory by now.

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. For this our tool slots a web spider ahead of actually starting the testing process (e.g., the security risk analysis and subsequent activities) for semi-automatically generating the $\mathcal{SP}$.

Web Spider The goal of our web spider is to automatically establish a model of a SUT, i.e., a web application in PHP/SQL. We did not reimplement the web spider from scratch, but instead used an already existing one written in Java, viz. Crawler4j [57]. For establishing a model of a SUT at an interface level, we configured Crawler4J to search pages for HTML input forms, i.e., the interface to the outside world. Further, it follow any links contained inside a page it

[13] The core components of our tool are available for download at http://qe-informatik.uibk.ac.at/vkb.

[14] Content folders may contain images or other resources, web sites or further content folders.

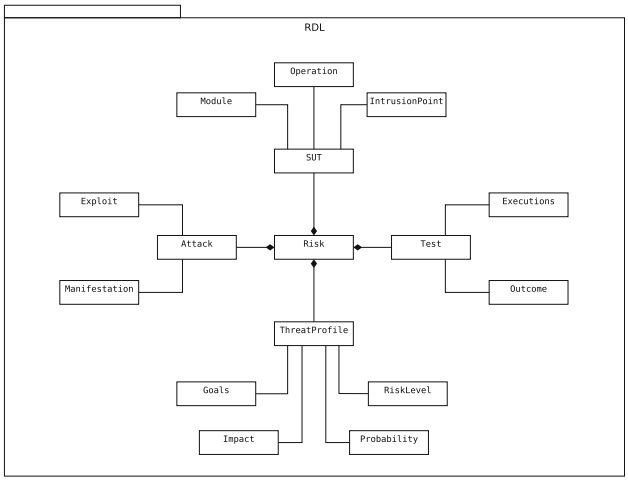


Fig. 16 Abstract syntax of our risk description language (RDL). Our RDL allows to formulate a risk profile $\mathcal{RP}$ w.r.t. a SUT. The risk profile $\mathcal{RP}$ represents both a test suite and executable specification for non-functional security testing. Further, the risk profile $\mathcal{RP}$ also subsumes a negative specification that is necessary for non-functional security testing

searched, yet with the constraint that the link is within the same domain as the base address of the SUT. The results of a "crawl" are stored in an XML file which is then further processed to generate the declarative system model as required by our method, i.e., the security problem $\mathcal{SP}$. It may occur however that some manual clean-up of the model is necessary, as discussed in Sect. 5.

The choice to use XML as the fundamental representation of our system model is that most model-based testing tools and approaches use XML (more precisely, XMI, a dialect of XML) as storage and interchange format. Thus, our tool can easily be combined with features of model evolution [58,59], or model-based regression testing [60,61].

3.6.2 Risk DSL

Contrary to our SDL that declared a rather small syntax, our risk DSL (or 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 subsumes a negative specification that is necessary for non-functional security testing. Figure 16 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 by virtue of 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. Finally, the `ThreatPro-file` 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.

We already mentioned that our risk profile $\mathcal{RP}$, besides representing both an executable specification and a test suite, also subsumes a necessary negative specification. We define a negative specification to be a set of negative requirements that states side-effect functionality (e.g., an operation allows to perform something illicit) potentially realized by the SUT. Each negative requirement is a quadruple of exploit, attack, operation, and parameter, or intrusion point. As each risk of a risk profile $\mathcal{RP}$ contains this information, the risk profile $\mathcal{RP}$ thus also subsumes such a negative specification.

Our two DSLs were implemented in the Scala programming language. Further, both DSLs follow the design principle that testers neglect any details of the target platform and other implementation specifics during modeling [62]. Concrete examples of our DSLs are shown later in Sect. 5 as part of our evaluation of the introduced method and its tool implementation.

So far, we have introduced the DSLs used by our tool. However, we still miss one important component of our modeling facilities, viz. the 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 as it results from our security risk analysis into a DSL-based risk profile $\mathcal{RP}$.

3.6.3 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 translating our DSL-based security problem $\mathcal{SP}$ into a declarative representation. Our SDL translator implements a visitor pattern, i.e., it traverses the input model by visiting each node and subsequently, for each node, calls some distinct template which expands in the resulting model.

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 with in a non logic programming context. 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 [63]. Scala's parser combinators offer a declarative syntax for easy development of language parsers. 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 some input model, our parser, for each visited node then invokes the corresponding expansion template which, in the end, manifests itself in the output model. We already implemented a model-2-text generator using the same techniques (e.g., template- and visitor-based) which is why we skip any further discussion of our model translators [64].

4 Experiments

We evaluated the model-based tool implementation of our method for testing web applications in PHP/SQL by Damn Vulnerable Web Application (DVWA), version 1.8 [8]. DVWA is a PHP/MySQL web application that is damn vulnerable. Its main goals are to be an aid for security 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 class room environment [8]. 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 [65].

DVWA is designed to implement a number of vulnerabilities w.r.t. OWASP's 2013 top ten list [2]. 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 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, for instance, eBanking, eHealth, 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 [2]. 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 (Sect. 3.2), and (ii) this would go beyond the scope of this article. The interested reader is advised to study OWASP's 2013 top ten list [2] 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 Experimental setup

For our experiments, 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. Figure 17 shows the setup we used for our experiments. A client machine that runs the testing tool sends request (as part of testing) to a web server machine that is running DVWA. Upon successful completion of a request, the result is sent back to the client machine.

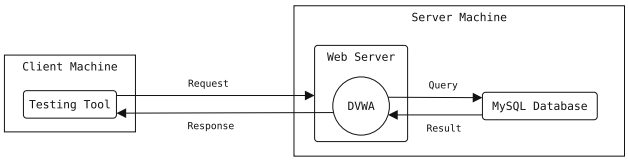


Fig. 17 Lab setup for our experiments. A client machine is running our testing tool. During testing the client sends requests to a web server that are forwarded to DVWA, which in turn creates the response. A MySQL database is running alongside the web server on the server machine

4.2 Collection and analysis of results

The results as presented and discussed in Sect. 5 were retrieved by executing test cases against DVWA and collecting the outcomes, i.e., responses returned by DVWA (which is done by our tool as it generates the test log). We then compared these results to DVWA. On the grounds of knowing whether a vulnerability really is present in DVWA or not, we derived the interpretations of the retrieved results as presented in Sect. 5.

A potential threat to the validity of our results and their interpretation is that we have performed all experiments under laboratory conditions and further, by using a deliberately vulnerable application, i.e., no field application. We, however, argue that performing experiments under laboratory conditions allows to thoroughly measure its effectiveness as we do not have to consider external noise, e.g., disturbing network traffic and the like. Further, the application that we used for our experiments, i.e., DVWA, reflects common real-life vulnerabilities as its contained vulnerabilities are based on OWASP 2013 top ten list.

5 Results and discussion

Following, we show and discuss the results of two experiments, in which we used our tool to detect vulnerabilities in DVWA. In the first experiment DVWA's security level is set low, whereas in the second experiment, the security level is set to medium. At first, we also tried to detect vulnerabilities with the security level set to high; however, we soon realized that this exceeded our tool's capabilities in detecting vulnerabilities, simply due to that at this point, attacks are too complex to be done in an automated manner (e.g., iterated fine-grained evaluation of the application's responses is necessary to finally come up with a working injection string). Figure 18 shows the reduced security problem $\mathcal{SP}$ for DVWA. We deliberately chose to reduce the initial security problem $\mathcal{SP}$ as returned from our web spider as our tool would fail to detect vulnerabilities for specific operations (in total 6), e.g., `/vulnerabilities/brute/#` which implements a brute force vulnerability.

```
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")
            ))
        )
      )
    )
  )
}
```

Fig. 18 Reduced security problem $\mathcal{SP}$ as established by our web spider for DVWA (clearly, the web spider did not reduce the model but we did manually)

5.1 Testing with low security level

For our first experiment, 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 $\mathcal{EDB}$, viz. SQLI and XSS vulnerabilities.

Table 2 shows the results of our first case study. A total number of 250 test cases were executed against the four operations from the security problem $\mathcal{SP}$ for DVWA (Fig. 18). The total execution time of all test cases (including test evaluation) was about 4 min, which we found quiet acceptable.

Table 2 Results from test execution for our first experiment with DVWA

Risks	Target	Executions	Verdict		
			PASS	FAIL	INCONCLUSIVE
Risk 1—SQLI (signature evasion)	/vulnerabilities/sqli/# (id)	10	9	1	0
Risk 2—SQLI (blind)	/vulnerabilities/sqli/# (id)	10	10	0	0
Risk 3—SQLI (type handling)	/vulnerabilities/sqli/# (id)	10	10	0	0
Risk 4—XSS (reflected)	/vulnerabilities/sqli/# (id)	10	0	10	0
Risk 5—XSS (stored)	/vulnerabilities/sqli/# (id)	10	0	4	6
Risk 6—SQLI (signature evasion)	/vulnerabilities/sqli_blind/# (id)	10	10	0	0
Risk 7—SQLI (blind)	/vulnerabilities/sqli_blind/# (id)	10	10	0	0
Risk 8—SQLI (type handling)	/vulnerabilities/sqli_blind/# (id)	10	10	0	0
Risk 9—XSS (reflected)	/vulnerabilities/sqli_blind/# (id)	10	0	10	0
Risk 10—XSS (stored)	/vulnerabilities/sqli_blind/# (id)	10	0	0	10
Risk 11—SQLI (signature evasion)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 12—SQLI (blind)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 13—SQLI (type handling)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 14—XSS (reflected)	/vulnerabilities/xss_r/# (name)	10	10	0	0
Risk 15—XSS (stored)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 16—SQLI (signature evasion)	/vulnerabilities/xss_s/# (txtName)	10	0	10	0
Risk 17—SQLI (blind)	/vulnerabilities/xss_s/# (txtName)	10	0	10	0
Risk 18—SQLI (type handling)	/vulnerabilities/xss_s/# (txtName)	10	0	10	0
Risk 19—SQLI (signature evasion)	/vulnerabilities/xss_s/# (mtxMessage)	10	0	10	0
Risk 20—SQLI (blind)	/vulnerabilities/xss_s/# (mtxMessage)	10	0	10	0
Risk 21—SQLI (blind)	/vulnerabilities/xss_s/# (mtxMessage)	10	0	10	0
Risk 22—XSS (reflected)	/vulnerabilities/xss_s/# (txtName)	10	10	0	0
Risk 23—XSS (stored)	/vulnerabilities/xss_s/# (txtName)	10	10	0	0
Risk 24—XSS (reflected)	/vulnerabilities/xss_s/# (mtxMessage)	10	10	0	0
Risk 25—XSS (stored)	/vulnerabilities/xss_s/# (mtxMessage)	10	10	0	0
		250	109	125	16

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. See Table 4 for false positives and negatives

At first sight, the results shown in Table 2 may be a little bit dizzying, as the verdict FAIL has been deduced for 125 test runs. Yet, this is desirable. Remember that our method is designed for non-functional security testing. Contrary 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 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 experiments.

Table 2 shows 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, it 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/#; obviously this operation does not implement an SQLI vulnerability). 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.

5.2 Testing with medium security level

For our second experiment, 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.

Table 3 shows the results for our second experiment with DVWA's security level set to medium. Again, a total number of 250 test cases were executed against the four operations from the security problem $\mathcal{SP}$ for DVWA (Fig. 18). Again executing all test cases (including test evaluation) only lasted for about 4 min. As expected, the results for our second experiment as shown in Table 3 differ from those of the first experiment, as shown in Table 2.

Obviously, our tool had a slightly harder time to detect vulnerabilities as compared to our first experiment 5.1. This is by virtue of security mechanism active in DVWA for the current experiment. However, the results for our second experiment again show the effectiveness of our method and its tool implementation for non-functional security testing by logic programming. Our method again was able to verify all existing vulnerabilities, yet, this time with a slightly lower success rate due to, as just mentioned, active security mechanisms 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.

5.3 False positives and false negatives

During our first experiment, 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 because our test evaluation 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 experiment, 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 remaining 4 false negatives occurred when executing Risk 14, i.e., a reflected XSS attack against operation

Table 3 Results from test execution for our second experiment with DVWA

Risks	Target	Executions	Verdict		
			PASS	FAIL	INCONCLUSIVE
Risk 1—SQLI (signature evasion)	/vulnerabilities/sqli/# (id)	10	9	1	0
Risk 2—SQLI (blind)	/vulnerabilities/sqli/# (id)	10	2	8	0
Risk 3—SQLI (type handling)	/vulnerabilities/sqli/# (id)	10	10	0	0
Risk 4—XSS (reflected)	/vulnerabilities/sqli/# (id)	10	0	10	0
Risk 5—XSS (stored)	/vulnerabilities/sqli/# (id)	10	0	10	0
Risk 6—SQLI (signature evasion)	/vulnerabilities/sqli_blind/# (id)	10	10	0	0
Risk 7—SQLI (blind)	/vulnerabilities/sqli_blind/# (id)	10	10	0	0
Risk 8—SQLI (type handling)	/vulnerabilities/sqli_blind/# (id)	10	10	0	0
Risk 9—XSS (reflected)	/vulnerabilities/sqli_blind/# (id)	10	0	10	0
Risk 10—XSS (stored)	/vulnerabilities/sqli_blind/# (id)	10	0	0	10
Risk 11—SQLI (signature evasion)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 12—SQLI (blind)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 13—SQLI (type handling)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 14—XSS (reflected)	/vulnerabilities/xss_r/# (name)	10	6	4	0
Risk 15—XSS (stored)	/vulnerabilities/xss_r/# (name)	10	0	10	0
Risk 16—SQLI (signature evasion)	/vulnerabilities/xss_s/# (txtName)	10	0	10	0
Risk 17—SQLI (blind)	/vulnerabilities/xss_s/# (txtName)	10	0	10	0
Risk 18—SQLI (type handling)	/vulnerabilities/xss_s/# (txtName)	10	0	10	0
Risk 19—SQLI (signature evasion)	/vulnerabilities/xss_s/# (mtxMessage)	10	0	10	0
Risk 20—SQLI (blind)	/vulnerabilities/xss_s/# (mtxMessage)	10	0	10	0
Risk 21—SQLI (blind)	/vulnerabilities/xss_s/# (mtxMessage)	10	0	10	0
Risk 22—XSS (reflected)	/vulnerabilities/xss_s/# (txtName)	10	10	0	0
Risk 23—XSS (stored)	/vulnerabilities/xss_s/# (txtName)	10	10	0	0
Risk 24—XSS (reflected)	/vulnerabilities/xss_s/# (mtxMessage)	10	10	0	0
Risk 25—XSS (stored)	/vulnerabilities/xss_s/# (mtxMessage)	10	10	0	0
		250	97	143	10

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. See Table 4 for false positives and negatives

/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/#.

In case of the 10 false positives that occurred during execution of our second experiment, 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 reasoning as in case of our first experiment applies.

Table 4 Summary of false positives and false negatives for both our experiments with DVWA

Case study Nr	False positives	False negatives
1	16 (6.4%)	0 (0%)
2	10 (4%)	12 (4.8%)

The values in the brackets show the rate of false positives and negatives w.r.t. the number of executed test cases

Table 4 summarizes the number of false positives and false negatives that occurred during the two experiments with DVWA.

Table 5 Timing-related execution metrics

	Low security	Medium security
Model generation	55.7	62.3
Test generation	45.2	46.0
Test execution and evaluation	231.5	235.2
Total	329.7	343.5

All durations are in seconds

5.4 Execution metrics

Table 5 summarizes the duration of execution of the various steps as conducted during our experiments, viz. model generation, test case generation as well as test execution and evaluation. As of online evaluation of test cases, i.e., immediately after being executed, we unfortunately fail to provide an exact duration for test evaluation. In total, each of the experiments lasted around 5–6 min. We thus conjecture that the proposed method is not only effective in terms of error detection but also efficient in terms of the necessary time to test a web application. For sure, given the amount of user operations provided by an application, execution times increase, yet we claim that our framework scales well as of its modular and inference-based implementation. Observe that we, however, did not consider network speeds and latencies in our experiments as they provide no additional source of explanation in terms of execution times related to the testing framework itself.

5.5 Threats to validity

Although we did not test our method and its tool implementation with a real-world application but instead with a playground for security experts, viz. DVWA, we still claim that our method is applicable for testing real-world, productive systems and that the achieved results are representative. This is by virtue of DVWA's design and implementation that starkly resemble common design and implementation errors occurring in real-world applications that ultimately yield security vulnerabilities. Our experiments have shown that our tool implementation is capable of detecting vulnerabilities under real-world conditions.

During the course of testing, it may occur that our tool misses certain vulnerabilities. This is, apart from testing against stored XSS vulnerabilities as discussed in the previous sections, by virtue of that the VKB just does not know about the necessary attack patterns to detect certain vulnerabilities. Therefore, our tool strongly relies on the VKB and its codified contents and thus ultimately on a responsible security expert. We argue however that such a single point of human failure generally cannot be eradicated, at least for now. As traditionally done in such a case of a single point of human failure, during our experiments, we applied the dual control principle for the created VKB.

6 Conclusions

In our article, we have introduced both a model-based tool implementation for non-functional security testing of web applications in PHP/SQL and its underlying method in Sect. 3. Our method is kept generic in a sense that indeed it requires a custom $\mathcal{EDB}$ (i.e., security vulnerability knowledge of attacks and their enabling vulnerabilities) and DCG(s) for dynamic test data generation, however, its remaining components, i.e., the security risk analysis and its therein contained generation of abstract test cases are generic.

The key idea of this article 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 of 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 $\mathcal{RP}$.

We conclude that our work has several implications as listed below:

- Non-functional 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.
– Effective testing tools for making web applications more secure are feasible.

In future, we especially plan to address autonomous learning of the $\mathcal{EDB}$, improving the $\mathcal{EDB}$ by learning from test feedback, automated generation of DCGs by mutation as well as regression testing aspects, which have already been considered for functional security testing [66]. Furthermore, we plan to perform empirical studies to show the effectiveness and efficiency of the approach in an industrial context.

Acknowledgements Open access funding provided by University of Innsbruck and Medical University of Innsbruck. This research was partially supported by the research Project MOBSTECO (FWF P 26194-N15).

References

1. Cenzic: Application Security Trends Report 2013 (March 2014)
2. OWASP, T.: Top 10–2013–the ten most critical web application security risks. The Open Web Application Security Project (2013)
3. Schieferdecker, I.: Model-based testing. IEEE Softw. **29**(1), 14–18 (2012)
4. Schieferdecker, I., Grossmann, J., Schneider, M.: Model-based security testing. In: Proceedings 7th Workshop on Model-Based Testing (2012)
5. Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschner, A.: Model-based security testing: a taxonomy and systematic classification. Softw. Test. Verif. Reliab. **26**(2), 119–148 (2016)
6. Legeard, B., Utting, M.: Practical Model-based Testing: A Tools Approach. Morgan Kaufmann, Burlington (2010)
7. McGraw, G., Potter, B.: Software security testing. IEEE Secur. Priv. **2**(5), 81–85 (2004)
8. RandomStorm: Damn Vulnerable Web Application (DVWA) (March 2014)
9. Zech, P., Felderer, M., Breu, R.: Towards risk-driven security testing of service centric systems. In: QSIC, pp. 140–143 (2012)
10. Zech, P., Felderer, M., Farwick, M., Breu, R.: A concept for language-oriented security testing. In: 2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C), pp. 53–62. IEEE (2013)
11. Zech, P., Felderer, M., Katt, B., Breu, R.: Security test generation by answer set programming. In: 2014 IEEE 8th International Conference on Software Security and Reliability-Companion (SERE), IEEE (2014)
12. Zech, P., Felderer, M., Breu, R.: Cloud risk analysis by textual models. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing, p. 5. ACM (2012)
13. Zech, P., Felderer, M., Breu, R.: Security risk analysis by logic programming. In: 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK), Springer (2014)
14. Gotlieb, A., Botella, B., Rueher, M.: Automatic Test Data Generation using Constraint Solving Techniques. In: ACM SIGSOFT Software Engineering Notes. Vol. 23, pp. 53–62. ACM (1998)
15. Meudec, C.: ATGen: automatic test data generation using constraint logic programming and symbolic execution. Softw. Test. Verif. Reliab. **11**(2), 81–96 (2001)
16. Jasper, R., Brennan, M., Williamson, K., Currier, B., Zimmerman, D.: test data generation and feasible path analysis. In: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, pp. 95–107. ACM (1994)
17. Vemuri, R., Kalyanaraman, R.: Generation of design verification tests from behavioral VHDLprograms using path enumeration and constraint programming. IEEE Trans. Very Large Scale Integr. Syst. **3**(2), 201–214 (1995)
18. Denney, R.: Test-case generation from prolog-based specifications. Softw. IEEE **8**(2), 49–57 (1991)
19. Bieker, U., Marwedel, P.: Retargetable self-test program generation using constraint logic programming. In: 32nd Conference on Design Automation, DAC'95, pp. 605–611. IEEE (1995)
20. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in CLP. Theory Pract. Logic Program. **10**(4–6), 659–674 (2010)
21. Lötzbeyer, H., Pretschner, A.: Testing concurrent reactive systems with constraint logic programming. In: 2nd Workshop on Rule-Based Constraint Reasoning and Programming, Singapore (2000)
22. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Applying constraint logic programming to SQL test case generation. In: Functional and Logic Programming, pp. 191–206. Springer (2010)
23. Gorlick, M.M., Kesselman, C.F., Marotta, D.A., Stott Parker, D.: Mockingbird: a logical methodology for testing. J. Logic Program. **8**(1), 95–119 (1990)
24. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Chapter one-security testing: a survey. Adv. Comput. **101**, 1–51 (2016)
25. Michael, C., Radosevich, W.: Risk-based and functional security testing. Build security In (2005)
26. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Secur. Priv. **3**(1), 84–87 (2005)
27. Bishop, M.: About penetration testing. IEEE Secur. Priv. **5**(6), 84–87 (2007)
28. Miller, B., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
29. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artec House, Norwood (2008)
30. Amland, S.: Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study. J. Sys. Softw. **53**(3), 287–295 (2000)
31. Wysopal, C., Nelson, L., Dustin, E., Dai Zovi, D.: The Art of Software Security Testing: Identifying Software Security Flaws. Addison-Wesley Professional, Boston (2006)
32. Felderer, M., Schieferdecker, I.: A taxonomy of risk-based testing. Int. J. Softw. Tools Technol. Transf. **16**(5), 559–568 (2014)
33. Stallbaum, H., Metzger, A., Pohl, K.: An automated technique for risk-based test case generation and prioritization. In: Proceedings of the 3rd International Workshop on Automation of Software Test, pp. 67–70. ACM (2008)
34. Blackburn, M., Busser, R., Nauman, A., Chandramouli, R.: Model-based approach to security test automation. In: Proceeding of Quality Week 2001 (2001)
35. Mouelhi, T., Fleurey, F., Baudry, B., Le Traon, Y.: A model-based framework for security policy specification, deployment and testing. In: Model Driven Engineering Languages and Systems, pp. 537–552. Springer (2008)
36. Jürjens, J., Wimmel, G.: Specification-based testing of Firewalls. In: Perspectives of System Informatics, pp. 308–316. Springer (2001)

37. Wimmel, G., Jürjens, J.: Specification-based test generation for security-critical systems using mutations. In: Formal Methods and Software Engineering, pp. 471–482. Springer (2002)
38. Jürjens, J.: UMLSec: Extending UML for secure systems development. In: UML 2002—The Unified Modeling Language, pp. 412–425. Springer (2002)
39. Jürjens, J.: Model-based security testing using UMLSec: a case study. Electron. Notes Theor. Comput. Sci. **220**(1), 93–104 (2008)
40. Wang, L., Wong, E., Xu, D.: A threat model-driven approach for security testing. In: Third International Workshop on Software Engineering for Secure Systems, SESS'07: ICSE Workshops 2007, pp. 10–10. IEEE (2007)
41. Marback, A., Do, H., He, K., Kondamarri, S., Xu, D.: Security test generation using threat trees. In: AST'09. ICSE Workshop on Automation of Software Test, 2009, pp. 62–69. IEEE (2009)
42. Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., Xu, W.: Automated security test generation with formal threat models. IEEE Trans. Dependable and Secur. Comput. **9**(4), 526–540 (2012)
43. Xu, D., Nygard, K.E.: Threat-driven modeling and verification of secure software using aspect-oriented petri nets. IEEE Trans. Softw. Eng. **32**(4), 265–278 (2006)
44. Avancini, A., Ceccato, M.: Towards security testing with taint analysis and genetic algorithms. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems. SESS '10, pp. 65–71. ACM (2010)
45. Büchler, M., Oudinet, J., Pretschner, A.: Semi-Automatic Security Testing of Web Applications from a Secure Model. In: 2012 IEEE Sixth International Conference on Software Security and Reliability (SERE), pp. 253–262 (2012)
46. Büchler, M., Oudinet, J., Pretschner, A.: SPaCiTE—web application testing engine. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 858–859 (2012)
47. Tappenden, A., Beatty, P., Miller, J., Geras, A., Smith, M.: agile security testing of web-based systems via HTTPUnit. In: Proceedings of Agile Conference, 2005, pp. 29–38 (2005)
48. Offutt, J., Wu, Y., Du, X., Huang, H.: Bypass testing of web applications. In: 15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004, pp. 187–197 (2004)
49. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 249–260. ACM (2008)
50. Xiong, P., Peyton, L.: A model-driven penetration test framework for web applications. In: 2010 Eighth Annual International Conference on Privacy Security and Trust (PST), pp. 173–180 (2010)
51. Chen, S., Miao, H., Qian, Z.: Automatic generating test cases for testing web applications. In: International Conference on Computational Intelligence and Security Workshops, 2007. CISW 2007, pp. 881–885 (2007)
52. Song, B., Gong, S., Chen, S.: Model composition and generating tests for web applications. In: 2011 Seventh International Conference on Computational Intelligence and Security (CIS), pp. 568–572 (2011)
53. Kiezun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: IEEE 31st International Conference on Software Engineering, 2009, ICSE 2009, pp. 199–209. IEEE (2009)
54. Diaz, D.: GNU Prolog. (1999). http://gprolog.univ-paris1.fr/
55. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, pp. 73–87. ACM (2000)
56. Felderer, M., Zech, P., Fiedler, F., Breu, R.: A tool-based methodology for system testing of service-oriented systems. In: 2010 Second International Conference on Advances in System Testing and Validation Lifecycle (VALID), pp. 108–113. IEEE (2010)
57. Ganjisaffar, Y.: Crawler4J (March 2014)
58. Breu, M., Breu, R., Low, S.: Living on the move: towards an architecture for a living models infrastructure. In: 2010 Fifth International Conference on Software Engineering Advances (ICSEA), pp. 290–295. IEEE (2010)
59. Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M., Innerhofer-Oberperfler, F.: Living models-ten principles for change-driven software engineering. Int. J. Softw. Inf. **5**(1–2), 267–290 (2011)
60. Zech, P., Felderer, M., Kalb, P., Breu, R.: A generic platform for model-based regression testing. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 112–126. Springer (2012)
61. Zech, P., Kalb, P., Felderer, M., Atkinson, C., Breu, R.: Model-based regression testing by OCL. Int. J. Softw. Tools Technol. Transf. **19**(1), 115–131 (2017)
62. Hartman, A., Katara, M., Olvovsky, S.: Choosing a test modeling language: a survey. In: Hardware and Software, Verification and Testing, pp. 204–218. Springer (2007)
63. Moors, A., Piessens, F., Odersky, M.: Parser combinators in scala. CW Reports (2008)
64. Felderer, M., Fiedler, F., Zech, P., Breu, R.: Flexible test code generation for service oriented systems. In: 9th International Conference on Quality Software (QSIC '09). IEEE (2009)
65. RandomStorm: RandomStorm (March 2014)
66. Felderer, M., Agreiter, B., Breu, R.: Evolution of security requirements tests for service–centric systems. In: International Symposium on Engineering Secure Software and Systems, pp. 181–194. Springer (2011)