Plan It! Automated Security Testing Based on Planning

Franz Wotawa, Josip Bozic

Plan It! Automated Security Testing Based on Planning

Franz Wotawa and Josip Bozic*

Institute for Software Technology
Graz University of Technology
A-8010 Graz, Austria
{wotawa,jbozic}@ist.tugraz.at
http://www.ist.tugraz.at

Abstract. Testing of web applications for common vulnerabilities still represents a major challenge in the area of security testing. The objective here is not necessarily to find new vulnerabilities but to ensure that the web application handles well-known attack patterns in a reliable way. Previously developed methods based on formalizing attack patterns contribute to the underlying challenge. However, the adaptation of the attack models is not easy and requires substantial effort. In order to make modeling easier we suggest representing attacks as a sequence of known actions that have to be carried out in order to be successful. Each action has some pre conditions and some effects. Hence, we are able to represent testing in this context as a planning problem where the goal is to break the application under test. In the paper, we discuss the proposed planning based testing approach, introduce the underlying concepts and definitions, and present some experimental results obtained from an implementation.

Keywords: Model-based testing, planning-problem, security testing, SQL injection, cross-site scripting.

1 Introduction

With the ever growing interconnectivity between systems in our world there is an even stronger growing need for ensuring systems' security in order to prevent unauthorized access or other malicious acts. Vulnerable applications do not only cause costs, they also negatively impact trust in applications and consequently in the companies providing the applications. Consequently preventing vulnerabilities should be a top priority of any provider of systems and services. It is worth noting that from 2010 and 2013 some vulnerabilities like SQL injection and cross-site scripting have belonged to the top three web application security flaws (see OWASP Top 10 from www.owasp.org for more information). More interestingly those two flaws have been under the top 6 from 2004 on, leaving

* Authors are listed in reverse alphabetical order.

the impression that there has not been enough effort spent in finding well known flaws.

In our research we focus on the topic of providing methods and techniques for testing web applications with respect to known vulnerabilities. In particular we are interested in automated methods for finding vulnerabilities without or at least with little user interactions. In this paper we introduce a method that is based on planning for computing test cases where a test case is a sequence of interactions with the web application under test. The underlying idea of using planning for test case generation originates from two sources. First, there is already publication available describing testing as a planning problem. Second, and even more important, when having a look of how to break a system, it becomes obvious that providing and executing an attack is nothing else than finding an interaction sequence that finally leads to a situation where we can exploit a vulnerability.

Let us briefly discuss the basic ideas using a variant of cross-site scripting (XSS) on an example used by Hoglund and McGraw [11]. The example makes use of a web application comprising an HTML page, where a user name can be entered, and a server side script handling a request from the HTML page. Communication between a web browser interpreting the HTML page and the server is performed using standard HTTP where sending information to the server is done using the GET message method. Let us now consider the following (partial) HTML page interpreted using the client side browser:

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

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

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

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

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

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

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

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

The contributions of this paper are the following: We present an approach for test case generation and execution in the security domain that is based on planning. We further discuss a first empirical evaluation indicating that the approach has similar capabilities for detecting vulnerabilities of web applications than previous approaches whereas the new approach is easier to adapt and extend.

In the following we discuss our approach. We start with a discussion on related research. Afterwards, we introduce the basic principles of the planning approach to solve security testing based on attacks, and give an algorithm that makes use of a planner for generating test cases, which are executed after generation. We illustrate our approach using a small example, and discuss first empirical results obtained from an implementation. The purpose of the empirical evaluation is to provide evidence that the approach can be used. Hence, we focus on the running time of the algorithm implementation and the capability of revealing the security flaw. Finally, we conclude the paper and discuss future research.

2 Related Work

Planning, i.e., finding actions that lead from an initial state to a goal state, can be considered an old challenge of artificial intelligence. Fikes and Nilsson [7] introduced STRIPS as a planning methodology for solving this challenge, where planning is performed under certain assumptions and where plan generation is separated from its execution. Later Nilsson [15] introduced a planner for solving dynamic real-world problems where plan generation and execution collapses under one framework. There the author introduces the notion of teleo-reactive (T-R) programs, which are artifacts that direct the execution towards a goal by responding to a changing environment according to the perceived sensor data. Teleo-reactive programs might provide a good basis for implementing the behavior of an attacker in the context of security testing. However, in this paper we follow Fikes and Nilssons original work and define attacks as planning problems based on STRIPS planning.

Although, planning is an old artificial intelligence problem, its use for testing is more recent. Howe et al. [12] have been one of the first dealing with the use of planning for test suite generation. Besides the test case generator, the authors compared their test case generator with another technique using a concrete example, i.e., the StorageTek robot tape library command language. Their tool Sleuth made use of the UCPOP 2.0 planner, for plan generation. Howe et al. divided the approach in three parts: problem description generation, plan generation, and transforming the plan into suitable test cases. If a plan was not able to be generated, another initial and goal states were used in order to find a plan fulfilling the final preconditions. Scheetz et al. [16] introduced a very much similar work a plan is derived taking into account initial variable conditions and concrete parameter values.

Froehlich and Link [8] introduced a method to obtain test suites from mapping of UML state charts to a STRIPS planning problem from which plans can be derived using planning tools. The transformation considers the preconditions of transitions. Despite the fact that the whole test suite generation process is automated, the generation of concrete test cases (considering the specification of the system under test) has still to be performed manually.

In [14, 13] the authors proposed an automatic contract-based testing strategy, that combines planning and learning for test case generation. The work is based on the programming language Eiffel where pre- and postconditions for methods can be easily specified. Those pre- and postconditions for methods can be directly mapped into a planning problem from which abstract test cases can be extracted. Galler et al. [9] presented similar work. In their paper, the authors discussed an approach called AIana, that is able to transform Design by Contract^{TM} specification for Java classes and methods into a PDDL representation, which then can be used by a planner for generating plans. The generated plan has to be transformed into Java method calls. Random values are generated for primitive type parameters and recursive calls for non-primitive parameters. The authors also provide two case studies for evaluation purposes. Very similar work includes [18], which elaborates a method for using planning for the generation of test cases from visual contracts. The latter are put into a PDDL representation so that the planning tool LAMA is able to produce a plan.

Armando et al. [1] analysed security issues in security protocols using SAT-based model-checking. The authors proposed a method for attacking these protocols using planning. A protocol insecurity problem specifies all execution paths of a protocol, including the possibilities to exploit security leaks, where the entire protocol is depicted by means of a state transition system. The security properties of the protocol are specified using the tool AVISS. The security properties are transformed into an Intermediate Format (IF) and finally, they are read by the model-checker SATMC so that a planning problem can be generated. The problem itself is represented in SAT using Graphplan-based encoding, which is mapped back into a SAT representation in order to produce a solution.

In this paper we follow our previous work [3–5] regarding vulnerability testing of web applications against common attacks. In those papers the introduced

attack pattern-based testing technique relies on UML state charts representing attack patterns. These attack patterns are executed accordingly to given paths by checking for satisfaction of transition guards in the model. [2] introduced a similar approach. In contrast to these previous papers, we do not use attack pattern models for testing in this paper. For a more general overview about model-based testing and techniques for security testing we refer the interested reader to [19, 6] and [17]. However, all of these approaches do not rely on planning for test case generation.

3 The Security Testing via Planning Approach

Planners are commonly used for intelligent agents and autonomous systems in order to generate action sequences that lead a system from the initial state into a defined goal state. Once specified, these plans instruct the system what to do in each step as long as all actions can be undertaken and typically considering that the environment does not change during plan execution. In our security testing approach we use the generated plan for testing web applications with respect to the wel- known vulnerabilities: SQL injection (SQLI), and reflected as well as stored cross-site scripting (XSS). For this purpose we have to specify the test case generation problem as a planning problem. Let us first define the planning problem in the classical way following [7].

Definition 1. *A tuple (I, G, A) is a planning problem, where I is the initial state, G is the goal state, and A is a set of actions, each of them comprising a precondition and an effect (or postcondition). For simplicity we assume that each state is given as a set of (first order logic) predicates that are valid within this state. We also assume that the preconditions and the effects of an action $a \in A$ can be accessed via a function $pre(a)$ and $eff(a)$ respectively.*

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

Definition 2. *A solution to the planning problem (I, G, A) is a sequence of actions $\langle a_1, \ldots, a_n \rangle$ such that $I \rightarrow_{a_1} S_1 \rightarrow_{a_2} \ldots \rightarrow_{a_{n-1}} S_{n-1} \rightarrow_{a_n} G$.*

In classical planning we assume that there is atomic time, i.e., the execution of an action can be done in finite time and no interruption is possible, there are no exogenous events, the action effects are deterministic, and there is omniscience on the part of the agent. In the context of our application all these assumptions are (more or less) fulfilled, when assuming stateless applications. In order to state security testing as a planning problem, we suggest the following representation:

- Each action that can be performed by an attacker has to be modeled as a planning action, considering the preconditions and the potential effects.

- The initial state considers the currently available information of a web application, i.e., the web address, the script to be executed, and the parameter to be used, etc.
- On the other hand, the goal state specifies what to expect from an application in case of a detected vulnerability.

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

In order to implement the proposed approach, we rely on ordinary planner and planning languages. In particular we assume to use the well-known Planning Domain Definition Language (PDDL) in order to specify the corresponding domain, i.e., the actions that are problem independent, and the problem file, i.e., application specific values, the initial state, and the goal state, which are specific to a certain application.

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

In order to adapt the planning-problem to security testing, we have to define a specific domain and problem description. Furthermore, we consider a generated plan as one abstract test case. An abstract test case is a test case that cannot directly be executed by the system under test (SUT). This is due to the fact that concrete values are missing or that the abstract actions do not provide any information on how to execute them in the current environment. In order to solve this issue and to come to a concrete test case, we specify for each action a corresponding method in Java. This method implements the interaction with the SUT and makes use of concrete values. In particular, this method contains the source code for communication with the SUT, which includes the generation and if necessary reading (parsing) of HTTP messages as well as detection mechanisms for SQLI and XSS. We handle the message traffic between tester and application using HttpClient[1] and jsoup[2] for parsing of responses. For more information on the entire attack and vulnerability detection mechanisms currently in use, we refer the interested reader to our previous publications [4, 5].

[1] `http://hc.apache.org/httpcomponents-client-ga/`
[2] `http://jsoup.org/`

In the following we discuss the algorithm **PLAN4SEC** that is for implementing the described approach. **PLAN4SEC** makes use of an ordinary planner. In our implementation we rely on the planning system Metric-FF[3] from [10], which itself relies on the FF planner[4]. However, our approach is not limited and other planners can be used as well. Algorithm 1 shows the **PLAN4SEC** algorithm.

Algorithm 1 PLAN4SEC – Plan generation and execution algorithm

Input: Domain D, problem file P, set of HTTP methods $M = \{\texttt{m0}, \ldots, \texttt{m}_n\}$, set of types $T = \{\texttt{t0}, \ldots, \texttt{t}_n\}$, set of active states $S = \{\texttt{s0}, \ldots, \texttt{s}_n\}$, set of log values $L = \{\texttt{t}, \texttt{f}\}$ and a function J mapping actions defined in D to Java methods.
Output: Set of plans $PL = \{\texttt{A0}, \ldots, \texttt{A}_n\}$ where each $A_i = \{\texttt{a0}, \ldots, \texttt{a}_n\}$

```
 1: PL = ∅
 2: for SELECT t ∈ T, m ∈ M, s ∈ S, l ∈ L do
 3:     P = Compose(P, t, m, s, l)          ▷ Initialize the planning problem
 4:     A = Planner(P, D)                   ▷ Generate a plan
 5:     PL = PL ∪ {A}
 6:     res(A) = FAIL
 7:     for i = 0 to n do                   ▷ Execute the plan
 8:         if Exec(J(a_i)) fails then
 9:             res(A) = TRUE
10:             Leave the surrounding for block
11:         end if
12:     end for
13: end for
14: Return (PL, res) as result
```

PLAN4SEC uses the domain D and the problem file P as inputs. Moreover, other information is used as well, i.e., (1) the type of attack (T), (2) the HTTP method (M) to be used, (3) the currently state of the testing process (S), (4) finally the information whether the user is logged in the application or not (L), which is stored in a set, and (5) a function J mapping actions to their corresponding Java method. The output of **PLAN4SEC** is a set of plans and a function res mapping $PASS$ or $FAIL$ to each plan, where $FAIL$ is assigned if the plan execution reveals a vulnerability, and $PASS$, otherwise.

PLAN4SEC works as follows: First, the set of results is set to empty. Second a new planning problem is generated, which makes use of a specific configuration. For example, we might want to generate a planning problem where the type t is set to **SQLI** or **RXSS** (reflected XSS). In order to generate such a new planning problem, we make use of the method call **Compose**(P, t, m, s, l), which sets the given values of t, m, s and l in the problem file P. Because of the fact that we test all combinations of values in T, M, S, and L, we achieve a large

[3] `http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`
[4] `http://fai.cs.uni-saarland.de/hoffmann/ff.html`

number of planning problem files. Third, a plan for the domain D and the currently instantiated planning problem P is generated. This plan is added to the set of plans. Afterwards, the plan is executed using the **Exec**$(J(a_i))$ call of a corresponding Java method for a specific action in the plan. If the execution fails, we know that the plan cannot be used for revealing a vulnerability and hence, assign a $PASS$.

Obviously **PLAN4SEC** has to terminate assuming that all called functions terminate. This is due to the fact that all input sets are finite, determining the number of iterations. The algorithm is polynomial in space and time when assuming the execution of external functions in unit time. In the next section we demonstrate our approach making use of a concrete example.

4 Running Example

We demonstrate our approach using the well-known web application DVWA[5]. We explain the plan generation process as well as the execution of plans. As mentioned in the previous section, first we specify the problem and domain files manually, accordingly to our testing purpose and the current SUT. Have a look at the following PDDL description of the problem:

```
(define (problem mbt-problem)
 (:domain mbt)
 (:objects
                x - active
                s - server
                si - status-si
                lo - status-lo
                se - status-se
                type - type
                url - address
                m - method
                a - action
                exp - expect
                un - username
                pw - password
                sqli - sqli
                xssi - xssi
                script - script
                resp - response
 )
 (:init
                (inInitial x)
                (Logged no)
                (not (statusinit two))
```

[5] http://www.dvwa.co.uk/

```
                (Type sqli)
                (= (sent se) 0)
                (not (Empty url))
                (GivenSQL sqli)
                (GivenXSS xssi)
                (Method post)
                (Response resp)
                (not (Found exp resp))
                (not (FoundScript script resp))
)
(:goal (inFinal x)))
```

Problem description in PDDL

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

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

```
(define (domain mbt)
 (:requirements :strips :typing :equality :fluents
    :adl)
 (:types active address server status-si status-lo
    status-se type expect result method integer sqli
    xssi response script)
 (:constants init - active no yes - status-lo two -
    status-si sqli rxss sxss - type get post - method)
 (:predicates
                (inInitial ?x)
                (inAddressed ?x)
                (inSentReq ?x)
                (inRecReq ?x)
                (inParse ?x)
                (inSQLI ?x)
                (inRXSS ?x)
                (inSXSS ?x)
                (GivenSQL ?sqli)
                (GivenXSS ?xssi)
                (inFinal ?x)
 )
 (:functions
                (statusinit ?si - status-si)
```

```
                (Method ?m - method)
)

(:action Start
                :parameters(?x - active ?url - address
                   ?lo - status-lo)
                :precondition (and (inInitial ?x)(not
                   (Empty ?url)))
                :effect (and (inAddressed ?x)(not
                   (inInitial ?x))(Logged yes))
)

(:action SendReq
                :parameters(?x - active ?lo - status-lo
                    ?se - status-se ?si - status-si)
                :precondition (and (inAddressed ?x)
                   (Logged yes))
                :effect (and (inSentReq ?x)(not
                   (inAddressed ?x))(assign(sent ?se)
                   1)(statusinit two)))
)
(:action Finish
                :parameters (?x)
                :precondition (inFound ?x)
                :effect (inFinal ?x))
)
```

Domain description in PDDL

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

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

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

Generated plan

Such a generated plan is read by the parser from JavaFF[6]. Names of the actions are translated into names of the corresponding Java functions. Note in our implementation that the action names and the names of their corresponding Java functions are the same. The Java functions are executed step by step. In addition the Java functions use concrete values for parameters and communication with the web application.

For example, let us assume that we have specified the URL address of the SUT and choose SQLI in the program. In this case we satisfy the initial values ($not(Empty\ url)$) and ($GivenSQL\ sqli$), $sqli$ being the input string. The object x from the type $active$ is meant to give the current status of the execution, for example $inInitial$ states that the execution has just started. When analysing the action definitions, all preconditions are met in order to satisfy the action $Start$. Now the program runs its corresponding method and then reads the next action, executing its next methods afterwards, thereby manipulating concrete variable by it's own. When picking the action $SendReq$, the program will send a HTTP request with HttpClient to the URL address. But, if the tester has not specified this in the program, a discrepancy emerges between the plan and the program. In that case, the program will not be able to follow the plan until the end and the execution stops immediately, setting the plan execution to $PASS$ because of not reaching a vulnerable state. Otherwise, execution continues until reaching the final action of the plan.

5 Evaluation

In order to provide a first evaluation we tested our planning approach to security testing on several web applications, including DVWA, BodgeIt[7] and NOWASP (Mutillidae)[8]. For the evaluation, we implemented the **PLAN4SEC** algorithm in Java. We used a domain and problem file similar to the ones used in Section 4 but extended it substantially. We used in sum 19 action definitions as well as more predicates and initial values. For carrying out the whole evaluation we used

[6] http://www.inf.kcl.ac.uk/staff/andrew/JavaFF/
[7] https://code.google.com/p/bodgeit/
[8] http://sourceforge.net/projects/mutillidae/

three values for the attack type, two for the method, two for the login status and 20 for the current status. The objective of the evaluation was to show the applicability of the approach both in running time as well as the capabilities of detecting vulnerabilities.

DVWA and Mutillidae comprise three difficulty levels, each one of them implementing more sophisticated filtering mechanisms in order to make the application safer against malicious input. Moreover, for every type of attack we used one input string for the concrete test cases. The input string does not cause any harm to the SUT but just indicates whether the application is vulnerable or not. Such an input string in practice might be a good starting point for a potential attacker. For the second application BodgeIt we also make use of one input string. However, this application has no difficulty levels to be set externally.

The obtained results are depicted in Table 1 where for each SUT, the difficulty level (DL), the total time (T) for carrying out the tests, the number of generated plans (#P), the total planning time (planT), the average plan generation time (avgPT), the number of generated actions (#A), the total plan execution time (execT), the average number of generated actions per plan (avgA), and information of how often SQLI, RXSS, and SXSS attacks have been successful, are given. All time values are in second (s).

Table 1. Evaluation Results

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

When executing **PLAN4SEC** on DVWA and Mutillidae, vulnerability could only be triggered on the first two security levels. In both cases, the third one remains impervious. Because of this reason we did not add a row for the third level of these two applications in Table 1. It is worth noting that stored XSS could not be detected on the second level of DVWA too. This is due to the fact that the used input string was successfully filtered by application. The second application BodgeIt has much more SQLI leaks but seems to be more resistant against XSS. From Table 1 we also see that the time for executing **PLAN4SEC** is acceptable. The approach is performed automatically only requiring the user to specify case specific information, like different URL addresses, and different expected values for SQLI. Beside the small amount of adaptation, no further changes were required. This holds especially for the domain specification, which is the same for all SUTs in this evaluation. Note that the Java methods, which correspond to the actions, have to be slightly changed. The unchanged domain

specification is also the reason behind the same number of generated plans for all applications.

The success of the exploitations heavily depends on the used input. Despite the fact that we test the system using different interactions for checking exploits, there is still a need for convert values to be executed. These values have to be adapted (maybe randomly) during execution, which is currently not done. As already said, we used only one type of input string for our evaluation. However, it is worth noting that this is not a principle restriction of the proposed approach.

Because of the relatively high number of potential combination of different input parameters, we received a higher number of plans but also a larger number of successful tests. What might also be interesting is the fact that the average number of actions is rather small. This indicates that the action definitions in the domain specifications use only a small number of preconditions and also originates from the underlying plan generation technique, i.e., Metric-FF. It is also worth mentioning that planning takes much more time than the execution part. Hence, when using the generated test suite for regression testing purposes only the execution time has to be considered, which is the result of the **PLAN4SEC** algorithm.

When we compare this approach and the results given in Table 1 with the results obtained using our previous method relying on models of attack patterns (from [4, 5]), we obtain a similar behavior regarding the detection capability for vulnerabilities for the same web applications. Hence, when considering the much more easy adaptation of the model to different SUTs, the planning based approach is indeed an improvement and worth being further investigated.

6 Conclusion and Future Work

In our work we introduce a novel approach to security testing based on planning. In particular we formalize the test case generation problem as a planning problem and make use of an ordinary planner for generating plans that represent abstract test cases. The algorithm described in this paper is meant for security testing of web applications. Besides test case generation the algorithm also allows for automated test execution. In the paper we discuss the underlying method from predefined specifications and explain its realization. A first empirical evaluation using well-known web applications indicates that the approach can be used in practice. The planning time including plan generation and execution is high but acceptable for the application domain. Also, the capabilities of detecting vulnerabilities are in line with other approaches to automated testing of web applications. In contrast to other testing methods the mapping to planning increases reuse of knowledge used for test generation and also makes adaptations to specific languages much easier. In particular, new actions can be added very simple.

However, we have not reached the limitations of the approach. The number of plans to be generated can be much higher when using either more initial values or planners that deliver different plans and not only one. The latter change can

be straightforwardly integrated into our algorithm. Even more there is space for improvement when considering the concretization of test cases. Here instead of relying only on one mapping, multiple strings or combination of strings might be used. The idea is to map one abstract test case to many concrete test cases. Finally, adding more actions of finer granularities in the initial specification might also increase the number of tests and make the approach more effectively in practice. We will tackle these improvements in our future research directions. In particular we are interested in the influence on the certain improvements to the vulnerability detection. Moreover, we want to improve the empirical evaluation and use more different web applications for this purpose.

Acknowledgement

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice) and ITEA-2 (DIAMONDS).

References

1. Armando, A., Compagna, L., Ganty, P.: Sat-based model-checking of security protocols using planning graph analysis. In: Proceedings of the 12th International Symposium of Formal Methods Europe (FME), LNCS 2805. Springer-Verlag. (2003) 875–893
2. Blome, A., Ochoa, M., Li, K., Peroli, M., Dashti, M.T.: Vera: A flexible model-based vulnerability testing tool. In: Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST'13). (2013)
3. Bozic, J., Wotawa, F.: Model-based testing - from safety to security. In: Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12). (October 2012) 9–16
4. Bozic, J., Wotawa, F.: Xss pattern for attack modeling in testing. In: Proceedings of the 8th International Workshop on Automation of Software Test (AST). (2013)
5. Bozic, J., Wotawa, F.: Security testing based on attack patterns. In: Proceedings of the 5th International Workshop on Security Testing (SECTEST 2014). (2014)
6. Busch, M., Chaparadza, R., Dai, Z.R., Hoffmann, A., Lacmene, L., Ngwangwen, T., Ndem, G., Ogawa, H., Serbanescu, D., Schieferdecker, I., Zander-Nowicka, J.: Model transformers for test generation from system models. In: Conquest 2006. Hanser Verlag, Berlin. (2006)
7. Fikes, R.E., Nilsson, N.J.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence **2** (1971) 189–208
8. Froehlich, P., Link, J.: Automated test case generation from dynamic models. In: Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00), Springer-Verlag (2000) 472–491
9. Galler, S.J., Zehentner, C., Wotawa, F.: Aiana: An ai planning system for test data generation. In: 1st Workshop on Testing Object-Oriented Software Systems. (2010) 30–37
10. Hoffmann, J.: Extending ff to numerical state variables. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02). (2002) 571–575

11. Hoglund, G., McGraw, G.: Exploiting Software: How to Break Code. Addison-Wesley (2004) ISBN: 0-201-78695-8.
12. Howe, A.E., von Mayrhauser, A., Mraz, R.T.: Test case generation as an ai planning problem. In: Automated Software Engineering, 4. (1997) 77–106
13. Leitner, A., Bloem, R.: Automatic testing through planning. Technical report, Technische Universität Graz, Austria (2005)
14. Leitner, A.: Strategies to automatically test eiffel programs. Master's thesis, Technische Universität Graz, Austria (2004)
15. Nilsson, N.J.: Teleo-reactive programs for agent control. In: Journal of Artificial Intelligence Research, 1. (1994) 139–158
16. Scheetz, M., von Mayrhauser, A., France, R., Dahlman, E., Howe, A.E.: Generating test cases from an oo model with an ai planning system. In: Proceedings of The 10th International Symposium on Software Reliability Engineering. IEEE Computer Society, Washington, DC, USA. (1999) 250–259
17. Schieferdecker, I., Grossmann, J., Schneider, M.: Model-based security testing. In: Proceedings of the Model-Based Testing Workshop at ETAPS 2012. EPTCS. (2012) 1–12
18. Schnelte, M., Güldali, B.: Test case generation for visual contracts using ai planning. In: INFORMATIK 2010, Beitr.ge der 40. Jahrestagung der Gesellschaft fuer Informatik e.V. (GI). (2010) 369–374
19. Zander, J., Dai, Z.R., Schieferdecker, I., Din, G.: From u2tp models to executable tests with ttcn-3 - an approach to model driven testing. In: F. Khendek and R. Dssouli, editors, Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference(TestCom 2005), volume 3502 of LNCS. (2005) 289–303