Software Quality Journal (2020) 28:307–334
https://doi.org/10.1007/s11219-019-09469-y

Planning-based security testing of web applications with attack grammars

Josip Bozic[1] · Franz Wotawa[1]

Published online: 9 March 2020

Abstract
Web applications are deployed on machines around the globe and offer almost universal accessibility. These applications assure functional interconnectivity between different components on a 24/7 basis. One of the most important requirements is data confidentiality and secure authentication. However, implementation flaws and unfulfilled requirements often result in security leaks that malicious users eventually exploited. In this context, the application of different testing methods is of utmost importance in order to detect software defects during development and to prevent unauthorized access in advance. In this paper, we contribute to test automation for web applications. In particular, we focus on using planning for testing where we introduce underlying models covering attacks and their use in testing of web applications. The planning model offers a high degree of extendibility and configurability and as well overcomes limits of traditional graphical representations. New testing possibilities emerge that eventually lead to better vulnerability detection, therefore ensuring more secure web services and applications.

Keywords Planning · Security testing · Model-based testing · Web applications

1 Introduction

Testing for vulnerabilities in software is without any doubt an important task during software development. Despite this fact, it is interesting to note that over the past few years the same vulnerabilities remain on the top 10 list of most common security leaks in web applications (OWASP 2018). The reasons behind may be human implementation errors or insufficient security mechanisms on the side of the application that are caused by lack of security knowledge but also insufficient tool support.

SQL injections (SQLI) and cross-site scripting (XSS) are among the most common attacks in the domain of web applications, despite increased awareness of their potential

✉ Josip Bozic
jbozic@ist.tugraz.at

Franz Wotawa
wotawa@ist.tugraz.at

[1] Institute of Software Technology, Graz University of Technology, A-8010, Graz, Austria

damage. In both attacks, malicious user inputs eventually get bypassed by the applications' input filters, usually resulting in theft of confidential data or causing other harm to the user or the system itself. As a consequence, there are corresponding costs needed for repairing the software but also loss of trust in the system, both of which can be critical for service providers. Therefore, means for improving security are of great importance and have to be implemented.

In this paper, we contribute to this challenge and provide an approach that is based on automated planning, which originates from classical artificial intelligence (AI). The planning was initially intended to be used in robotics and intelligent agents (Russell and Norvig 1995). However, there has been research using planning in the context of security testing, e.g. see Durkota and Lisy (2014) and Backes et al. (2017). In these papers, sequences of actions are generated in order to test certain types of systems for intrinsic leaks. In general, security testing of a system is done by checking whether a running system can be tested or hacked by malformed inputs or whether it can be broken down completely.

Other techniques include combinatorial testing (CT). One of the main motivations behind CT is test set reduction (Kuhn et al. 2015; Simos et al. 2016; Raunak et al. 2017), where a security leak is eventually detected using a reduced number of tests. This implies a reduction of testing time but still ensures the ability to identify causes of vulnerabilities. Some other techniques rely on models, either of the application (Krämer and Legeard 2016; Felderer et al. 2016) or of the attacks themselves (Duchene et al. 2014; Shameli-Sendi et al. 2017). A system under test (SUT) is checked regarding whether it behaves in line with its specification using a graphical representation that corresponds to the SUT's expected behavior.

In this paper, we contribute to the application of AI for security testing of web applications, which makes use of automated planning for obtaining test suites to test common XSS and SQLI vulnerabilities. The proposed paper represents an extension to our previously presented approach in Bozic and Wotawa (2018). In addition to test case generation, we discuss a test execution framework encompassing a crawler, which helps in exhaustive testing of the SUT. The idea behind our approach is to automatically obtain tests that mimic unexpected user sequences and input values for detecting vulnerabilities. We elaborate the approach in detail using a running example and also discuss first empirical results based on using our approach for testing example applications.

We organize the paper as follows: Section 2 explains the intention of the paper, whereas in Section 3, we introduce planning for security testing and outline the basic concepts of automated planning. Afterwards, we discuss the functionality of the automated test execution framework in Section 5. In Section 6, we illustrate our approach using an example followed by an empirical evaluation. In Section 8, we discuss related work and finally conclude the paper in Section 9.

2 Approach overview

The main goal of this paper is to provide a testing approach that uses planning on multiple levels. In the proposed approach, we split test cases into two distinct types: first, test cases that instruct the testing system what to do, which serve as testing guidelines; second, test cases that contain concrete values that will be processed by a SUT. The former are labeled abstract test cases, whereas the latter are described as concrete tests. Thus, the test generation process will be split into two layers with distinguished mechanisms. The results from both implementations will consist of two distinct input sets, which will be combined before test execution starts.

One of the main proposals in this paper is planning models. Models have been already used in testing for multiple purposes (Krämer and Legeard 2016). In most cases, these models represent graphical depictions that come in two flavors: In case of white-box testing, the model depicts the internal structure or expected behavior of a SUT. Under such circumstances, test cases are generated from the model, thereby comparing obtained results with expected ones. On the other hand, black-box testing approaches lack any information about the SUT. In these cases, a model can be inferred from the system's behavior or, from the offensive point of view, a model depicts attacks against such system. Our approach follows a black-box definition where models are used for generation of attacks against a system. Planning models encompass their own unique set of definition languages. As will be explained throughout the paper, advantages of such models can be summarized as follows:

- *Exploration*: Detection of potential paths and unintended behavior in a system. This feature plays an important role in negative testing.
- *Model reduction*: Avoidance of model explosion in case of complex systems
- *Configurability*: Conditions can be set on individual definitions of the model that act as constraints and guide the planning process. Such mechanism can produce more meaningful inputs for a certain domain. This represents a stark contrast to concepts that rely on (semi-) randomness (Stephens et al. 2016) or overly permutations (Raunak et al. 2017).
- *Extendibility*: Manipulations of the model can be added easily, eventually resulting in wide-ranging consequences.

We use such planning representations in both layers of the test generation process. In both cases, the model is adapted individually for its specific purpose. Although planning represents the bulk of this paper, it represents one part in an overall testing framework. This implementation connects the individual parts and executes the testing process. In general, the overall structure of the approach can be split into multiple challenges:

1. *Generation of abstract test cases*: Planning model as attack model
2. *Generation of concrete test cases*: Planning gives guidelines for generation of concrete values.
3. *Test execution*: Combine and execute the results from (1) and (2).

Each task will be addressed separately in the following sections. However, first we will give a short introduction of the basic concepts of AI planning.

3 Planning for testing

Automated planning has been already used in testing in general and security testing in particular. We already presented an adaptation of planning for security testing of web applications (see Bozic and Wotawa 2014, 2018) that we further extended for testing of the transport layer security (TLS) protocol in Simos et al. (2018). However, in this paper, we present an improved approach of Bozic and Wotawa (2018) in terms of test case generation (number and variety of plans) and execution. In previous papers, we made use of planners that returned only one (optimal) solution for a certain security testing problem, which limits the potential for detecting failures and vulnerability. Therefore, we extended our approach and use a different planning system where the plan generation process can be guided by the tester. The planning system takes a planning problem as input and thereby eventually returns a set of solution in the form of plans, where a plan itself comprises a sequence of actions that lead from an initial to a final state. The initial state is usually an idle state where the execution of the SUT starts. The final state is a state when the execution terminates, returning either a passing or a failing test verdict.

In the following, we first formally define the planning problem and afterwards discuss modeling for security testing. The used underlying definitions for planning originate from Fikes and Nilson (1971). In our previous work (Bozic and Wotawa 2014), we already introduced the adaptations necessary to deal with security testing. To be self-contained, we briefly discuss the underlying definitions starting with the planning problem, which is defined as follows:

Definition 1 The tuple (P, I, G, A) states a planning problem, where P defines the set of predicates, I and G are the initial and goal states, and A denotes the set of actions. A predicate $p \in P$ represents a first-order logic formula and is used by $a \in A$. Every state is specified by predicates that are true in this state. The planning problem is to obtain a sequence of actions from A leading from an initial state to a goal state.

In case that the preconditions of an action a are true in a certain state S, then it will be added to the plan. In this case, a will act as a transition to the new state S', thus $S \xrightarrow{a} S'$. The predicates from its postconditions will be valid in S' and act as the new state's precondition.

Definition 2 A solution for a planning problem (P, I, G, A) is a plan that comprises a set of actions so that $I \xrightarrow{a_1} S_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} G$.

The definition of predicates and their corresponding conditions will guide the planning process. Sometimes more solutions do exist for a planning problem. During its search, the planner decides what path to choose according to the underlying planning algorithm. On the other hand, if no solution can be produced for a problem, then the planner terminates further search.

3.1 Planning model

In order to specify a planning problem, we create a planning model. In contrast to other graphical representations, e.g. UML state machines, we describe our model in the Planning Domain Definition Language (PDDL) (McDermott et al. 1998).

The reasons for this decision are the potential extendibility and configurability of such planning models. For example, actions can be defined without preconditions, which makes the overall model more comprehensive. Thus, model explosion can be avoided. New specifications can be added easily or several models can be combined. Consequently, every change in the model will result in the generation of diverse plans. Here we are able to obtain plans with greater variety that eventually improve fault detection. In addition to that, a planning model might cause execution traces that are usually not defined in a model of the specification of a SUT.

Regarding PDDL, two descriptions have to be specified, namely:

1. Domain: Comprehends the definitions that are present in every problem
2. Problem: Comprehends the definitions that are valid only in a specific problem

In general, PDDL uses a type-object hierarchy and the definition of first-order logic predicates. The objects are used in both specifications and are put as parts of predicates. Additionally, actions are specified by parameters, pre- and postconditions. The conditions are defined by predicates and describe when an action is selected and what effects it will cause when executed.

The problem specification usually defines application specific parameters. However, our approach keeps both specifications as minimal as possible. The problem defines objects and the initial and final states. The corresponding problem specification in PDDL looks as follows.

```
(define (problem mbtp)
 (:domain mbtd)
 (:objects x)
 (:init (inInitial x))
 (:goal (and (inFinished x))))
```

Listing 1 Problem description in PDDL

The initial state represents the starting point of the test execution. This is the point before any action has been carried out. On the other hand, the final state is reached when a test case, i.e. a plan, has been carried out. In reality, the initial state is the point before the execution framework even accesses a specific URL address of a SUT. The final state is eventually reached after an attack and all its actions have been carried out and a test verdict is given. Our specification relies on only one object, `x`, that takes the role inside a specific state during execution.

In addition to that, PDDL's domain encompasses, among others, the definitions of predicates and actions. Our definition for testing of web applications is given below.

The specification defines all possible states that are encountered during test execution as predicates. These are used for the definition of actions, together with the already mentioned object `x` as a parameter. The action's pre- and postconditions can be specified by zero, one or more predicates that guide the plan generation. Actions can be picked multiple times as part of one plan if the conditions are met. Actually, they are defined in a way that ensures that many plans can be generated by the planner, for example by omitting or using multiple conditions. On the concrete level, all actions will be carried out over a HTTP infrastructure, as will be described in Section 5.

The planning specifications are saved in the form of PDDL files and submitted to a planning system for constructing a plan. Then, plans are generated by the planner according to a planning algorithm (see Section 3.2). An example of a generated plan with seven actions is given below.

```
[GetSite(x), AttackSQLGet(x), GetSite(x), AttackSQLPost(x),
 AttackXSSPost(x), GetSite(x), AttackXSSGet(x)]
```

As can be seen in this example, the plan comprehends actions as specified in the domain definition. A generated plan represents an abstract test case that guides the test execution. In fact, the planning model is a model of potential attacks. To be concise, it comprehends all actions that can be carried out under certain circumstances. The conditions guide the plan generation, so the planner will search for all possible sequences that satisfy the criteria. From the practical point of view, this depicts all possible malicious actions that a user might take for obfuscating a system. It focuses on client-side messages, so it omits

```
(define (domain mbtd)
 (:requirements
    :strips :typing :negative-preconditions)

 (:predicates
    (ininitial ?x)
    (inGotSite ?x)
    (inAttackedSQLI ?x)
    (inAttackedXSS ?x)
    (inFinished ?x))

 (:action GetSite
    :parameters(?x)
    :precondition (and )
    :effect (and (inGotSite ?x)))

 (:action AttackXSSGet
    :parameters(?x)
    :precondition (and (inGotSite ?x) (inAttackedSQL ?x)))
    :effect (and (inAttackedXSS ?x) (inFinished ?x)))

 (:action AttackXSSPost
    :parameters(?x)
    :precondition (and (inGotSite ?x) (inAttackedSQL ?x))
    :effect (and (inAttackedXSS ?x) (inFinished ?x)))

 (:action AttackSQLGet
    :parameters(?x)
    :precondition (and (inGotSite ?x))
    :effect (and (inAttackedSQL ?x) (inFinished ?x)))

 (:action AttackSQLPost
    :parameters(?x)
    :precondition (and (inGotSite ?x) (inAttackedXSS ?x))
    :effect (and (inAttackedSQL ?x) (inFinished ?x)))
```

Listing 2 Domain description in PDDL

man-in-the-middle attacks where the attacker emulates the server in front of a victim. Hence, server-side processes will not be taken into consideration in the planning model. (However, the reactions from the server to the client's actions will be checked automatically by the framework during execution.) Also, the goal of this approach is to test an application, i.e. to prove a vulnerability that could be exploited in the aftermath. Figure 1 depicts the plan generation process from the model. All generated plans constitute an abstract test set.

The reason why we intend to generate plans of different shapes is to stress the SUT with unexpected actions. In the best case, a defect will be triggered due to a specific sequence of actions. The plan generation process is explained in the next section.

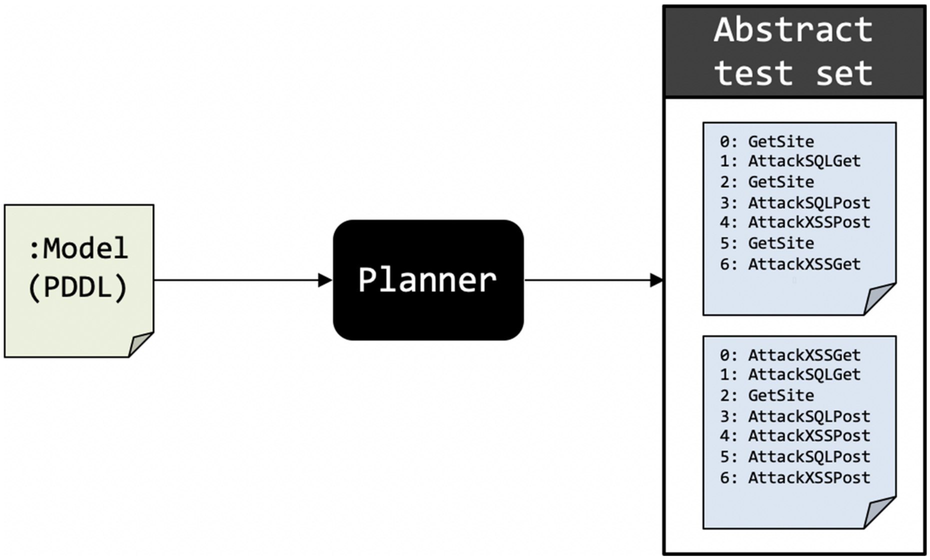

Fig. 1 Abstract test case generation from the planning model

3.2 Planning system

In previous work, we relied on Metric-FF (2016) for plan generation. Unfortunately, that planner usually returns only one plan for a planning problem. However, in the current approach, we generate a high number of plans. This number is multiplied with the number of concrete values so a much higher number of test cases is the result. For this purpose, we use a Java implementation of the Graphplan algorithm, JavaGP (2017).

The Graphplan algorithm was initially introduced in Blum and Furst (1995). Here a planning graph is constructed in the search for a solution. The algorithm starts in an initial state and extends the graph level as long as no plan is generated. First, the planner checks whether the goals of the effects in the last graph level are true in the conditions of the initial level by using backward chaining. If this is not the case, then new actions are selected and the graph is extended. Then the newly obtained goals act as the starting point for solution extraction. In this case, the depth of the planning graph is increased. In JavaGP, this parameter can be set manually in order to define or restrict the number of generated plans. The higher the value, the more plans will be generated. With increasing size of the planning graph, the number of necessary steps to obtain a plan increases as well.

Finally, if a path is found that reaches from the initial to the last graph level, then a plan has been found. The individual actions on that path will constitute the atomic parts of the plan.

In our case, the planning system reads the generated PDDL files and returns a set of plans that will be submitted to the test execution framework. A small excerpt of the test set is depicted in Table 1.

The main motivation for generating multiple plans is to cause unintended behavior, eventually confusing the SUT. Also, the planning algorithm generates plans with repetitive actions, which means that some actions are executed multiple times in one test case. The test suite encompasses plans with a broad diversity. As depicted in Table 1, the length of the

Table 1 Generated plans for XSS and SQLI

1	`GetSite(x), AttackSQLGet(x), AttackXSSGet(x), AttackSQLPost(x)`
2	`GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSGet(x), AttackSQLPost(x)`
3	`GetSite(x), GetSite(x), AttackSQLGet(x), AttackXSSGet(x), AttackSQLPost(x), GetSite(x), AttackXSSPost(x)`
4	`GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSGet(x), GetSite(x), AttackSQLPost(x)`
5	`GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSGet(x),GetSite(x), GetSite(x), AttackSQLPost(x), AttackXSSGet(x)`
6	`GetSite(x), GetSite(x), AttackSQLGet(x), AttackXSSGet(x), GetSite(x), GetSite(x), AttackSQLPost(x), AttackXSSPost(x)`
7	`GetSite(x), AttackSQLGet(x), AttackXSSPost(x), GetSite(x), AttackSQLPost(x)`
8	`GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSPost(x), AttackSQLPost(x), GetSite(x), AttackXSSPost(x)`
9	`GetSite(x), AttackSQLGet(x), GetSite(x), GetSite(x), AttackXSSPost(x), AttackSQLPost(x), GetSite(x), GetSite(x), AttackXSSGet(x), AttackSQLPost(x)`
10	`GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSPost(x),GetSite(x), AttackSQLPost(x), GetSite(x), AttackXSSGet(x), GetSite(x), AttackSQLPost(x)`

plans differs in number, whereas some bigger plans comprehend smaller ones. Also, we set the initial precondition to cause the action `GetSite` to be the first action in every plan.

3.3 Planning for web applications

The communication between a client and an application on a server proceeds over HTTP(S). Usually, the client accesses a web application via its URL address by sending a HTTP request. The server returns a corresponding HTTP response. Both message types encompass a standardized structure with unique parameters, whereas its values are generated when these messages are created. All message content, including harmful ones, between peers is transferred via HTTP messages. Vulnerabilities are triggered either due to security leaks on the side of the client or the sever. A detailed description of both attacks can be found in Fogie et al. (2007) and Clarke et al. (2012).

HTTP request messages encompass methods that are usually GET or POST. In the first case, a client requests data from a URL address, whereas data is submitted to the destination in the second case. However, both methods can be used to submit and receive data. An example of the request line of HTTP GET looks like:

```
GET /site/login_form.php?username=john&password=!js123
HTTP/1.1
```

On the other hand, the counterpart POST can have the following form:

```
POST /site/login_form.php HTTP/1.1
Host: w3company.com
username=john&password=!js123
```

Here both depictions describe an example of a login form with two parameters. However, in certain cases, it does make sense to send a GET message attempt where a POST is expected and vice versa. For this reason, we labeled actions to resemble such HTTP requests so that a request is sent with another method instead of the expected one. In such way, we are able to trigger unexpected executions by the tester. The corresponding parameters will be instantiated with malicious inputs and sent as part of a real HTTP message to the server.

In this section, the generation of abstract test cases is elaborated. However, attacks need to be instantiated with concrete values before being submitted. Therefore, our novel technique for generation of concrete test cases is introduced in the next section.

4 Planning-based attack concretization

Until now, we have described the use of a planner in order to generate sequences of actions. However, since PDDL lacks the support for concrete parameter values, a method for generating concrete test cases is needed.

We define malicious inputs as attack vectors. These are concrete values that an attacker could use in order to exploit an application. In our approach, they are meant for testing purposes and can be either XSS scripts or SQL commands. This means that two sets of concrete test cases will be used, one for every vulnerability. Both test sets are attached separately to the framework and accessed at runtime during execution.

For this reason, a method is introduced for the generation of attack vectors. It comprises two parts, namely the specification of attack grammars and the definition of attack vector models. By combining these definitions, a set of concrete values is automatically generated for both types of vulnerabilities, respectively.

4.1 Attack grammars

Formal grammars have been already used for security testing purposes (Su and Wassermann 2006; Godefroid et al. 2008). Such grammars consist of finite sets of terminal and nonterminal symbols that can be specified in the Backus-Naur form (BNF) structure (Backus 1959; Naur 1960). Actually, our definition of an attack grammar is motivated by the lack of a concrete structure for XSS and SQLI attack vectors. So, our assumption is, since no general specification for such structures exists, that the order and occurrence of specific elements, in our case nonterminals, is not strictly specified as well. This guarantees a certain degree of flexibility for the construction of concrete inputs.

In our approach, grammars are defined for both types of attacks with the goal to generate concrete attack vectors. For this reason, the symbols are defined in a way that they resemble building blocks for individual parts of an attack vector. For this reason, we label these specifications as attack grammars. An excerpt from our grammar for XSS in BNF form looks as follows.

```
<pre>::= \ | " | ' | > | ; | ! | -
<opening>::= < <html> <content> >
<html>::= SCRIPT | OBJECT | IFRAME | FORM | SRC | BODY |
    LINK | IMG
<content>::= <attribute> = <value>
<attribute>::= href | src | style | title | Content |
    disabled | onmouseover | onclick | onerror | onmousemove
<value>::= alert(document.cookie) | "alert(document.cookie)"
     | "javascript:alert(XSS)" | x | java\0script:alert(\"XSS
    \")>";' | JaVaScRiPt:alert('XSS') | "0;url=javascript:
    alert('XSS');" | "jav  ascript:alert(0);" | " &#14;
    javascript:alert(0);" | "http://xss.rocks/xss.js" | '
    vbscript:msgbox("XSS")' | "&{alert('XSS')}" | "<http://
    xss.rocks/xss.css>; REL=stylesheet" | "xss:expr/*XSS*/
    ession(alert('XSS'))" | 'no\xss:noxss("*//*"); | "text/
    javascript">alert('XSS'); | "xss:expression(alert('XSS'))
    " | "behavior: url(xss.htc);" | "0; URL=http://;URL=
    javascript:alert('XSS');" | "background-image: url(&#1;
    javascript:alert('XSS'))" | "width: expression(alert('XSS
    '));" | "http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6
    D" | "h tt p://6  6.000146.0x7.147/" | "javascript:
    document.location='http://www.google.com/'" | "http://www
    .google.com/ogle.com/"
<wobject>::= alert(0) | alert(XSS) | alert('XSS')
<closing>::= < / <html> >
<post>::= \ | " | ' | > | ; | ! | -
```

Listing 3 Attack grammar for XSS

As can be seen, this grammar comprehends various HTML elements, as well as attribute definitions and values. In order to construct the grammar, we relied on the HTML specification (2018), our own experience, and other sources (e.g. XSS 2018). The grammar comprises values and characters that are known to cause XSS issues.

Usually, in order to carry out a successful SQLI, only the `WHERE` clause of the SQL statement needs to be specified (Su and Wassermann 2006). This means that the `SELECT` statement from SQL commands does not need to be modeled. As was the case with XSS, the foundation for constructing the grammar specification for SQLI has been external sources (2018) as well as our previous work (Bozic and Wotawa 2012). The attack grammar for SQLI is depicted below.

```
<expression>::= <pre> <whitespace> OR <whitespace> <var>
                <whitespace> <symbol> <var2> <post>
                <whitespace>
<pre>::= ' | a' | 1
<outer>::= ' | a | %27 | 1 | '%a%'
<var>::= version | database() | user() | '' | 1
<symbol>::= != | = | LIKE
<whitespace>::= _ | /**/
<var2>::= <outer> <whitespace>
<post>::= # | -- | #'
```

Listing 4 Attack grammar for SQLI

The motivation behind this technique is to find a combination of terminals that, when concatenated into one sequence, will be instantiated with concrete values. Such a sequence will represent an attack vector. This link between attack grammars and planning outputs will be explained in the next section in more detail.

4.2 Attack vector model

In the proposed approach, we want to use planning in order to generate concrete test cases in addition to abstract ones. For this sake, we create a new planning specification. This representation, called attack vector model, will guide the generation of concrete values in the further steps. It is important to denote that the function of the planning model from Section 3.1 does not resemble the function of the attack vector model. As already mentioned, XSS and SQLI attacks lack a standard structure. Therefore, the attack vector model is constructed in a way so that plans and, subsequently, concrete test cases with greater variety are generated.

The plans from these models, generated by the same planning system, will contain sequences of actions that resemble symbols from the grammars in Section 4.1. In fact, an obtained sequence of actions will be interpreted as an expression according to the BNF specification. A small example of such interpretation is depicted in Table 2. To be more precise, a single action from a plan corresponds to a specific nonterminal from the grammar. This means that a sequence of actions will result in the selection of a sequence of nonterminals, which will be aligned in accordance to the plan. Basically, a plan can be seen as a concatenation of nonterminals, which represents an expression.

For this sake, a model of attack vectors has to be specified for both vulnerabilities. As was the case in Section 3.1, we rely on PDDL for the planning specification. Both domain and problem definitions for XSS are depicted below.

As can be seen, the labels of actions resemble the names of nonterminals from Section 4.1. The attack vector model for SQLI can be seen below.

Table 2 Mapping of a plan to expression

#	Action
0:	(pre)
1:	(opening)
2:	(wobject)

→

Expression
[pre(x), opening(x), wobject(x)]

```
(define (domain priord)
 (:requirements
    :strips :typing :negative-preconditions)

 (:predicates
    (ininitial ?x)
    (inOpened ?x)
    (inPre ?x)
    (inPost ?x)
    (inClosed ?x)
    (inContented ?x)
    (inContented2 ?x))

 (:action pre
    :parameters (?x)
    :precondition (and (ininitial ?x))
    :effect (and (inPre ?x) (not (ininitial ?x))))

 (:action opening
    :parameters (?x)
    :precondition (and (inPre ?x))
    :effect (and (inOpened ?x)))

 (:action content
    :parameters (?x)
    :precondition (and (inOpened ?x) (inPre ?x)
     (not (inContented ?x)) )
    :effect (and (inContented ?x)))

 (:action post
    :parameters (?x)
    :precondition (and (inContented ?x) (inPre ?x) (inOpened
         ?x))
    :effect (and (inPost ?x)))

 (:action wobject
    :parameters (?x)
    :precondition (and (inPost ?x) (inContented ?x)
     (inPre ?x)
     (inOpened ?x) (not (inContented2 ?x)) (not
     (inClosed ?x)))
    :effect (and (inContented ?x) (inContented2 ?x)))

 (:action closing
    :parameters (?x)
    :precondition (and (inPre ?x))
    :effect (and (inClosed ?x)))

(define (problem priorp)
 (:domain priord)
 (:objects x)
 (:init (ininitial x) (not (inContented x))
        (not (inContented2 x)) (not (inClosed x)))
 (:goal (and (inContented x) (inOpened x) (inClosed x))))
```

Listing 5 Attack vector model for XSS

```
(define (domain priordsqli)
 (:requirements
    :strips :typing :negative-preconditions)

 (:predicates
    (ininitial ?x)
    (inVar ?x)
    (inPre ?x)
    (inVar2 ?x)
    (inClosed ?x)
    (inVared ?x)
    (inPosted ?x))

 (:action pre
    :parameters (?x)
    :precondition (and (ininitial ?x))
    :effect (and (inPre ?x) (not (ininitial ?x))))

 (:action var
    :parameters  (?x)
    :precondition (and (inPre ?x))
    :effect (and (inVar ?x)))

 (:action symbol
    :parameters  (?x)
    :precondition (and (inVar ?x) (inPre ?x) (not
                  (inVared ?x)))
    :effect (and (inVared ?x)))

 (:action var2
    :parameters  (?x)
    :precondition (and (inVared ?x) (inPre ?x) (inVar ?x))
    :effect (and (inVar2 ?x)))

 (:action post
    :parameters  (?x)
    :precondition (and (inVar2 ?x) (inVared ?x) (inPre ?x)
     (inVar2 ?x) (not (inPosted ?x)) (not (inClosed ?x)))
    :effect (and (inVared ?x) (inPosted ?x)))

(define (problem priorpsqli)
 (:domain priordsqli)
 (:objects x)
 (:init (ininitial x))
 (:goal (and (inVared x) (inPosted x))))
```

Listing 6 Attack vector model for SQLI

For the combination of the plans and the attack grammar, we make use of a modified Grammar Solver (2018). In addition, the implementation parses the generated plans and searches for the corresponding nonterminals in the grammar. For each expression, the program will generate concrete inputs according to the BNF grammar. In order to illustrate the idea behind the approach, let's take for example the case in Table 2 from the XSS model specification. For every element of the expression, the implementation searches for the corresponding symbol in the grammar. In this example, the selection will comprise the following symbols.

```
<pre>::= \ | " | ' | > | ; | ! | -
<opening>::= < <html> <content> >
<wobject>::= alert(0) | alert(XSS) | alert('XSS')
```

Now, the implementation recursively generates elements of the grammar for the selected symbols. The output will be a concatenation of these symbols, thus resulting in a new grammar expression. As already stated, the symbols can be terminals, or in case of nonterminals, the program will search further by going down the parse tree. In fact, this will result in the selection of a subpart of the overall attack grammar. In the above example, the symbol `<opening>` includes the following expressions.

```
<html>::= SCRIPT | IMG | ...
<content>::= <attribute> = <value>
<attribute>::= href | src | style | ...
<value>::= alert(document.cookie) | "javascript:alert(XSS)"|...
```

Then, the tree will be traversed where the terminals of each node will be combined into a Cartesian product. Then, the product will be combined again with the terminals of the upper node. This process continues as long as the search inside the entire parse tree of the concatenated expression is not exhausted.

For example, let A, B, and C be matrixes that contain combinations of terminal symbols inside a specific nonterminal. The final set of attack vectors AV will be a Cartesian product of all matrixes.

$$A \times B \times C = AV$$

The individual terminals inside the matrixes are represented by letters. The resulting matrix will represent the final test set of attack vectors. Here the individual elements in the matrix equal a concrete attack vector.

$$\left\{ a \ b \right\} \times \left\{ \begin{matrix} e & f \\ c & d \end{matrix} \right\} \times \left\{ g \ h \right\} = \left\{ \begin{matrix} (a,e,g) & (a,f,g) & (a,c,g) & (a,d,g) \\ (b,e,g) & (b,f,g) & (b,c,g) & (b,d,g) \\ (a,e,h) & (a,f,h) & (a,c,h) & (a,d,h) \\ (b,e,h) & (b,f,h) & (b,c,h) & (b,d,h) \end{matrix} \right\}$$

For the above example, some of the attack vectors will look as follows:

```
\<SCRIPT href="alert(document.cookie)">alert('XSS')
"<IMG style="javascript:alert('XSS');">alert(0)
"<IMG src="javascript:alert('XSS');">alert(XSS)
```

The generated result according to all expressions, i.e. plans, comprises a set of attack vectors. Each individual result represents an attack vector that will be combined with abstract test cases in the next step. The combination of both results will finally lead to the concrete test set. The reason for using the planning-based concatenation represents the fact that for every different plan, a different concrete test set is obtained. If the attack vector models are slightly modified by the tester, the resulting plans will produce a completely different test result. In fact, the sequence of actions, i.e. nonterminals, is only dependent on the specified pre- and postconditions in the planning specification. This is one major difference to other techniques, e.g. regex or fuzzing, where a sequence is usually specified in advance. Figure 2 depicts the entire attack vector generation technique.

By combining planning and attack grammars, we are able to generate concrete test sets that will be used further by the framework. The establishment of a communication between attacker and SUT, as well as the plan-guided test execution, is described in the next section.

5 Test execution framework

After the previous data generation is finalized, a mechanism is needed to combine both input sets. Then, the resulting concrete test cases will be submitted against the SUT. Both tasks are handled by the implemented testing framework. This implementation resembles the communication between the client and the server over HTTP. For this purpose, it implements the functionality of creating, sending, receiving, and reading HTTP messages. Additionally, HTTP responses are parsed and critical elements for the detection of XSS and SQLI are extracted. This plays a crucial role for the test oracle (see Section 5.2), which is consulted every time an attack was carried out.

For functionality of the HTTP communication, we rely on HttpClient (2018), which offers the possibility for configuring HTTP messages. A Java library, jsoup (2018), serves as the parser for incoming responses from the SUT. The framework is programmed in Java. The overall structure of the implemented approach, which combines Figs. 1 and 2, is summarized in Fig. 3.

However, in order to fully comprehend the overall approach, let's take a look on its underlying algorithm in the next section.

5.1 PLAN4SEC 3.0

In the previous sections, we described the use of a planning technique for security testing on two layers. One layer focuses on the generation abstract test cases, whereas the other

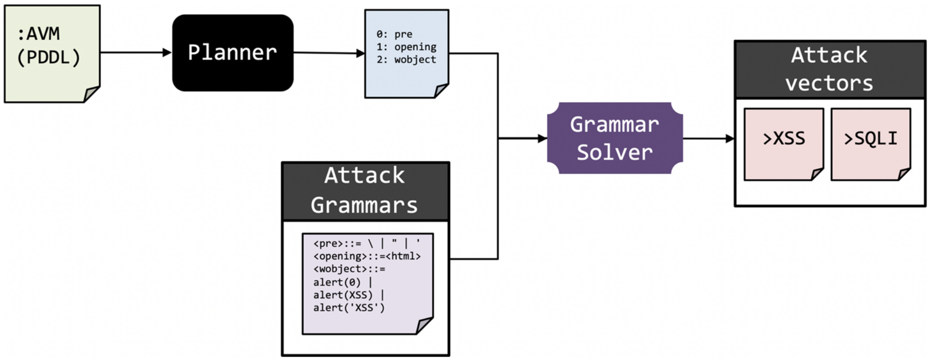

Fig. 2 Concrete test case generation from the attack vector model

deals with the construction of attack vectors. Both planning applications are represented in the form of a plan generation and execution algorithm. In Bozic and Wotawa (2014), the authors of this work introduced the **PLAN4SEC** algorithm that has been improved further in the form of a second version (Bozic and Wotawa 2015). Now, in this work, we present an additionally adapted version, namely **PLAN4SEC 3.0**, which is depicted in Algorithm 1.

Algorithm 1 PLAN4SEC 3.0 – Plan generation and execution algorithm with attack grammars.

Input: Planning model PM, attack vector models AVM, attack grammars $G = \{\mathtt{g}_0, \mathtt{g}_1\}$, set of HTTP methods $M = \{\mathtt{m}_0, \mathtt{m}_1\}$, set of concrete actions $C = \{\mathtt{c}_0, \ldots, \mathtt{c}_n\}$, a set of addresses $URL = \{\mathtt{l}_0, \ldots, \mathtt{l}_n\}$, a function $\Psi = g, AP \mapsto AV$ that maps the grammar and the attack plan to attack vectors and a function $\Phi = a \mapsto c$ that maps abstract actions to concrete ones.
Output: Set of plans $P = \{\mathtt{A}_0, \ldots, \mathtt{A}_n\}$ where each $A_i = \{\mathtt{a}_0, \ldots, \mathtt{a}_n\}$, set of attack vector plans $AVP = \{\mathtt{AP}_0, \ldots, \mathtt{AP}_n\}$ where each $AP_i = \{\mathtt{ap}_0, \ldots, \mathtt{ap}_n\}$, a set of attack vectors $AV = \{\mathtt{av}_0, \ldots, \mathtt{av}_n\}$ and a set of HTML elements $E = \{\mathtt{e}_0, \ldots, \mathtt{e}_n\}$.

```
1:  $P = \emptyset$
2:  $AVP = \emptyset$
3:  $AV = \emptyset$
4:  for $SELECT\ PM, AVM, G, C, URL, M$ do
5:      $A =$ makePlan($PM$)                         ▷ Generate abstract test cases
6:      $P = P \cup \{A\}$
7:      $AP =$ makePlan($AVM$)                       ▷ Generate attack vector plans
8:      $AVP = AVP \cup \{AP\}$
9:      $AV =$ generateAV($G, AVG, \Psi$)             ▷ Generate set of attack vectors
10:     $URL =$ crawl($URL$)                          ▷ Return hyperlinks for an address
11:     while $P.hasNext()$ do
12:         for $l \in URL$ do                        ▷ Pick a URL
13:             for $av \in AV$ do                    ▷ Pick an attack vector
14:                 for $a \in A$ do                  ▷ Execute abstract test case
15:                     $res(A) = FAIL$
16:                     $E =$ parse($l$)              ▷ Identify user input fields
17:                     $a' =$ ConcreteAct($a, \Phi, E, av, M$)   ▷ Generate concrete test case
18:                     if Exec($a'$) fails then      ▷ Execute concrete test case
19:                         $res(A) = FAIL$
20:                     else
21:                         $res(A) = PASS$
22:                     end if
23:                 end for
24:             end for
25:         end for
26:     end while
27: end for
28: Return $(P, a', res)$ as result
```

The main difference to the previous version is the dynamic generation of numerous, more diverse, plans and the inclusion of attack grammars. As was the case before, the planner

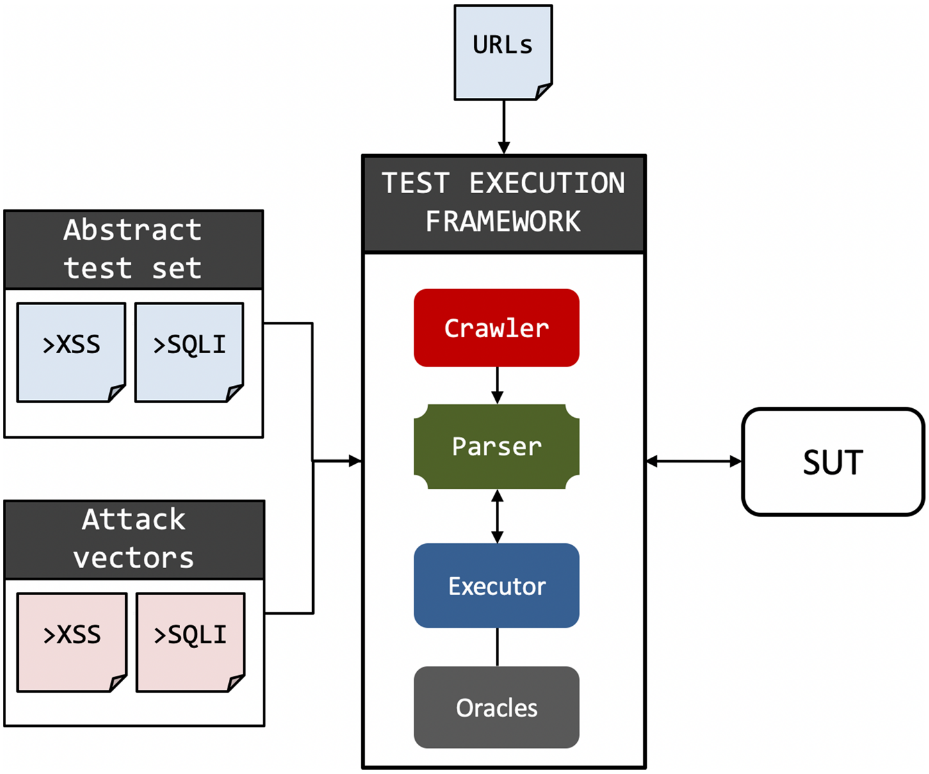

Fig. 3 Planning-based testing framework

relies on the domain and problem definitions in PDDL for both the abstract test cases and attack vector modeling.

First, the planner generates a set of plans (P) from the planning model (PM) that act as abstract test cases. Then, the same planner reads the definitions for both attack vector models (AVM) and returns a set of sequences (AVP) in conjunction with the attack grammars (G). As already stated, the attack grammars are specified in the BNF form. The resulting plans resemble BNF expressions. The function Ψ takes as inputs attack grammars and attack vector plans and returns two sets of attack vectors (AV), for XSS and SQLI respectively. In the framework, this is done with the Grammar Solver. Both test sets are attached separately to the framework and are retrieved at runtime during execution.

Another important part of the framework is the crawler. The task of the crawler is to ensure that a SUT is tested completely. It accepts a manually specified list of URL addresses that are meant for testing and searches for their corresponding hyperlinks. All encountered links are added to the list of seeds (URL). After the list of addresses has been finalized, the testing process can start. In such way, we want to ensure that every web application with all connected hyperlinks is automatically tested.

An abstract plan (A) from P guides the execution on the abstract level. The execution traverses through all generated hyperlinks (l from URL) in search for HTML elements to test. The first attack vector (av) from the set of concrete values (AV) is selected and the actions of the abstract plan guide the execution.

During execution, websites are parsed and various user input HTML elements, e.g. input fields and textareas, are extracted by the mentioned parser. This information (E) is processed for further test execution, which assigns the attack vectors to these input elements. During the testing process, every element (e) is tested against every attack vector from AV. However, in case that multiple input elements are found on a website, every single one of these will be tested individually.

In this approach, we rely only on two HTTP methods (M), namely `GET` and `POST`. For example, a HTTP `GET` attack against a website with two user input elements may look as follows:

```
GET /site/login_form.php?username=<script>alert(0)</script>
&password=
```

Parameter values have to be URL-encoded for submission (e.g. the XSS script `<script>alert(0)</script>` would be submitted as `%3Cscript%3Ealert%280%29%3C%2Fscript%3E`) but for demonstration purposes, we denote the plain form of the attack vectors.

In the example, the parameter `username` is tested while `password` remains blank. However, afterwards, the same attack will be repeated but `password` will be tested so `username` remains empty.

```
GET /site/login_form.php?username=&password=<script>alert(0)
</script>
```

The same principle, when adapted to HTTP `POST`, will have the following form:

```
POST /site/login_form.php HTTP/1.1
Host: w3company.com
username=<script>alert(0)</script>&password=
```

```
POST /site/login_form.php HTTP/1.1
Host: w3company.com
username=&password=<script>alert(0)</script>
```

The same principle is used when testing for SQLI, only the attack vector will differ in that case.

The test executor starts with the first abstract test case (A) from P. However, since abstract actions cannot hold any concrete value, a concretization phase is needed in order to carry out instantiated test cases. This is done automatically by the testing framework.

Here concrete Java methods (C) are implemented that are called according to the current action (a) from an abstract sequence. In fact, every action from the abstract test case corresponds to one Java method. In addition to that, it should be noted that the number of plan does not determine the overall number of concrete test cases. In fact, this is not known in advance but figured out during execution since we do not know how many plans will be generated.

The function Φ takes as arguments an abstract action (a) from the plan (A) and maps it to its corresponding Java method (c). The result represents an instantiated method call, i.e. a concrete test case (a').

Now, every single abstract test case will be executed for every HTML element from a website, i.e. the SUT, against every attack vector. After every test, a test oracle is consulted in order to check whether a vulnerability has been triggered (see Section 5.2). If an action from a plan is executed and a passing test verdict is returned after the test, then we consider this plan to be a successful test case (thus, $res(A) = PASS$). Otherwise, the execution framework concludes a $FAIL$ for the given plan.

Afterwards, the next plan will be picked from the test set. This process will continue as long as attack vectors are available or the execution crashes.

This demonstrates how only one action from the plan, one attack vector for every type of attack, and two input elements from one website will affect the testing procedure. Therefore, the importance of planning is the guidance of the execution. A plan with a different action sequence may lead to different testing results.

5.2 Test oracles

After every attack, the resulting response from the server is parsed and checked according to a criterion. Then, a verdict is given whether the attack was successful with the submitted attack vector. In order to get an insight about the functionality of the test oracle, Figure 4 depicts the behavior of an application that has been tested with a XSS script.

When the user sends data to a website, this can be reflected back to the sender inside the body of the HTTP response. However, this responded information can serve as an indication whether the test was successful or not.

Application-intern security measures include the filtering of user inputs. Whereas some inputs are allowed, others are on the black list. Characters like < and >, among others, are usually on this list. Since XSS relies on scripts being activated and doing some harm, it is important that they do not get detected by input filters. Critical elements are usually neutralized by an application with mechanisms like HTML escape encodings. In such way, critical symbols are rewritten and cannot cause any harm on the victim's size. Here < is escaped with `&lt;` while > is encoded with `&gt;`.

Critical XSS elements encompass a variety of different elements like images (`<img>`) and frames (`<frame>`). In fact, all our XSS attack vectors contain such elements with a code attached to them. The execution framework decides that a test was successful when the submitted script was not detected and is reflected back to the sender in its original form. Thus, the communication would look like the one in Fig. 5.

Before the attack vector is submitted to a website, all of its critical HTML elements are parsed and their occurrence is noted. After the test has been carried out, the implementation reads the response from the server and recounts the critical HTML elements again. Here the number of these elements should be increased by one, since the sent script should have been

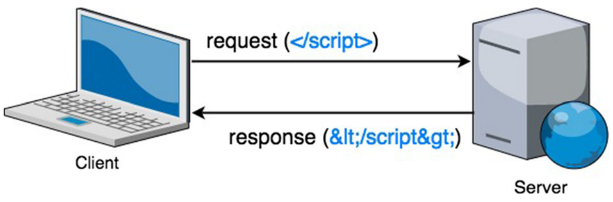

Fig. 4 Filtered XSS input

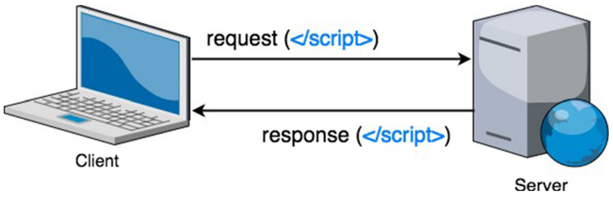

Fig. 5 Unfiltered XSS input

added to the response body as well. For example, if the attack vector contains a `<script>`, then the occurrence of this (unfiltered) element will be higher than before. In this case, our program returns a $PASS$. However, if the number remains the same, then the framework indicates a $FAIL$. In this case, the attempt was probably filtered out or something else happened. The same mechanism is used to test both reflected and persistent XSS, since both rely on a surplus of HTML elements.

SQL injections work similar with regard to input filtering. However, in this case, a SQL statement is submitted in order to access data from the SUT's database. In order for the attack to succeed, information is retrieved from the database and unintentionally sent to the user. Figure 6 shows the procedure. However, since the behavior of an application is impossible to predict, we specify one expected value that the tester excepts to retrieve from the database. For example, if a company's database is tested, then the attacker could figure out information like email addresses via social engineering. This key value is searched automatically inside the server's response body after the attack. If found, the framework returns a $PASS$. Otherwise, the submitted SQL injection will be escaped, similar to XSS, and remains ineffective.

6 Case study

We demonstrate the applicability of our approach to testing of a known web application for testing purposes, namely Mutillidae (2018). Before the execution starts, we let the planner generate a set of plans according to the planning model. It should be noted that a minor change in the planning specification will result in different plans. The tester can manipulate the specification if the necessity emerges or in order to increase the variety of plans. Since the attack vectors for SQLI and XSS are already attached to the implementation, we need only to specify the URL address of the SUT. Since the crawler is in use, the tester can submit the address of the homepage in order to test the application in depth or add the address for one single website. For this demonstration, we add the root URL of the locally employed

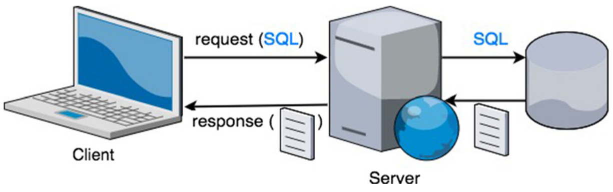

Fig. 6 Procedure for SQL injection

SUT: `http://localhost/mutillidae/`. Then, the crawler returns, among many others, the following links:

```
/index.php?page=dns-lookup.php
/index.php?page=user-info.php
/index.php?page=login.php
/set-up-database.php
```

Now the executioner reads the first plan that resembles the one shown in Section 3.1. The first action `GetSite` in the plan indicates to read the first URL from the list and extracts important data like user inputs and other HTML elements. For the first URL address, the parser returns two fields, namely `username` and `password`. Afterwards, a SQLI attack is carried out against both input elements with the method `GET`. We specify an expected value that resembles an existing entry in the database, e.g. `john@smiths.com`. The first attack vector is `' OR 1 = 1 --` . However, the server's response will be negative for both tested elements since it does not contain the expected email address. Then, another action will instruct to carry out the same attack but with a `POST` method. In contrast to the last attempt, the attack here is successful for both input elements.

The framework grabs the next action and reloads the current URL address again, thereby grabbing its elements. Then, the execution continues to XSS and takes the first input from the list of attack vectors. It submits the input against both elements and checks for the existence of an additional HTML element. In this case, the submitted attack vector is a `<script>alert(0)</script>`. Since it was successful with the `POST` method, the parser encountered a surplus of one `<script>` in the response, thus returning a $PASS$. In fact, a XSS vulnerability was detected in both input elements.

The execution continues further and tries additional XSS and SQLI attempts but with different methods. In both cases, the framework returns a $FAIL$. After the last action from the plan is carried out, the execution repeats the execution but uses other attack vectors and checks whether different values lead to different test verdicts.

After both sets of attack vectors are exhausted, the next URL address will be picked for testing. However, when testing the second entry from the list, we could detect only XSS vulnerabilities since the website does not comprehend a database entry at all.

The execution continues as long as all URLs have been tested and returns a finalized test result overview.

If a plan triggers a vulnerability, then the action sequence and their values that lead to the detection can be traced back. The obtained information can give an insight about the security leak of the application.

7 Evaluation

In this section, we describe the initial test results for the proposed approach. For this sake, the planning-based abstract test cases and attack vectors are taken into consideration. However, the number of concrete test cases does explode in case a high number of abstract test cases are generated, when multiplied with a high number of BNF expressions and the corresponding combination of terminals. Since the size of such test set would be intuitively incomprehensible, we decided to reduce its size drastically for a proof-of-concept evaluation. Therefore, we rely on a small set of only three abstract test cases. Also, only one small expression is used for XSS and SQLI, respectively, with only an excerpt from both attack grammars.

In such way, the framework relies upon 120 concrete test cases in total, 60 for every type of attack. For demonstration purposes, this results in $60 \times 60 = 3600$ concrete inputs for only one action from an abstract plan for only one input field. This gives an overview about the size of a test set with only a small fraction of the overall attack vector model and grammar. It should be noted that the executor tests for both vulnerabilities simultaneously, i.e. by combining every SQLI attack vector with every XSS input in case that the abstract test case instructs an attack.

Three web applications were tested, namely Mutillidae, BodgeIt (2018), and Gruyere (2018). These programs are used for security testing purposes and represent a good starting point for an initial evaluation.

Table 3 depicts the obtained test results according to indicating categories. Besides the tested application (`SUT`), it gives information about the number of found hyperlinks from the initial URL (`#URLs`) and the attack type (`atype`). In addition to that, the exploitation rate, i.e. the number of attack vectors that succeeded in triggering a vulnerability, is defined by the number of individual successful attack vectors, divided by the total number of successful attack vectors (`ER(#)`). This means that the individual attack vector succeeded more than once during testing.

One of the general remarks is the fact that the final test verdicts relies completely on the parser and the test oracles. The parser does extract HTML elements for testing purposes. However, vulnerabilities can be triggered by other means as well, e.g. by inserting the attack vector right after the URL address. This fact was taken into consideration when implementing the framework.

An additional feature is that the executor can bypass eventual intrinsic restrictions on websites. For example, the fixed length of text inputs, hidden input fields, and read-only restrictions are mitigated. Both features influenced the test results.

It should be noted that the depicted test verdicts comprehend all successful test cases. They were able to trigger a vulnerability at least once during execution for a certain HTTP method, i.e. `POST` or `GET`. The latter fact ensures that attacks were carried out for both methods, even if unintended by a website.

When considering all tested SUTs, the test execution consumed the most time when testing Mutillidae. Here 73 hyperlinks have been found by the crawler, with plenty of HTML elements. Tested elements have been, among others, login fields, search fields, and textual posts. For XSS, every tested attack vector was at least once successful in triggering a vulnerability somewhere. Regarding SQLI, only the attack vectors that started with a `'` (apostrophe) triggered a vulnerability. The other test cases that started with a different symbol have been ineffective.

Table 3 Test results for SUTs

SUT	#URLs	atype	ER (#)
Mutillidae	74	XSS	60/902
		SQLI	14/5400
BodgeIt	24	XSS	60/706
		SQLI	0/0
Gruyere	9	XSS	46/92
		SQLI	0/0

For the BodgeIt Store, the obtained results are explained in the following way. One of the obvious observations when analyzing the results is the fact that most of the successful test cases have been achieved when testing input elements that are read-only. For these cases, the test execution framework was able to mitigate these restrictions in many cases and successfully trigger a vulnerability. A possible explanation for this could be that the developer did not implement security mechanisms for these HTML elements since she or he believed that the set restriction alone would hinder an exploitation. Besides that, a search field was the only element where additional XSS could be triggered. In general, this SUT comes in short regarding user input elements. The generated XSS attack vectors have been quite successful. Also, both XSS-related abstract test cases resulted in success at least at one point during the execution. Regarding SQLI, this vulnerability can be triggered. However, once triggered, the expected key value for this type of attack (see Section 5.2) will result in the fact that, once detected for the first time, it will be present at the website in the future. That means that a false positive will be given for every SQLI attack vector afterwards. For this case, we decided to omit this detection from the statistics.

Regarding Gruyere, it is a rather small application without many user input fields. For this SUT, we generated slightly different attack vectors for XSS. In general, the behavior of this application was different than the other applications. Namely, our framework did not trigger any vulnerability on an input field. However, it did succeed in triggering a XSS vulnerability when putting the attack vector just after the current URL, as mentioned above. This fact indicates that a XSS attack attempt can be launched not just by targeting user inputs. Unfortunately, we could not trigger a SQLI with the same approach.

One of the common observation for all tested SUT was the fact that a positive testing verdict was given in case a critical HTML element was encountered. This means that the attack vectors with these unfiltered were considered overall positive.

Despite the positive results for the proposed approach, there are some drawbacks as well. One of the disadvantages of the parser is that it does not take into consideration dynamically created DOM structures on websites. This limits the applicability of the testing framework to a certain degree. This fact also implies that the test oracle cannot be consulted in the aftermath for these cases.

Also, the test oracles put the focus on analyzing critical HTML keywords, e.g. `SCRIPT` or `IMG`, but not on their content, like the attributes and their values.

The relatively small number of SQLI exploits is justified according to our test oracle, i.e. a technique that relies on the definition of expected outputs. Since the structure of the underlying database behind the SUT is not known and the framework cannot directly retrieve its entries, the improvement of its test oracle needs more consideration in the future.

However, some of these drawbacks are due to the implemented framework and do not represent a restriction of the approach itself. Since this work's main focus lies on the test case generation and the proof of the applicability of the planning-based approach, the practical implementation of its parts represents a different challenge.

8 Related work

Two major fields represent the bulk of the work in the presented paper, namely planning and security testing of web applications. Several works address either one or both areas and provide some insight into proposed methods. Planning was adapted on different case studies in different ways. On the other hand, methods like model-based testing focus more on (security) testing issues by usually relying on a graphical representation of the SUT.

The first adaptation of planning to software testing was proposed in 1989 (Anderson and Fickas 1989). In this approach, the authors discuss the idea to formalize the system specification as a planning problem. A detailed introduction in planning is given in Ghallab et al. (2004).

A more advanced approach that combines planning and testing includes Durkota and Lisy (2014). The authors rely on attack graphs by taking into consideration probability and cost measures for choosing attack actions. They introduce an algorithm for selection of an optimal attack strategy and use an attack policy. A policy has a cost so an attacker can determine the probability of attack by choosing different (sub)policies. A Markov decision process (MDP) is used to produce an optimal attack, i.e. an attack that has an overall minimized cost, in the form of an attack graph. The algorithm does an exhaustive search of every action with regard to its cost at any possible decision point during the search. Although a generated plan from our approach resembles an attack path as well, we do not rely on probabilistic calculations for the attack construction but attack with a huge variety of plans instead.

Another work, which discusses a planning-based testing approach of networks is elaborated in Backes et al. (2017). Here an automated pentesting method is applied according to models of SUTs, i.e. networks, by taking into consideration multiple mitigation actions. As was the case with the previous work above, cost and probability assignments to actions are done as well. They use mitigation strategies to simulate an attacker as part of a what-if analysis mechanism. In this way, attack scenarios can be evaluated before their actual execution.

In Shmaryahu et al. (2018), the authors elaborate a testing approach for detecting defects in networks. This represents a learning approach, where the attacker learns from previous attacks. For this case, the authors combine contingent planning (Hoffmann and Brafman 2005) with partially observable Markov decision processes (POMDP). Initially, information about a SUT is obtained by fingerprinting the network. Eventually, this information provides insight about the system's configuration. Then, a PDDL specification is defined and enchanted with probability and cost measures. Plans are extracted from that representation, thereby inferring attack trees that can be executed against the network under test.

In Simos et al. (2018), a planning-based approach is introduced for testing of TLS implementations. The authors introduce a testing framework that is meant for the detection of vulnerabilities during the TLS handshake. For this issue, a planning specification is defined that gives guidelines for negative testing purposes. Several SUTs are stressed with unexpected behavior and values from the client, thereby recording its output. The generated plans guide the attacks that are meant to downgrade the security mechanisms or break the SUT altogether.

In contrast to planning-based testing applications, testing of web applications represents a common issue for a long time.

The authors of Büchler et al. (2012) introduce a semi-automatic model-based testing approach for testing of web applications. The intention behind the work is to apply the results of a model checker for security testing purposes. The process starts with a secure model that is manually generated by a tester. Then, the model is mutated in order to introduce common vulnerabilities like XSS. A model checker generates attack traces and, with the help of an intermediate language, produces the source code. The automated execution proceeds at the browser level of the web application with the help of the Selenium framework.

Appelt et al. (2014) introduced a mutation-based testing technique for SQL injections. The target system represents web services that rely on web application firewalls (WAF). The automated approach, μ**4SQLi**, generates test cases by applying three types of mutations to web service parameters: behavior-changing, syntax-repairing, and obfuscation. With the help of a simple grammars, malicious SQLI are generated that are submitted against two different SUTs. The authors conclude that their approach succeeded in bypassing the WAF of the SUT.

The authors of Simos et al. (2016) introduce a method that combines combinatorial testing with XSS detection. The approach relies on a XSS grammar to derive attack vectors in a combinatorial manner. After execution, the fault localization approach, BEN, analyzes the test cases that were able to trigger a XSS vulnerability. The structure of each input is analyzed in order to identify critical combinations that lead to the detected exploit.

Further offensive testing approaches against web applications and networks can be found in Duchene et al. (2014), Felderer et al. (2016), Sudhodanan et al. (2016), and Shameli-Sendi et al. (2017), respectively. A general introduction and analysis of model-based testing is elaborated in Krämer and Legeard (2016) .

9 Conclusion and future work

In this work, we adapted automated planning and scheduling to security testing of web applications. A planning model is specified in PDDL and serves as an input for the planner. The output is a set of plans that represents abstract test cases that are read by a test execution framework. Every action from a plan guides the execution against the SUT. In addition to that, we implemented a concretization phase that encompasses HTTP functionality. A website is parsed and HTML user input elements are extracted for testing. The crawler supports the execution by adding hyperlinks to the list of initial seeds.

In addition to that, we introduce a new planning-based test case concretization technique. Here we specify two attack grammars for each type of attack. Then, a planner generates sequences that represent expression in the BNF structure of the grammar. Every plan will result in the generation of a specific test set of attack vectors.

Two sets of attack vectors act as user inputs in order to test a SUT against XSS and SQLI. Initial test results with a reduced test set have proven that planning-based security testing is able to detect both types of vulnerabilities in an automated manner.

The use of planning models can help in keeping the representation relatively small but maintain wide test case generation possibilities. Adjustments can be added easily to the model, whereas the plan generation can be controlled by the tester.

In the future, we plan to undertake a more extensive evaluation by using the full possibility of the test concretization. Also, the attack grammars can be extended to generate even more diverse attack vectors. An extension of the PDDL specification for attack vectors would add to the diversity as well.

As already mentioned, the test oracles should be refined in order to better detect vulnerabilities. The main motivation of this work is, besides its introduction, the proof-of-concept based on some first empirical results.

In general, the planning model specification can be extended by taking more real-world activities into consideration. Also, other attack types can be modeled as well. In fact, since a hacking procedure represents a sequence of actions, other malicious activities can be included into the model. Another possibility is the combination of different attack types for more exhausting security testing. A comparison with other state-of-the-art techniques from both planning and testing will help to measure performance metrics of our approach as well.

Funding information Open access funding provided by Graz University of Technology. The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 865248 (Securing Web Technologies with Combinatorial Interaction Testing - SecWIT) and by the Cooperation Programme Interreg V-A Slovenia-Austria under the project AS-IT-IC (Austrian-Slovene Intelligent Tourist Information Center).

References

Anderson, J.S., & Fickas, S. (1989). A proposed perspective shift: viewing specification design as a planning problem. In *Proceedings of the 5th international workshop on software specification and design (IWSSD'89).*

Apache HttpComponents - HttpClient (2018) https://hc.apache.org/httpcomponents-client-ga/. Accessed 2 Feb 2018.

Appelt, D., Nguyen, C.D., Briand, L., Alshahwan, N. (2014). Automated testing for SQL injection vulnerabilities: an input mutation approach. In *Proceedings of the 2014 international symposium on software testing and analysis (ISSTA'14).*

Backes, M., Hoffmann, J., Kunnemann, R., Speicher, P., Steinmetz, M. (2017). Simulated penetration testing and mitigation analysis. arXiv:1705.05088 (2017).

Backus, J.W. (1959). The antics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the international conference on information processing, UNESCO* (pp. 125–132).

Blum, A., & Furst, M. (1995). Fast planning through planning graph analysis. In *IJCAI95* (pp. 1636–1642).

Bozic, J., & Wotawa, F. (2012). Model-based testing - from safety to security. In *Proceedings of the 9th workshop on systems testing and validation (STV'12)* (pp. 9–16).

Bozic, J., & Wotawa, F. (2014). Plan It! Automated security testing based on planning. In *Proceedings of the 26th IFIP WG 6.1 IFIP international conference on testing software and systems (ICTSS'14)* (pp. 48–62).

Bozic, J., & Wotawa, F. (2015). PURITY: a planning-based secURITY testing tool. In *2015 IEEE international conference on software quality, reliability and security-companion (QRS-c)* (pp. 46–55).

Bozic, J., & Wotawa, F. (2018). Planning-based security testing of web applications. In *Proceedings of the 13th international workshop on automation of software test (AST'18).*

Büchler, M., Oudinet, J., Pretschner, A. (2012). Semi-automatic security testing of web applications from a secure model. In *IEEE Sixth international conference on software security and reliability*.

Clarke, J., Fowler, K., Oftedal, E., Alvarez, R.M., Hartley, D., Kornbrust, A., O'Leary-Steele, G., Revelli, A., Siddharth, S., Slaviero, M. (2012). *SQL injection attacks and defense*, 2nd edn. Elsevier: Syngress.

Duchene, F., Rawat, S., Richier, J.L., Groz, R. (2014). KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In *CODASPY* (pp. 37–48): ACM.

Durkota, K., & Lisy, V. (2014). Computing optimal policies for attack graphs with action failures and costs. In *7th European starting AI researcher symposium (STAIRS'14).*

Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschner, A. (2016). Model-based security testing: a taxonomy and systematic classification. In *Software testing, verification and reliability*, Vol. 26.

Fikes, R.E., & Nilsson, N.J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. In *Artificial intelligence* (pp. 189–208).

Fogie, S., Grossman, J., Hansen, R., Rager, A., Petkov, P.D. (2007). XSS attacks: cross site scripting exploits and defense syngress.

Ghallab, M., Nau, D., Traverso, P. (2004). Automated planning: theory and practice. In *Morgan Kaufmann*.

Godefroid, P., Kiezun, A., Levin, M.Y. (2008). Grammar-based whitebox fuzzing. In *Proceedings of the ACM conference on programming language design and implementation (PLDI)* (pp. 206–215).

Google Gruyere - App Engine. (2018) https://google-gruyere.appspot.com/. Accessed: 18 Jul 2018.

Grammar-solver. (2018) https://github.com/bd21/Grammar-Solver. Accessed: 13 Jul 2018.

Hoffmann, J., & Brafman, R. (2005). Contingent planning via heuristic forward search with implicit belief states. In *Proceedings of the 15th international conference on automated planning and scheduling (ICAPS'05)*.

HTML Tutorial. (2018) https://www.w3schools.com/html/. Accessed: 13 Jul 2018.

JavaGP - Java implementation of Graphplan. (2017) https://github.com/pucrs-automated-planning/javagp. Accessed: 11 Dec 2017.

jsoup: Java HTML Parser. (2018) https://jsoup.org/. Accessed: 2 Feb 2018.

Krämer, A., & Legeard, B. (2016). *Model-based testing essentials - guide to the ISTQB certified model-based tester : foundation level*. New York: Wiley.

Kuhn, D.R., Bryce, R., Duan, F., Ghandehari, L.S., Lei, Y., Kacker, N.R. (2015). Combinatorial testing: theory and practice. In *Advances in computers*, Vol. 99.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D. (1998). PDDL - The planning domain definition language. In *The AIPS-98 planning competition comitee*.

Metric-FF. (2016) http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html. Accessed: 2 Dec 2016.

Naur, P. (1960). Revised report on the algorithmic language ALGOL 60. In *Communications of the ACM*, (Vol. 3 pp. 299–314).

OWASP Mutillidae 2 Project. (2018) https://www.owasp.org/index.php/OWASP_Mutillidae_2_Project. Accessed: 4 Feb 2018.

OWASP Top Ten Project. (2018) https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Accessed: 31 Jan 2018.

Raunak, M., Kuhn, D., Kacker, R. (2017). Combinatorial testing of full text search in web applications. In *2017 IEEE international conference on software quality, reliability and security companion (QRS-c)*.

Russell, S.J., & Norvig, P. (1995). *Artificial intelligence: a modern approach*. Englewood Cliffs: Prentice Hall.

Shameli-Sendi, A., Dagenais, M., Wang, L. (2017). Realtime intrusion risk assessment model based on attack and service dependency graphs. In *Computer communications*.

Shmaryahu, D., Shani, G., Hoffmann, J., Steinmetz, M. (2018). Simulated penetration testing as contingent planning. In *Proceedings of the twenty-eighth international conference on automated planning and scheduling (ICAPS 2018)*.

Simos, D.E., Bozic, J., Garn, B., Leithner, M., Duan, F., Kleine, K., Lei, Y., Wotawa, F. (2018). Testing TLS using planning-based combinatorial methods and execution framework. In *Software quality journal (2018)*.

Simos, D.E., Kleine, K., Ghandehari, L.S.G., Garn, B., Lei, Y. (2016). A combinatorial approach to analyzing cross-site scripting (XSS) vulnerabilities in web application security testing. In *IFIP international conference on testing software and systems (ICTSS'16)*.

SQL Injection Bypassing WAF. (2018) https://www.owasp.org/index.php/SQL_Injection_Bypassing_WAF. Accessed: 13 Jul 2018.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G. (2016). Driller: augmenting fuzzing through selective symbolic execution. In *NDSS'16*.

Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. In *Symposium on principles of programming languages* (pp. 372–382).

Sudhodanan, A., Armando, A., Carbone, R., Compagna, L. (2016). Attack patterns for black-box security testing of multi-party web applications. In *NDSS'16*.

The Bodgeit Store. (2018) https://github.com/psiinon/bodgeit. Accessed: 27 Jul 2018.

XSS Filter Evasion Cheat Sheet. (2018) https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Accessed: 13 Jul 2018.

Josip Bozic is a postdoctoral researcher at Graz University of Technology, at the Institute of Software Technology. His research interests include cybersecurity, artificial intelligence and model-based testing of web applications, network protocols and AI systems.

Franz Wotawa is a full professor for software engineering at the Graz University of Technology, at the Institute of Software Technology. His research interests cover the areas model-based reasoning, software testing and software debugging, as well as classical AI.