An Approach for Guiding Developers in the Choice of Security Solutions and in the Generation of Concrete Test Cases

Sébastien Salva, Loukmen Regainia

Noname manuscript No.
(will be inserted by the editor)

An Approach for Guiding Developers in the Choice of Security Solutions and in the Generation of Concrete Test Cases.

Sébastien Salva · Loukmen Regainia

the date of receipt and acceptance should be inserted later

Abstract This paper tackles the problems of choosing security solutions and writing concrete security test cases for software, which are two tasks of the software life cycle requiring time, expertise and experience. We propose in this paper a method, based upon the notion of knowledge base, for helping developers devise more secure applications from the threat modelling step up to the testing one. The first stage of the approach consists of the acquisition and integration of publicly available security data into a data-store. This one is used to assist developers in the design of Attack Defense Trees expressing the attacker possibilities to compromise an application and the defenses that may be implemented. These defenses are given under the form of security pattern combinations, a security pattern being a generic and re-usable solution to design more secure applications. In the second stage, these trees are used to guide developers in the test case generation. Test verdicts show whether an application is vulnerable to the threats modelled by an ADTree and whether the consequences of the chosen security patterns are observed from the application (a consequence leading to some observable events partly showing that a pattern is correctly implemented). We applied this approach to Web applications and evaluated it on 24 participants. The results are very encouraging in terms of the two criteria Comprehensibility and Effectiveness.

Keywords Security; Security patterns; Attack Defense Trees; Test case generation.

1 Introduction

One of the main motivations for software security is to prevent attackers from exploiting application defects, in order to compromise the security of critical systems

Sébastien Salva
University Clermont Auvergne, IUT of Clermont-Ferrand, LIMOS, F-63000 CLERMONT-FERRAND, FRANCE
E-mail: sebastien.salva@uca.fr

Loukmen Regainia
University Clermont Auvergne, LIMOS, F-63000 CLERMONT-FERRAND, FRANCE
E-mail: loukmen.regainia@uca.fr

or to disclose and delete user data. But, is this motivation sufficient to counterbalance the layers of complexity required to build secure applications? Software developers must indeed be educated on a wide array of security concerns. They must take security into consideration from the modelling step up to the validation step of an application. Different kinds of expertise are then required, e.g., to represent threats, to select the most appropriate security solutions w.r.t. an application context, to write tests in this context, etc. Besides, these tasks are time-consuming.

Several researchers, organisations or companies have published digitalised security bases, documents and papers, e.g., (OWASP, 2016; Mitre corporation, 2015; del Pilar Salas-Zárate et al, 2015), to guide developers in designing and coding secure applications. For instance, the CAPEC base makes publicly available around 1000 attack descriptions, including their goals, steps, techniques, the targeted vulnerabilities, etc. In another context, security pattern catalogues, e.g., (Slavin and Niu, 2017; Yskout et al, 2015), list 176 re-usable solutions for helping developers design more secure applications. This plethora of diverse documents makes developers drown in a sea of details because these documents take security at different levels of the software life cycle and hence are presented with different viewpoints (attackers, defenders, etc.), abstraction levels (security principles, attack steps, exploits, etc.) or contexts (system, network, etc.). Developers actually lack guidance for choosing security solutions in these documents and for generating concrete security test cases.

This work focuses on these issues and proposes an approach and a tool for helping developers devise more secure applications from the threat modelling step up to the testing one. The originality of the approach resides in the fact that it is knowledge based. Indeed, the initial stage of the method consists of the acquisition and integration of publicly available security documents into a data-store. This knowledge base is then exploited to assist developers in the threat modelling stage, in the choice of security solutions, and in the testing process. Our approach should be suitable for several sort of applications, provided that enough documentation is available to fill the data-store. We consider Web applications in the paper.

We introduced in (Salva and Regainia, 2017b) a method for building a data-store integrating relationships among attacks, attack steps, techniques, security principles and security patterns. Our first contribution is to complete this data-store with new data and relations, e.g., test cases using the pattern "Given When Then" (shortened GWT), so that the data-store can now be employed for testing. The other contributions of the paper are summarised below.

- The approach assists a first team of engineers in the threat modelling stage, which is a process consisting in identifying and describing the attacker goals and capabilities, as well as identifying the potential threats of an application. The approach guides developers in the generation of detailed Attack Defense Trees (ADTrees (Kordy et al, 2012)), which express the attacker possibilities to compromise an application. They also give the defenses that may be put in place to prevent attacks with security patterns. We have chosen this tree model because it offers the advantage of being easy to understand even for novices in security.
- The second part of the approach assists a team of developers in the writing of concrete security test cases. We assume that this team knows how the application is implemented. A test suite is automatically extracted from a given

ADTree and the data-store. The test suite is composed of GWT test case stubs that are associated with generic procedures. These are fulfilled of comments or blocs of code, which aim at helping developers write concrete test cases. Test cases are used to experiment an application under test (shortened AUT), seen as a black-box. The resulting test verdicts show whether the AUT is vulnerable to the threats modelled by an ADTree.
- The test suite is a set composed of lists of ordered GWT test cases, a list being devoted to check whether an AUT is vulnerable to an attack, which is segmented into an ordered sequence of attack steps. This test suite organisation is used to reduce the test costs with the deduction of some test verdicts under certain conditions.
- The GWT test cases also aim to check whether security patterns are correctly implemented in the AUT. As we consider that the application is a black-box, we do not have access to its structure. We hence focus on the consequences of security patterns. These consequences list the impacts of the observable changes brought by the correct implementation of a security pattern. The approach helps developers write test cases to get test verdicts expressing whether security pattern consequences are detected in the AUT behaviour.

The generated test cases could be selected in several testing types or levels. After the completion of the GWT test cases, these can be executed to detect security issues or to verify that an application complies with some standards, as security patterns are useful to evaluate existing systems (Fernandez et al, 2008). Hence, the test cases can be used in System or Operational testing. As soon as some specific functions of an application are available, these test cases can certainly help detect vulnerability, but they can also be executed to check whether security patterns are implemented. Hence, the test cases can help cover both functional and non-functional aspects of an application.

As a proof of concept, we have implemented a tool and generated a data-store specialised to the context of Web applications (Web sites). We employed them to evaluate on 24 human subjects the benefits of using our approach. This evaluation shows encouraging results with regard to the two following criteria, Comprehensibility and Effectiveness.

Paper organisation: Section 2 outlines the background of this work. We introduce the notion of security pattern, the ADTree model, the related work and our motivations. Section 3 presents the data-store architecture. Next, we give an overview of the steps of the approach in Section 4. We describe them more formally in the two next sections. The threat modelling stage is detailed in Section 5, the test case generation and execution are explained and defined in Section 6. Subsequently, Section 7 provides an evaluation of the approach and the threats to validity. We finally conclude in Section 8.

2 Background

2.1 Security Patterns

Security patterns provide guidelines for secure system design and evaluation (Yoder et al, 1998). Schumacher (2003) postulates that *the security pattern intuitively*

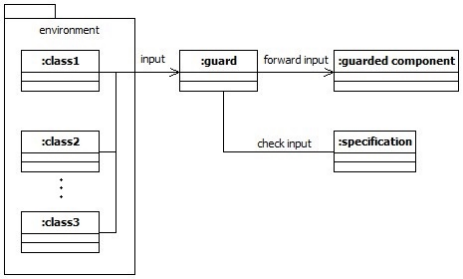


Fig. 1 Class layout of the security pattern "Input Guard".

relates countermeasures to threats and attacks (stated in the problem) in a given context. Generally, security patterns are described with texts or schema (UML diagrams), and are characterised by a set of structural and behavioural properties. Security patterns have to be selected in the design stage, integrated in application models, and eventually implemented.

Several security pattern catalogues are available in the literature, e.g., (Slavin and Niu, 2017; Yskout et al, 2015), themselves extracted from other papers. In these documents, a security pattern is often characterised with its solutions called intents, its interests called forces and the consequences of the pattern, which are observable events resulting from the good integration and implementation of the patterns in the application. A security pattern may have different relationships with other patterns. These relations may noticeably help combine patterns together and not to devise unsound composite patterns. Yskout et al (2006) proposed the following annotations between two patterns: "depend", "benefit", "impair" (the functioning of the pattern can be obstructed by the implementation of a second one), "alternative", "conflict".

As example, Figure 1 portrays the class diagram of the security pattern "Input Guard" whose purpose is to check the validity of inputs. This security pattern structures an application in such a way that the input validation logic is centralised and decoupled from the functional logic of the application. It may benefit from the security pattern "Output Guard" in order to protect the application from disclosing information in case of failures. This pattern is also an alternative to the pattern "Application Firewall".

2.2 Attack Defense Trees (ADTrees)

ADTrees *are graphical representations of possible measures an attacker might take in order to compromise a system and the defenses that a defender may employ to protect the system* (Kordy et al, 2012). ADTrees have two different kinds of nodes: attack nodes (red circles) and defense nodes (green squares). A node can be *refined* with child nodes and can have one child of the opposite type (linked with a dashed line). Node refinements can be disjunctive or conjunctive. The former is recognisable by edges going from a node to its children. The latter is graphically distinguishable by connecting these edges with an arc. We extend these two refinements with the sequential conjunctive refinement of attack nodes, defined by the same authors in (Jhawar et al, 2015). This operator expresses the execution order

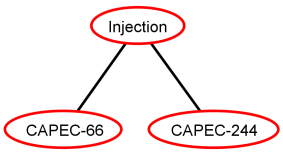


Fig. 2 ADTree example modelling two injection attacks

of child attack nodes. Graphically, a sequential conjunctive refinement is depicted by connecting the edges, going from a node to its children, with an arrow.

For instance, the ADTree of Figure 2 illustrates the goal of an attacker to supply untrusted inputs to an application. The root node is here detailed with a disjunctive refinement connecting two child nodes labelled with IDs of the CAPEC base: the node CAPEC-66 refers to "SQL Injection" and the node CAPEC-244 refers to "Cross-Site Scripting via Encoded URI Schemes".

An ADTree T can be formulated with an algebraic expression called ADTerm and denoted $\iota(T)$. In short, the ADTerm syntax is composed of operators having types given as exponents in $\{o, p\}$ with o modelling an opponent and p a proponent. $\vee^s, \wedge^s, \overrightarrow{\wedge}^s$, with $s \in \{o, p\}$ respectively stand for the disjunctive refinement, the conjunctive refinement and the sequential conjunctive refinement of a node. A last operator c expresses counteractions (dashed lines in the graphical tree). $c^s(a, d)$ intuitively means that there exists an action d (not of type s) that counteracts the action a (of type s). The ADTree of Figure 2 can be represented with the ADTerm $\vee^p$(CAPEC-66, CAPEC-244).

2.3 Related Work

A plethora of papers proposed methods for generating concrete test cases from models to check the security of systems, protocols or applications. Among them, several papers focused on models not to describe the implementation behaviour but rather to express the attacker goals or the vulnerability causes of the system. Such models are conceived during the threat modelling phase of the system (Torr, 2005), which is considered as a critical phase of the software life cycle since *"you cannot build a secure system until you understand your threats!"* (Howard and LeBlanc, 2003). Schieferdecker et al (2012) presented a survey paper referencing some approaches in this area. For instance, Xu et al (2012) proposed to test the security of Web applications with models as Petri nets to describe attacks. Attack scenarios are extracted from the Reachability graphs of the Petri nets. Then, test cases written for the Selenium tool [1] are generated by means of a MIM (Model-Implementation Mapping) description, which maps each Petri net place and transition to a block of code. Bozic et al (2014) proposed a security testing approach associating UML state diagrams to represent attacks, and combinatorial testing to generate input values used to make executable test cases derived from UML models.

Other authors preferred focusing on trees (Attack trees, vulnerability Cause Graphs, Security Activity Graphs and similar models) to represent the threats, attacks or vulnerability causes that should be prevented in an application. From

[1] `http://www.seleniumhq.org/`

these models, test cases are then written to check whether attacks can be successfully executed or whether vulnerabilities are detected in the implementation. Morais et al (2009) introduced a security testing approach specialised for network protocols. Attack scenarios are extracted from an Attack tree and are converted to Attack patterns and UML specifications. From these, attack scripts are manually written and are completed with the injection of (network) faults. In the security testing method proposed by Marback et al (2009), data flow diagrams are converted into Attack trees from which sequences are extracted. These sequences are composed of events combined with parameters related to regular expressions as in (Xu et al, 2012). These events are then replaced with blocks of code to produce test cases. The work published in (El Ariss and Xu, 2011) provides a manual process composed of eight steps. Given an Attack tree, these steps transform it into a State chart model, which is iteratively completed and transformed before using a model-based testing technique to generate test cases. In (Marback et al, 2013), test cases are generated from Threat trees. The latter are previously completed with parameters associated to regular expressions to generate input values. Security scenarios are extracted from the Threat trees and are manually converted to executable test scripts. Shahmehri et al (2012) proposed a passive testing approach, which monitors an AUT to detect vulnerabilities. The undesired vulnerabilities are modelled with security goal models, which are specialised directed acyclic graphs showing security goals, vulnerabilities and eventually mitigations. Detection conditions are then semi-automatically extracted and given to a monitoring tool.

Besides security testing, some works tackled the verification or testing of security patterns. Verification of patterns on models was studied in (Dong et al, 2010; Hamid et al, 2012; Kobashi et al, 2015; Regaigna et al, 2016). In these papers, security pattern goals or intents or structural properties are specified with UML sequence diagrams (Dong et al, 2010) with OCL expressions (Hamid et al, 2012; Kobashi et al, 2015) or with LTL properties (Regaigna et al, 2016) and are verified with tools on UML models. But, these papers are out of the scope of this work, which deals with security pattern testing. Surprisingly, we found only one paper about this topic. Yoshizawa et al (2014) introduced a method for checking whether the behavioural and structural properties of security patterns are detected in application traces. Given a security pattern and a UML model of the application, two test templates (OCL expressions) are written, one test Aspect template to specify the pattern structure and another test template to describe its behaviour. Then, developers build test cases by making these templates concrete with respect to an application model (class and sequence diagrams). They have to provide input values and write Selenium scripts. The application, which is instrumented with debugging tools, is experimented with these test cases to return a set of traces (method calls). Finally, the approach checks whether the OCL expressions given in the test templates hold on the trace set.

2.4 Open Issues and Contributions

On the one hand, the writing of detailed threat models requires a lot of expert knowledge and of documents. The referred papers neither guide developers in the threat modelling phase nor provide security solutions. On the other hand, some methods propose to generate test cases from (formal) specifications. These test

cases are often abstract, they cannot be directly used to experiment an AUT. Some methods tried to tackle this problem using a mapping technique. However, this kind of technique is usually very limited in its capability to translate abstract tests into concrete ones. Hence, most of the security testing approaches, especially those taking threat models as inputs, rely on developers to write concrete test cases. But, they do not give any recommendation on how to write and structure executable tests or to make them reusable.

This work brings together the notions of security documents, threat modelling, the writing of concrete test cases and their executions to help developers in these tasks. Once he or she has given its initial test requirements with a first ADTree, our approach semi-automatically completes it with attack steps, techniques and defenses. Our approach assists developers in the test suite generation, by structuring test cases and by completing them with comments or blocks of code. The test case execution provides verdicts expressing whether the application is vulnerable to the threats modelled in the ADTree or whether its behaviour includes the observable consequences of the security patterns. Hence, our approach is compatible with and complementary to the method given in (Yoshizawa et al, 2014). We have also taken into consideration several quality criteria to design this approach. In particular, we emphasised Navigability and Comprehensibility, which are quality criteria, given in (Alvi and Zulkernine, 2012; Yskout et al, 2015), respectively related to: the ability to direct a software designer among collaborative and related patterns; the ease to understand patterns by both a novice and expert developer. Furthermore, we have concentrated our efforts on several criteria given by Rojas et al (2015). These authors studied the effects of using an automated test generation tool during development and evaluated some criteria on human subjects. They concluded that the Readability of the test cases, the Integration of the test generation approaches in the software life cycle and Education are the most important criteria to improve the Efficiency and Effectiveness of developers. These two last criteria are used for evaluating our approach.

Prior to this paper, we proposed a semi-automatic data integration method in (Salva and Regainia, 2017b) to extract security pattern classifications. Section 3.1 summarises the results of this work used in this paper, i.e., the first meta-model version of the data-store. For this paper, we extend the data-store meta-model to support the generation of concrete security test cases. We proposed a preliminary version of this work in (Salva and Regainia, 2017a). The present paper is an extended version, which provides an overview of the approach from the developer's viewpoint, details the ADTree generation, adds the inconclusive test verdict , and includes additional evaluation results and examples. Besides, we consider in this paper that the most concrete attacks are detailed with sequences of steps, and we generate test suites composed of ordered lists of test cases (we always used sets in the original paper). We take advantage of this sequential order of steps to propose a test case model of execution based on real executions and implicit deductions. This helps reduce test costs.

3 Data-store Architecture

Our approach is based on the notion of knowledge extracted from various available resources, which are integrated into a data-store. We introduce in this section the

data-store architecture we devised to later generate Attack Defense Trees and test cases. We refer to (Salva and Regainia, 2017b) for the detailed description of the manual or automatic steps required to build the first part of this data-store. Then, we present an extension, which also integrates the notions of GWT test cases and test architectures.

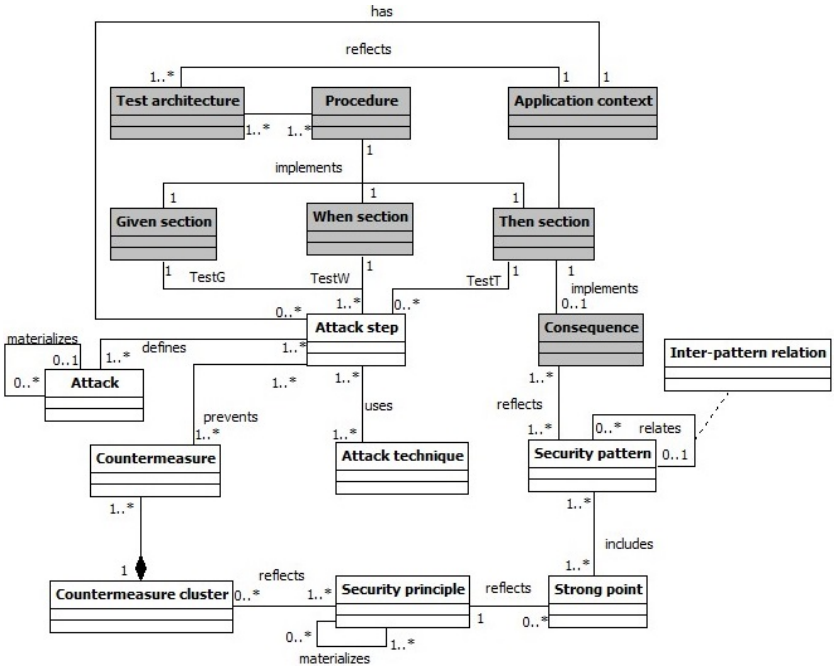


Fig. 3 Data-store meta-model

3.1 Data-store Meta-model for the ADTree Generation

We proposed in (Salva and Regainia, 2017b) a semi-automatic data integration method to build a data store, which exposes relationships among attacks of the CAPEC base, security principles and security patterns of the catalogue given in (Yskout et al, 2015). Figure 3 exposes the meta-model used to structure this data-store (white entities). The entities refer to security properties and the relations encode associations among them.

This meta-model is the result of observations we made from the literature and some security documents, e.g., the CAPEC base: we consider that an attack can be documented with more concrete attacks, which can be segmented into ordered steps; an attack step provides information about the target or puts an application into a state, which are reused by a potential next step. Attack steps are performed with techniques and can be prevented with countermeasures.

Security patterns are characterised with strong points, which are pattern features extractable from their descriptions. The meta-model also captures the inter-pattern relationships defined in Yskout et al (2006), e.g., "depend" or "conflict".

In both sides, countermeasures and strong points refer to the same notion of attack prevention. But finding direct relations between countermeasures and strong points is tedious as these properties have different purposes. To solve this issue, we used a text mining and a clustering technique to group the countermeasures that refer to the same security principles, which are desirable security properties. To link clusters and strong points, we chose to focus on these security principles as mediators. As exposed in Figure 3, we organise security principles into a hierarchy, from the most abstract to the most concrete principles. We provide a complete description of this hierarchy in (Salva and Regainia, 2017b). In short, we collected and organised 66 security principles covering the security patterns of the catalogue given in Yskout et al (2015). The hierarchy has four levels, the first one being composed of elements labelled by the most abstract principles, e.g., "Access control", and the lower level exhibiting the most concrete principles, e.g., "File authorization".

3.2 Test Case Representation and Data-store Meta-model Update

We consider in this paper that a test case is a piece of code that lists stimuli supplied to an AUT and responses checked by assertions assigning (local) verdicts. To make test cases readable and re-usable, we use the behaviour driven approach using the pattern "Given When Then" (shortened GWT) to break up test cases into several sections:

- Given sections aim at putting the AUT into a known state;
- When sections trigger some actions (stimuli);
- Then sections are used to check whether the conditions of success of the test case are met with assertions. In the paper, we consider two kinds of Then sections. We use Then sections to check if an AUT is vulnerable to an attack step st. In this case, the Then section returns the verdict "$Pass_{st}$". Otherwise, it provides the verdict "$Fail_{st}$". When a unexpected event occurs, we also assume that "$Inconclusive_{st}$" is returned. Furthermore, we consider other Then sections for testing the detection of pattern consequences in the AUT behaviour. These Then sections return "$Fail_{sp}$" if a consequence of the security pattern sp is not detected.

To later generate GWT test case stubs, we extend the data-store briefly mentioned before with new entities and relations. The additional entities are depicted in grey in Figure 3. We now associate every attack step to one Given, When, and Then section. Likewise, we map every security pattern consequence onto one Then section. In addition, a test case section is linked to one procedure, which implements the test case section. A section or a procedure can be reused with several attack steps or security patterns.

The meta-model of Figure 3 also reflects the fact that an attack step is associated with one "Test architecture" and with one "Application context". The former refers to textual paragraphs explaining the points of observation and control, testers or tools required to execute the attack step on an AUT. An application context refers to a family, e.g., Android applications, or Web sites. As a consequence, a GWT test case section (and procedure) is classified according to one application context and one attack step or pattern consequence.

```
@When("^spider the application")
public void theApplicationIsSpidered() {
//  Try one of the following techniques :
//    1. Spider web sites for all available links
//    2. Sniff network communications.
url = "URL to be scanned";
try { spider(url);} catch (InterruptedException e){e.printStackTrace();}
waitForSpiderToComplete();}
```

Fig. 4 An example of procedure related to a GWT test case section When.

In some specific application contexts, a procedure, composed of comments or of blocks of code, can be reused with all the applications of the same context. This is usually the case for procedures calling penetration testing tools. For instance, Figure 4 illustrates a procedure calling the tool Zaproxy [2] to explore a Web application through its URLs. This procedure needs to be completed as no URL is provided. But it is generic in the sense that it can be reused with any Web application. We call this kind of procedures *Generic procedure* and we propose to store them into the data-store to ease the test case development. The data-store must only contain generic procedures related to an Application context.

Definition 1 (Generic procedure) Let C be an application context. A generic procedure is a block of code, related to a Given, When or Then test case section that can be used with any application of the context C.

As a proof of concept, we have generated a data-store specialised for the Web application context. The paper (Salva and Regainia, 2017b) details the steps that perform the data acquisition and integration. The data-store includes information about 215 attacks (209 attack steps, 448 techniques), 26 security patterns (43 consequences, 36 strong points), 66 security principles. We also generated 627 GWT test case sections (Given, When and Then sections) and 209 procedures. The latter are composed of comments explaining: which techniques can be used to execute an attack step, which observations reveal that the AUT is vulnerable, or which observations indicate that a pattern consequence is not detected. Examples of procedures are given in Figures 8 and 9. With the Web application context, we observed that several procedures can be generic. We manually completed 32 procedures, which cover 43 attack steps. We used the testing framework Selenium and the penetration testing tool ZAProxy, which covers varied Web vulnerabilities. This data-store is available in (Regainia and Salva, 2017).

4 Approach Overview

We introduce in this section a motivating example that covers the main steps of our approach from the developer's viewpoint (the formal aspects are concealed here). We have implemented the approach into a tool publicly available in (Regainia and Salva, 2017), which consists of two main parts: a set of command lines allowing to build the data-store, and a software program that semi-automatically generates ADTrees and GWT test cases. This tool only supports Web applications (websites)

[2] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

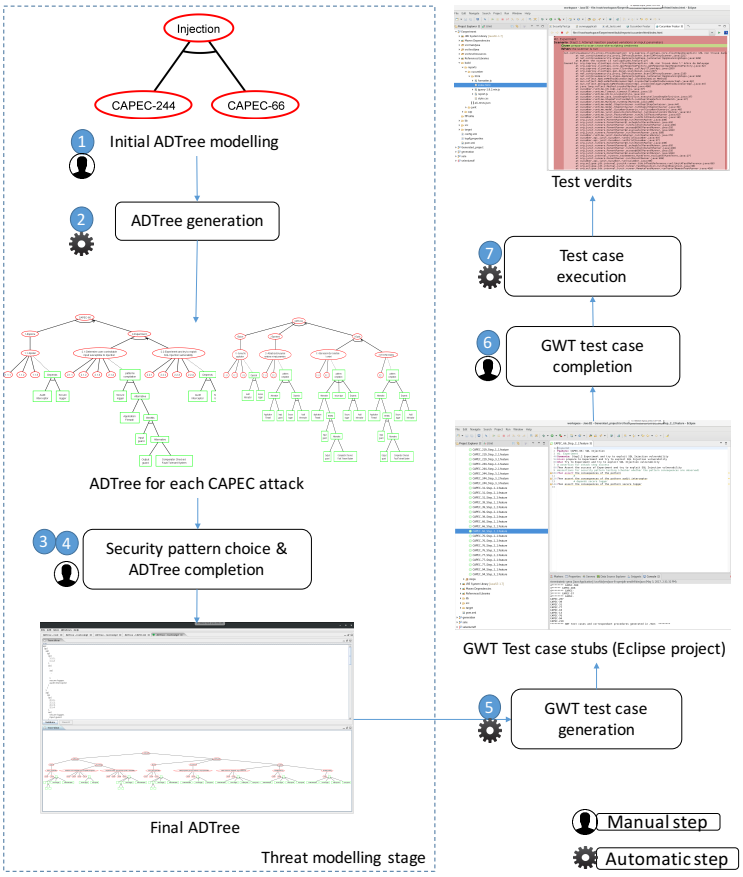


Fig. 5 Threat modelling and test case generation

at the moment. This is why we consider the AUT is here a Web application whose behaviour is captured by means of its HTTP messages.

As illustrated in Figure 5, the purpose of this approach is twofold. During the requirement analysis, it aims at guiding developers through the elaboration of a threat model (left side of the figure). Then, it helps generate and execute test suites to check whether the AUT is vulnerable to attacks and whether security pattern consequences are detected in the AUT behaviour (right side of the figure).

4.1 Threat Modelling Stage

The threat modelling is split up into four steps.
Step 1: the developer establishes a first raw ADTree T_0 whose root node represents the attacker goal and child nodes its refinement. We restrict the relations between nodes to disjunctive and conjunctive refinements here. The tool *ADTool*, proposed by Kordy et al (2013), can be used to edit this ADTree. An ADTree example is given in Figure 2 and is taken back in Figure 5. The ADTree T_0 is often incomplete, hence implementing a secure application or deriving test cases from it remains a tedious task.

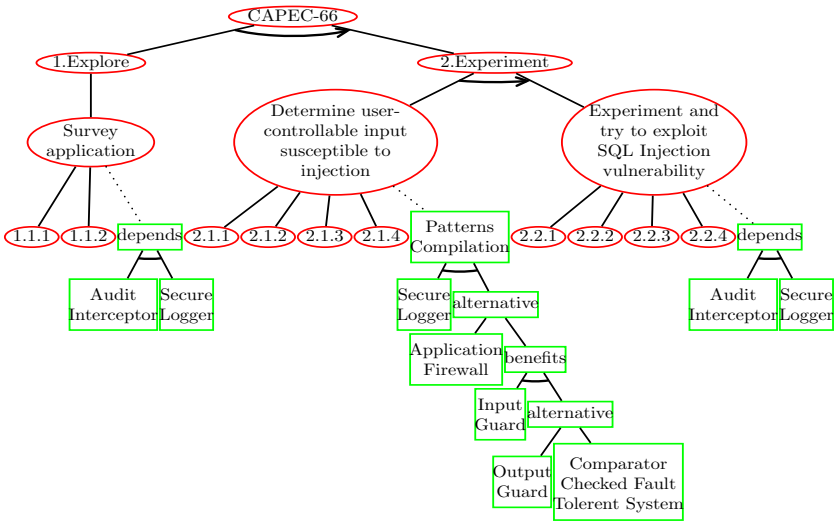


Fig. 6 ADTree of the Attack CAPEC-66

Step 2: the approach generates an ADTree for every attack node labelled with a CAPEC attack in T_0. This new ADTree subdivides the CAPEC attack into sub-attacks and so forth. The lowest attack steps are linked to techniques (attack leaves) and to defense nodes expressing security pattern combinations. Every ADTree can be edited with the tool *ADTool*. For instance, Figure 6 depicts the ADTree of the attack CAPEC-66, which was exported from ADTool. Every attack step has a defense node expressing pattern combinations. The attack leaves are techniques. The attack step 1.1, which relates to the application exploration or "spidering" can be countered with a combination of two security patterns "Audit Interceptor" and "Security Logger".

Step 3: the developer edits the generated ADTrees. He or she can remove some attack nodes and has to select a combination of security patterns for each attack step node in such a way that a defense node only has conjunctive refinements. These conjunctive refinements represent the security pattern sets that have to be integrated and implemented in the application. The generated ADTrees are designed to guide him or her in this step. Indeed, patterns are combined with classical logic operations. Besides, ADTrees provide inter-pattern relationships revealing the conflicting or dependent patterns. During these editions, we believe that a novice in security is educated on attacks and security patterns, as an ADTree gives the attack functioning and the related defenses.

If we take back the example of ADTree given in Figure 6, the step 2.1 can be countered with the patterns "Secure Logger", "Input Guard" and "Output Guard". The last two patterns may be replaced by "Application Firewall".

Step 4: the tool automatically completes the ADTree T_0 with the generated ones and yields one final ADTree denoted T_f.

```
@capec66
Feature: CAPEC−66: SQL Injection
#2. Experiment
Scenario: Step2.1 Determine user−controllable input susceptible to injection
Given prepare to Determine user−controllable input susceptible to injection
When Try to Determine user−controllable input susceptible to injection
# assertion for attack step success
Then Assert the success of Determine user−controllable input susceptible to injection
#assertions for security pattern testing (observable pattern consequences)
Then Assert the consequences of the pattern Input guard
Then Assert the consequences of the pattern Output guard
Then Assert the consequences of the pattern Secure logger
```

Fig. 7 An example of Feature file

4.2 Test Case Generation and Execution

Step 5: for every attack step labelled in T_f, the approach automatically generates a GWT test case stub, which is composed of two parts. A first file called *feature* contains scenarios, themselves split up into GWT test case sections. The test case sections still have a high level of abstraction, but they refer to procedures that implement them. These are stored into a second file. Our tool generates GWT test case stubs based up the Cucumber framework, which supports several languages. At the moment, our tool generates test case stubs written with Java and Junit, which are gathered into an Eclipse project.

Figure 7 lists the GWT test case generated for the attack step "2.1" illustrated in the ADTree of Figure 6. Two kinds of Then sections are used. The first Then section aims at testing whether the AUT has an interface susceptible to injection. The last three Then sections test whether the consequences of the security patterns "Input Guard", "Output guard" and "Secure logger" are detected from the AUT behaviour (HTTP messages with our example).

Step 6: the developer has to complete the procedures of the GWT test case stubs. We believe that the separation of every test case into sections and the links to the ADTree T_f (associations among steps, security patterns and procedures) make this step easier.

In our context of Web applications, a procedure can be implemented by means of a testing framework such as Selenium or can call penetration testing tools. Figure 8 shows an example of complete procedure related to the GWT test case section "When Try to Determine user-controllable input susceptible to injection". The comments correspond to the techniques given in the ADTree T_f, extracted from the data-store. This procedure was completed to call the tool ZaProxy to apply the second technique. The tool covers the application by means of the URLs collected by the previous step Explore of the attack CAPEC-66. ZaProxy tries to inject code and malicious inputs in all of the HTTP requests and analyses the HTTP responses. If the application is vulnerable, alerts are received in the section "Then Assert the success". Figure 9 shows another example of procedure implementing the section "Then Input Guard security pattern is present". If "Input Guard" is correctly implemented, the sending of erroneous inputs should bring the application to a quiescent state (observed with the HTTP status 503, 408) or the latter should return messages reporting that errors have been detected.

```
@When("Try to Determine user-controllable input susceptible to injection")
public void trydetermineusercontrollableinputsusceptibletoinjection (){
//  Try one of the following techniques :
//1.  Use web browser to inject input through text fields or through HTTP GET
      parameters.
//2.  Use a web application debugging tool such as Tamper Data, TamperIE,
      WebScarab,etc. to modify HTTP POST parameters, hidden fields, non-freeform
      fields, etc.
//3.  Use XML files to inject input.
//4.  Use network-level packet injection tools such as netcat to inject input
//5.  Use modified client (modified by reverse engineering) to inject input.
List<HarEntry> URLlist = j.getHistory();
for (int i=0 ; i<URLlist.size();i++ ) {
url = j.getHistory().get(i).getRequest().getUrl();
j.scan(url);
int complete = 0;
int scanId = j.getLastScannerScanId();
while (complete < 100) {
complete = j.getScanProgress(scanId);
try {Thread.sleep(1000);} catch (InterruptedException e) { e.printStackTrace();} }}}
```

Fig. 8 The procedure related to the When section of Figure 7

```
@Then("^Input Guard security pattern is present$")
public void Input_Guard_security_pattern_is_present() {
try{
//check that an erroneous input is not propagated
//Recept of empty outputs or outputs showing unauthorised accesses
//Recept of HTTP status showing incorrect, unauthorised accesses
assertThat("Pass_sp",app.getdriver().getPageSource(), anyOf(equals(""),
      containsString( "error"), containsString("forbidden"),
      containsString("unauthorized")));
//HTTP status(503, 408 for quiescent state, the others for Unauthorized accesses)
assertThat("Pass_sp",con.getResponseCode(),anyOf(is(200),is(503),is(408),is(400),
      is(401),is(403),is(405),is(409),is(500));
}catch(Exception e){fail("Inconclusive_sp")}}
```

Fig. 9 The procedure related to the last THEN section of Figure 7

Another possible consequence of the pattern is that the application may stop its execution or crash (which can be observed with the HTTP status 500). However, the fact of catching an unexpected exception leads to the inconclusive verdict.

Step 7: once, all the test cases are completed, the developer deploys a test architecture. The report generated in Step 4 helps him or her in this task. Thereafter, the AUT is experimented with test cases. Several test verdicts are pulled out of the test execution logs. These are defined in Section 6. Table 1 informally summarises the meaning of some test verdicts and some corrections that may be followed in case of failure.

5 Threat Modelling, Attack Defense Tree Generation

This section describes more formally the threat modelling phase depicted in Figure 5. We introduce a few definitions that we use to establish test verdicts.

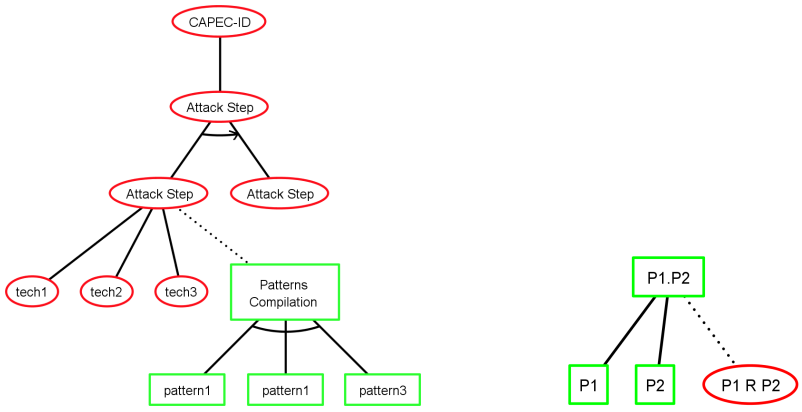
Table 1 Test verdict Summary and solutions

Vulnerable(T_f)	Unsat^c($SP(T_f)$)	Inconclusive(T_f)	Corrective actions
False	False	False	No issue detected
True	False	False	At least one attack-defense scenario is successfully applied on the application. Fix the pattern contextualisation or implementation. Or the chosen patterns are inconvenient.
False	True	False	Some pattern consequences are not detected from the AUT behaviour. Check the pattern implementations. A pattern may be incorrectly implemented or another pattern conceals the consequences of the former.
True	True	False	The chosen security patterns are useless or incorrectly implemented. Fix the security patterns, models and AUT.
T/F	T/F	True	The test case execution crashed or returned unexpected exceptions. Check the test architecture, the application interfaces, or the test case codes.

The developer firstly builds a first ADTree T_0, which describes some combination of attacks that can be applied on the application to be developed and tested. Different methods can be followed, e.g., DREAD (OWASP, 2016), to build this threat model. As our data-store is framed upon the CAPEC base, we assume that the ADTree T_0 has leaves labelled by CAPEC attack identifiers. Otherwise, a semantic alignment may be required to replace some attack labels by similar attack identifiers available in the CAPEC base.

As stated previously, for every attack A labelled in T_0, an ADTree $T(A)$, is generated from the data-store. This ADTree has the generic form given in Figure 10(a) and is achieved by means of the following steps:

1. the ADTree $T(A)$ has a root attack node labelled by A. This root node is linked to other attack nodes A_i with a disjunctive refinement if the attack has sub-attacks. This step is repeated for every sub-attack;
2. for each attack A_i of the preceding tree, we collect all its steps from the data-store. The node labelled by A_i is refined with a sequential conjunction of attack nodes, one for each of its steps. We repeat this process if a step is itself composed of steps. For each step st, the related techniques are extracted from the data-store and are linked to the node labelled by st with a disjunctive refinement;
3. for each step st, we extract the set of security patterns P that are countermeasures of st. Given a couple of patterns $(p_1, p_2) \in P$, we illustrate their relationships with new nodes and logic operations as follows. If we have:
   - $(p_1\ R\ p_2)$ with R a relation in $\{depend, benefit\}$, we build three defense nodes, one parent node labelled by $p_1\ R\ p_2$ and two nodes labelled by p_1, p_2 combined with this parent defense node by a conjunctive refinement;
   - $(p_1\ alternative\ p_2)$, we build three defense nodes, one parent node labelled by $p_1\ alternative\ p_2$ and two nodes labelled by p_1, p_2, which are linked by a disjunctive refinement to the parent node;
   - $(p_1\ R\ p_2)$ with R a relation in $\{impair, conflict\}$, we would want to use the xor operation. Unfortunately, the latter is not available with ADTrees. Therefore, we use the sub-tree depicted in Figure 10(b) where the simultaneous use of p_1 and p_2 is modelled by an attack node labelled by $(p_1\ R\ p_2)$ meaning that two conflicting security patterns used together constitute a kind of attack;



(a) Generic example of ADTree

(b) Conflicting pattern modelling with ADTree

Fig. 10 ADTree general forms

- p_1 having no relation with any pattern p_2 in P, we add one parent defense node labelled with p_1.

The parent defense nodes are combined to a defense node labelled by "Pattern Composition" with a conjunctive refinement. This last defense node is linked to the attack node labelled by *st*.

The developer can now edit every ADTree $T(A)$. He or she also has to select combinations of security patterns in such a way that the defense nodes linked to attack nodes only have conjunctive refinements of nodes labelled by security patterns. The resulting ADTrees are composed of attacks, having different levels of abstraction: with regard to Step 2, a root attack node may be disjunctively refined with other attacks and so on; we assume that only the most concrete attacks are materialised by ordered sequences of steps. We call them *concrete attacks*. We formulate in the next proposition that these nodes or sub-trees can also be encoded with ADTerms:

Proposition 1 *Given the ADTree $T(A)$ achieved by the previous steps, the ADTerm $\iota(T(A))$ representing $T(A)$ has one of these forms:*

1. *ca_1, modelling a concrete attack;*
2. *$\vee^p(a_1, \ldots, a_n)(n > 1)$, with a_i an attack having a form given either in 1) or 2).*

We denote $\mathrm{CA}(T(A))$ the set of concrete attacks labelled in an ADTree $T(A)$. A concrete attack of an ADTree $T(A)$ is composed either of a sequential conjunction of steps or of one step only. These steps may be disjunctively refined by techniques and are linked to a defense node expressing a security pattern combination. In the remainder of the paper, we define the ADTerm modelling a step node by a *Basic Attack Defence Step*, shortened as BADStep. The GWT test cases actually come from these BADSteps.

Proposition 2 (Concrete Attacks) *Let $T(A)$ be an ADTree. $ca \in \mathrm{CA}(T(A))$ is expressed by an ADTerm having one of these forms:*

1. $c^p(st, sp)$;
2. $\overrightarrow{\wedge}^p(st_1, \ldots, st_n)_{(n>1)}$ *with* st_i *having a form given either in 1) or 2).*

Definition 2 (BADSteps) We define an ADTerm of the form $c^p(st, sp)$ as a BADStep, where st is a step only refined with techniques and sp an ADTerm of the form :

1. sp_1, with sp_1 a security pattern,
2. $\wedge^o(sp_1, \ldots, sp_m)$ modelling the conjunction of the security patterns $sp_1, \ldots, sp_m (m > 1)$.

defense$(c^p(st, sp)) =_{def} \{sp_1\}$ iff $sp = sp_1$, or defense$(c^p(st, sp)) =_{def} \{sp_1, \ldots, sp_m\}$ iff $sp = \wedge^o(sp_1, \ldots, sp_m)$.
BADStep(ca) denotes the set of BADSteps of the concrete attack ca.

In Step 4, every attack node A of the initial ADTree T_0 is now automatically replaced with the ADTree $T(A)$. This step is achieved by substituting every term A in the ADTerm $\iota(T_0)$ by $\iota(T(A))$. We denote $\iota(T_f)$ the resulting ADTerm and T_f the final ADTree. It depicts a logical breakdown of the options available to an attacker and the defences, materialised with security patterns, which have to be inserted into the application model and then implemented. During this step, we also build a report from the data-store giving the description of the test architectures needed for executing the attacks on the AUT and observing its reactions.

6 Test Suite Generation and Test Execution

We are now ready to generate test suites and define test verdicts.

ADTrees have various semantics, allowing to carry out different security analyses. We consider here that the semantics of an ADTree is interpreted with attack-defense scenarios over concrete attacks. Intuitively, an attack-defense scenario is a minimal combination of events leading to the root attack, minimal in the sense that, if any event is omitted from the attack-defense scenario, then the root goal will not be achieved. We extract attack-defense scenarios over the set of concrete attacks (not on their BADSteps) to later build test suites that will check whether concrete attacks are effective on the AUT. Informally speaking, as we do not unfold the concrete attacks (which are sequential conjunctions of BADSteps), of an ADTrees T_f, the ADTerm $\iota(Tf)$ only includes conjunctive and disjunctive operators. The set of attack-defense scenarios of T_f can be hence extracted by means of the disjunctive decomposition of $\iota(Tf)$:

Definition 3 (Attack-defense scenarios) Let T_f be an ADTree and $\iota(T_f)$ be its ADTerm. The set of attack-defense scenarios of T_f, denoted $SC(T_f)$ is the set of clauses of the disjunctive normal form of $\iota(T_f)$ over CA(T_f).

An attack-defense scenario s of $Sc(T_f)$ is still an ADTerm. Its satisfiability means that the main goal of the ADTree T_f is feasible by achieving the scenario formulated by s. An attack-defense scenario s of an ADTree T_f is an ADTerm composed of concrete attacks. CA(s) and BADStep(s) denote the set of concrete attacks and BADSteps of s. These sets shall be used to build test suites composed of lists of test cases. We also denote $SP(s)$ and $SP(T_f)$ the security pattern sets found in an attack-defense scenario s and in $\iota(T_f)$.

6.1 Test Suite Generation

Let $s \in SC(Tf)$ be an attack-defense scenario, and ca be one of its concrete attacks. Given a BADStep $b = c^p(st, sp) \in \text{BADStep}(ca)$, Step 5 of the approach assembles, from the data-store, the GWT test case stub $TC(b)$ composed of the following sections:

1. the data-store provides, with the relations $testG$, $testW$ and $testT$, one Given section, one When section and one Then section, each related to one procedure. This Then section aims to assert whether the AUT is vulnerable to the attack step st;
2. for each security pattern sp of defense(b), the data-store provides a set of Then sections associated to procedures. These Then sections aim to check whether the consequences of each security pattern in defense(b) can be detected in the AUT behaviour.

As the BADSteps of a concrete attack ca are sequentially ordered in the ADTree T_f and its ADTerm, we gather the test cases derived from these BADSteps in an ordered list denoted $TS(ca)$. The final test suite TS, derived from an ADTree T_f, is the set of test case lists obtained from the concrete attacks of T_f. This is captured by the following definition:

Definition 4 (Test suites) Let T_f be an ADTree, $s \in SC(Tf)$ and $ca \in \text{CA}(s)$. We denote $TS(ca)$ the test suite of the concrete attack ca. $TS(ca)$ is the ordered list $(T(b_1), \ldots, T(b_n))$ with $b_i (1 \leq i \leq n) \in \text{BADStep}(ca)$.
$TS(s) = \bigcup_{ca \in \text{CA}(s)} TS(ca)$. The final test suite is denoted $TS = \bigcup_{s \in SC(T_f)} TS(s)$.

6.2 Test Case Execution

Step 6 corresponds to the manual completion of the test case procedures by a developer. After this step, we assume that the test cases are correctly developed with assertions in Then sections as stated in Section 3.2: a Then section of a test case $TC(b)$ returns the verdict "$Pass_{st}$" if an attack step st has been successfully applied on the AUT and "$Fail_{st}$" otherwise; a Then section returns "$Fail_{sp}$" if the consequence of the security pattern sp is not detected in the AUT behaviour; when $TC(b)$ returns an unexpected exception or fault, we get the verdict "$Inconclusive_{st}$".

The experimentation of the AUT with the test suite TS is carried out by Step 7. A test case $TC(b)$ of TS, which aims at testing whether the AUT is vulnerable to an attack step st, is composed of more than one Then section. Consequently, its execution, denoted $TC(b)||AUT$, may provide different sets of verdicts, which can be:

- $\{Pass_{st}\}$ means the AUT is vulnerable to the attack step st although all the consequences of the security patterns are detected;
- $\{Fail_{st}\}$ means the AUT does not appear to be vulnerable to the attack step st and that the consequences of the security patterns are detected;

- $\{Pass_{st}, Fail_{sp_1}, ..., Fail_{sp_k}\}$ reflects the fact that the AUT is vulnerable to the attack step st and that the consequences of some security patterns $sp_1, \ldots, sp_k$ are not detected;
- $\{Fail_{st}, Fail_{sp1}, ..., Fail_{sp_k}\}$ means the AUT does not appear to be vulnerable to the attack step st but the consequences of the security patterns $sp_1, \ldots, sp_k$ are not detected;
- $\{Inconclusive_{st}\}$ or $\{Inconclusive_{st}, Fail_{sp1}, ..., Fail_{sp_k}\}$ reflects the fact that some procedures of the test case $TC(b)$ have not been executed due to various problems, e.g., incomplete test architecture, network issues, etc.

Definition 5 (Test verdict sets) Let AUT be an application under test and T_f an ADTree resulting from Step 4. Let also $b = c^p(st, sp) \in \text{BADStep}(ca)$ be a BADStep of a concrete attack $ca \in \text{CA}(T_f)$, with $\text{defense}(b) = \{sp_1, \ldots, sp_m\}_{(m>0)}$. F stands for the power set $\mathcal{P}(\{Fail_{sp_i} \mid sp_i \in \text{defense}(b)\}) \setminus \{\}$.

We define the following test verdict sets:

- $VUL = \{Pass_{st}\}$;
- $NVUL = \{Fail_{st}\}$;
- $VUL/VIOLATE = \{Pass_{st}\} \times F$;
- $NVUL/VIOLATE = \{Fail_{st}\} \times F$;
- $INCONCLUSIVE = \{Inconclusive_{st}\} \cup \{Inconclusive_{st}\} \times F$;

The (local) test verdicts, which can be obtained from a test case, are defined below:

Definition 6 (Local test verdict) $\text{Verdict}(TC(b)||AUT)$ denotes the test verdict given by the test case $TC(b)$. $\text{Verdict}(TC(b)||AUT) = V$ with $V \in \{VUL, NVUL, VUL/VIOLATE, NVUL/VIOLATE, INCONCLUSIVE\}$.

We propose a specific test case model of execution in which a test verdict $\text{Verdict}(TC(b)||AUT)$ is either the result of real test case execution $TC(b)||AUT$ or the result of an implicit deduction. Indeed, Step 5 of the approach generates a test suite $TS(ca)$ for any concrete attack ca found in the ADTree T_f. $TS(ca)$ is an ordered list of test cases $(TC(b_1), \ldots, TC(b_n))$, one test case per step of the attack ca. We can take advantage of these ordered steps to deduce test verdicts. We assume that if the AUT is not vulnerable to a step b_i, then the AUT is not vulnerable to the next step b_{i+1}, as b_i does not provide the requirements, e.g., data or application states, required by b_{i+1}. Several attack bases, e.g., the CAPEC, are structured in this way. More precisely, we postulate in the following proposition that if we have two BADSteps b_i, b_{i+1} with $\text{defense}(b_i) = \text{defense}(b_{i+1})$ such that the verdict of $TC(b_i)||AUT$ expresses that the AUT is not vulnerable, then we set the same test verdict for $TC(b_{i+1})||AUT$. However, if the defenses of the two BADSteps are different, the test case $TC(b_{i+1})$ is even so executed to check whether the consequences of the patterns in $\text{defense}(b_{i+1})$ are detected in the AUT behaviour. We apply a similar reasoning with the test verdict INCONCLUSIVE. If a test case $TC(b_i)$ provides an inconclusive verdict, then we somehow terminate the execution of the test suite $TS(ca)$. It results from these assumptions that the test efforts may be reduced.

Proposition 3 *Let $ca \in \text{CA}(T_f)$ be a concrete attack and $TS(ca) = (TC(b_1), \ldots, TC(b_n))$ be a test suite.*

1. Verdict$(TC(b_i)||AUT)_{(0<i<n)} \in \{NVUL, NVUL/VIOLATE\}$, defense$(b_i)$ = defense$(b_{i+1}) \implies$ Verdict$(TC(b_{i+1})||AUT)$ = Verdict$(TC(b_i)||AUT)$;
2. Verdict$(TC(b_i)||AUT)_{(0<i<n)} = INCONCLUSIVE \implies$ Verdict$(TC(b_{i+1})$ $||AUT) = INCONCLUSIVE$.

Subsequently, we define final test verdicts with regard to the ADTree T_f. These verdicts are given with the predicates Vulnerable(T_f), Unsat$^c(SP(T_f))$ and Inconclusive(T_f) returning boolean values.

The predicate Vulnerable(ca) is firstly defined on a concrete attack ca to later apply a substitution σ : CA$(s) \rightarrow \{true, false\}$ on an attack-defense scenario s. This substitution is used to evaluate attack-defense scenarios. A scenario s holds if the evaluation of applying the substitution σ to s (i.e. replacing every concrete attack term ca with the evaluation of Vulnerable(ca)) returns true. When a scenario of T_f holds, then the threat modelled by T_f can be achieved on the AUT. This is defined with the predicate Vulnerable(T_f). Unsat$^c(SP(T_f))$ denotes whether the security pattern consequences are detected in the AUT behaviour.

Definition 7 (Final test verdicts) Let AUT be an application under test, T_f be an ADTree, $s \in SC(T_f)$ and $ca \in$ CA(s).

1. Vulnerable(ca) $=_{def}$ $true$ if $\exists b \in$ BADStep(ca) : Verdict$(TC(b)||AUT) \in \{VUL, VUL/VIOLATE\}$; otherwise, Vulnerable$(ca)$ $=_{def}$ $false$;
2. σ : CA$(s) \rightarrow \{true, false\}$ is a substitution $\{ca_1 \rightarrow$ (Vulnerable$(ca_1), \ldots,$ $ca_n \rightarrow$ Vulnerable$(ca_n)\}$;
3. Vulnerable(T_f) $=_{def}$ $true$ if $\exists s \in SC(T_f)$: $eval(s\sigma)$ returns true ; otherwise, Vulnerable(T_f) $=_{def}$ $false$;
4. Unsat$^c(SP(T_f))$ $=_{def}$ $true$ if $\exists s \in SC(T_f), b \in$ BADStep(s): Verdict$(TC(b)||$ $AUT) \in \{VUL/VIOLATE, NVUL/VIOLATE\}$; otherwise, Unsat$^c(SP(T_f))$ $=_{def}$ $false$;
5. Inconclusive(T_f) $=_{def}$ $true$ if $\exists s \in SC(T_f), b \in$ BADStep(s): Verdict$(TC(b)||$ $AUT) = INCONCLUSIVE$; otherwise, Inconclusive(T_f) $=_{def}$ $false$.

6.3 Limitations

Our approach suffers from several limitations. We provide some of them below, which could lead to future work:

- the current data-store is not exhaustive: it includes 215 attacks out of 569 (for any kind of application) and 26 security patterns out of 176. It can be completed with new attacks of the CAPEC base automatically. But the completion of the data-store with new security patterns or test architectures is manually done at the moment;
- several steps of the approach require manual interventions that are prone to errors, e.g., the security pattern choice (Step 3) or the completion of the test case procedures (Step 6). To better guide the writing of concrete test cases, the data-store could integrate more data coming from other bases, e.g., Exploit-db (Offensive Security, 2017), which is a well-known repository of exploits for security experts;

- the ADTree size is not taken into account by our approach, yet, when an attack has a high level of abstraction, we observed that the ADTree size may be large because it includes a set of sub-attacks themselves linked to several patterns. The generation of large ADTrees is a strong limitation of our approach since these ADTrees become unreadable, which contradicts our objectives;
- our approach checks neither the behavioural nor the structural properties of security patterns, as it is performed in (Yoshizawa et al, 2014). These properties are yet important to ensure that security patterns are well integrated in a model and implemented. Both approaches could be combined. But, we believe that the use of OCL expressions to model pattern properties, (as proposed in (Yoshizawa et al, 2014)), will strongly affect Comprehensibility;
- it is well-known that the implementation of countermeasures sometimes causes the introduction of new vulnerabilities. The same problem could occurs with security patterns. This kind of side effect is not addressed in this paper.

7 Evaluation

We empirically studied two scenarios on 24 participants to assess whether developers can benefit of our approach. The duration of each scenario was set to at most to one hour and half. The participants are third to fourth year computer science undergraduate students, having good skills in the development and test of Web applications. They have good knowledge about classical attacks and are used to handle design patterns, but not security patterns.

The participants were given the task of choosing security pattern combinations to prevent two attacks, CAPEC 244: Cross-Site Scripting via Encoded URI Schemes, and CAPEC 66: SQL Injection, on two deliberately vulnerable Web sites, *RopeyTasks* and *The Bodgeit Store*. We also asked the participants to write test cases with the tool Selenium in order to: report that both Web sites are vulnerable to the attacks, show that the application behaviours do not include at least one consequence of the security patterns "Input Guard" and "Output Guard".

In the first scenario, denoted Part 1, we supplied these documents to the students: the CAPEC base, two concrete examples detailing how to manually perform each attack along with the expected outcomes, and the security pattern catalogue published with Yskout et al (2015), composed of 36 patterns. The participants had to: read the intents and consequences of the patterns, follow our examples of attacks and read the CAPEC base to write concrete test cases. In the second scenario, denoted Part 2, we supplied additional documents for the two attacks: the ADTrees of the two attacks (given in Step 2 of the approach) and the generated *GWT* test case stubs (Step 5). Among the 28 Generic procedures we provided, 8 were composed of blocks of code and required few modifications. The others included comments extracted from the CAPEC base and the security pattern catalogue. At the end of each scenario, the students were invited to fill in a form listing these questions:

- Q1: Was it difficult to choose security patterns?
- Q2: Was it difficult to use the CAPEC documentation (in Part 1) / our ADTrees (in Part 2)?
- Q3: Was it difficult to use the security pattern documents (in Part 1) / our ADTrees (in Part 2)?

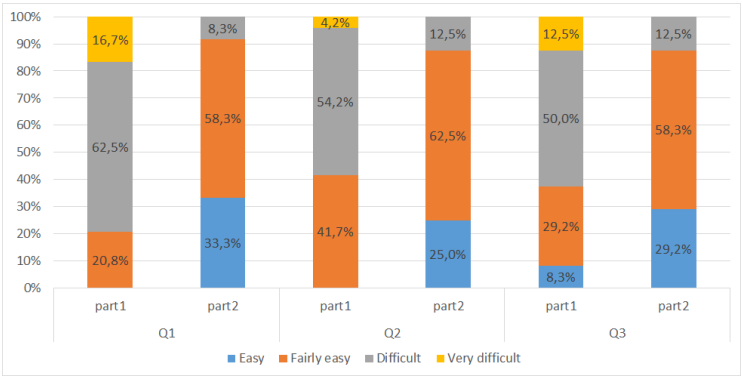


Fig. 11 Response rates for Q1 to Q3

– Q4: Was it easy to write test cases?
– Q5: How long did you take for writing test cases?
– Q6: How confident are you about your test cases?
– Q7: Provide your test cases (or suites).

With these questions, we evaluated our approach using three criteria:

– C1: Comprehensibility: does our method ease the choice of the security patterns and the test case development?
– C2: Effectiveness: can the test cases detect defects?
– C3: Efficiency: does our method help reduce the time needed for writing tests?

7.1 Experiment Results

From the answers returned by the participants (available in (Regainia and Salva, 2017)), we extracted the following results. Firstly, Figure 11 illustrates the percentages of answers to the questions Q1 to Q3. For these, we proposed this four-valued scale: easy, fairly easy, difficult, very difficult. Similarly, we collected the answers of Question Q4 (again on this four-valued scale). Figure 12 (left side) depicts the distribution of the participant opinions.

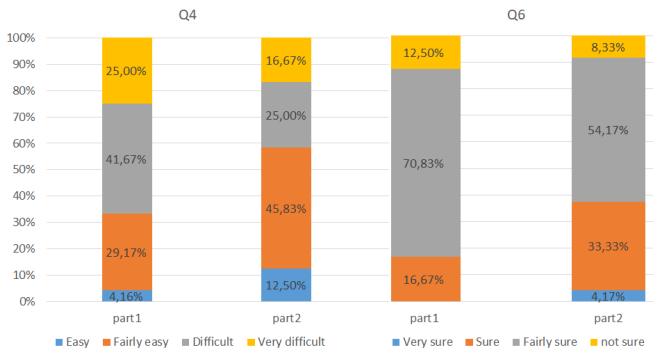


Fig. 12 Response rates for Questions Q4 and Q6

We measured the time required for the participants to write test cases. They consumed between 15 and 70 minutes in Part 1, while they took between 20 minutes and 86 minutes in Part 2. On average, they spent 46 minutes in Part 1 and 60 minutes in Part 2. The levels of confidence of the participants are estimated with Question Q6. The possible answers were for both scenarios: *very sure, sure, fairly sure, not sure*. Figure 12 (right side) depicts the percentages of answers.

We finally analysed the test cases and evaluated their correctness with regard to four aspects: 1&2: detection (with at least one test case) that both applications are vulnerable to the attacks CAPEC 66 and CAPEC 244; 3&4: detection that the application behaviours do not include the consequences of the patterns "Input Guard" and "Output Guard". As we considered this last aspect as difficult for students, we expected at least one Then test case section for every pattern. Figure 13 presents the number of participants who meet these aspects.

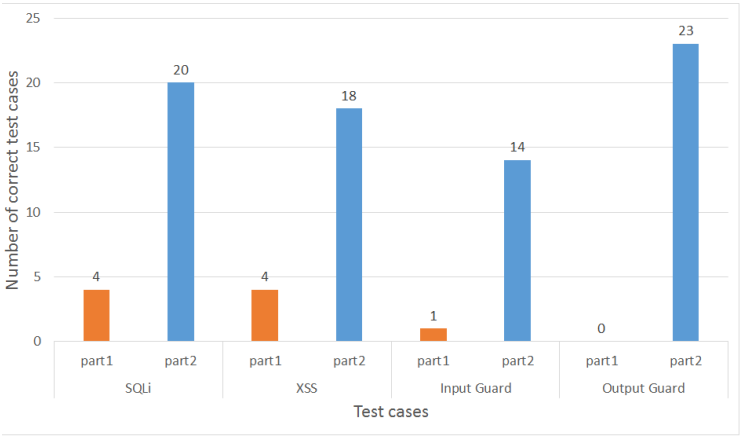


Fig. 13 Test case correctness (Question Q7)

7.2 Result Interpretation

C1: Comprehensibility

Figure 11 shows that 33% of the participants estimated that the pattern choice was easier with our ADTrees (Q1). In contrast, no participant found that the choice was easy when using only the security pattern catalogue. The rate of "Easy" "Fairly Easy" increased by 70,8% between Part 1 and Part 2. With Question Q2, 41,7% of the participants found "Fairly easy" the use of the CAPEC base, whereas 87,5% esteemed our ADTrees "Easy" and "Fairly Easy" to use. Similarly, only 37,5% of the participants considered "Easy" and "Fairly easy" the reading of the pattern catalogue. This rate reaches 87,5% with our ADTrees. Consequently, Figure 11 shows that our generated ADTrees make the pattern choice easier and that they are simpler to interpret than the security pattern catalogue.

Concerning test cases, Figure 12 shows that 66,7% of the participants found the test case writing difficult in Part 1 against 41,7% in the second part. In contrast, 4,17% of them found the test case development *"very easy"* in Part 1 against 12,5%

in Part 2. In addition, one quarter of the students found the test writing easier with our test case stubs. After discussion with the participants, it turned out that the link of the test case sections with the attack steps of the ADTrees helped them understand what to develop. Figure 12 shows that the average confidence level of the participants about the accuracy of their test cases (Question Q6) increased by 20,83%.

As a whole, we conclude that the participants found their tasks easier with our approach since the ADTrees and GWT test case stubs offer better readability.

C2: Effectiveness

Figure 13 depicts the results about the test case correctness. The columns "SQLi" and "XSS" provide the number of test cases allowing to reveal that both attacks can be successfully executed on the applications. In Part 1, few participants developed complete test cases. We indeed observed assertions were missing in most of them. If we leave aside the assertions, the number of test cases running the attacks rises to 14. The number of correct test cases strongly increases in Part 2, by means of the comments found in the procedures, even though these texts were also available in the CAPEC base given in Part 1. The columns "Input Guard" and "Output Guard" give the number of Then sections (and procedures) allowing to report that the consequences of these security patterns are not observed from the application behaviours. This task was much more difficult for the participants as they are not yet expert in security pattern. Hence, it is not surprising to see that only one participant was able to write at least an assertion allowing to show that an Input Guard consequence is not detected. The number of correct Then sections rises to 14 in Part 2 by means of the comments found in the procedures, which were derived from the security pattern descriptions also given in Part 1. With the pattern "Output guard", the number of correct Then sections rises from 0 to 23 (almost all the participants) in Part 2.

Consequently, we conclude that the test case correctness strongly increases with our approach, thanks to the amount of information (comments, blocks of code) found in the procedures.

C3: Efficiency

This criterion addresses the participant efficiency in terms of time spent for writing test cases. On average, the participants took 46 minutes for writing them from scratch and 60 minutes with the use of our method. We have to admit that we expected a different outcome. But, the additional time spent in Part 2 can be explained when we alongside focus on Effectiveness and Comprehensibility. Indeed, after discussion with the participants, we concluded that they spent more time for interpreting the ADTrees or reading the comments in the procedures. This supplement of information and this extra time made the participants more effective. In the long run, we believe that the extra time spent when using our approach should decrease. The interpretation of attacks with ADTrees should educate the users, they should also be more confident on developing GWT test cases after a while.

7.3 Threat to Validity

This preliminary experimental evaluation is applied on Web applications only. This is a threat to external validity, in the sense that the results about Comprehensibility and Accuracy cannot be generalised to all software systems. This is why we avoid drawing any general conclusion. But, we believe that this threat is somewhat mitigated by our choice of application, as the Web application context is a rich field in great demand in the software industry. Furthermore, this well-studied application context helped us propose experimentations involving participants having the adequate knowledge on testing and security.

This leads to the second threat to validity concerning the public. Our evaluation was performed on students following a block release training. This sort of audience is sometimes considered as a bias, as any strict process should help them improve their work. But, we do not think that students bring a bias in security technique evaluations. This is indeed confirmed in several studies, e.g. (Daun et al, 2017). We believe that we would have achieved the same results with a group of developers from the industry, as they often do not have any software security skills.

Another threat relates to the learning effect, which may happens between the two stages of the evaluation: we applied the same approach to the same participants. We only replaced the Web application in Part 2 (to avoid another bias, we took care to provide an application providing similar functionalities and exposing the same vulnerabilities), and we completed the available security documents by a list of ADTrees. We deliberately chose the apply the same approach in Part 1 and 2 to evaluate the difficulties encountered by the participants (Q1-4) and their confidence on their work (Q6). It is indisputable that this kind of experiment may influence our results. But, we believe that the amount of improvement revealed in Part 2 cannot only be explained by a learning effect.

Finally, we provide more documents (ADTrees, test cases) in Part 2. This makes the tasks easier but this might make developers less efficient as there is more reading required. As stated before, we believe that the extra time required when using our approach should decrease in the long run. This is why we avoid drawing any conclusion on Efficiency in this paper.

8 Conclusion

Today's developers have to know how to develop secure software and how to test it. These two tasks are particularly time-consuming and difficult since a lot of expertise is required. To help them in these tasks, we have proposed an approach based on the notion of knowledge base. This paper brings two main contributions. It helps developers in the writing of concrete security test cases and ADTrees. It also checks whether an application is vulnerable to attack-defense scenarios and whether security pattern consequences are detected in the application behaviour. We conducted an evaluation on 24 participants, which suggests that our approach: a) eases the security pattern choice and test case development; b) makes them more effective on security testing.

We also mentioned several limitations, which lead to some immediate lines of future work. Among them, we firstly plan to focus on the size of the ADTrees generated by the threat modelling stage. Indeed, the larger the ADTrees, the

more difficult are their interpretations, which enters in contradiction with the basic objectives of this work. An ADTree reduction (or minimisation) could be a first solution on this problem. But, reducing such trees remains a hard problem as the node meaning must be taken into account in the node aggregating process. We will also focus on the generation of test cases for checking whether security pattern behavioural or structural properties hold in application behaviours. One of the problems here is to assist developers, who are usually not expert in formal modelling, to express these properties.

References

Alvi K Aleem, Zulkernine M (2012) A Comparative Study of Software Security Pattern Classifications. 2012 Seventh International Conference on Availability, Reliability and Security pp 582–589

Bozic J, Simos DE, Wotawa F (2014) Attack pattern-based combinatorial testing. In: Proceedings of the 9th International Workshop on Automation of Software Test, ACM, New York, NY, USA, AST 2014, pp 1–7

Daun M, Hübscher C, Weyer T (2017) Controlled experiments with student participants in software engineering: Preliminary results from a systematic mapping study. CoRR abs/1708.04662, 1708.04662

Dong J, Peng T, Zhao Y (2010) Automated verification of security pattern compositions. Inf Softw Technol 52(3):274–295

El Ariss O, Xu D (2011) Modeling security attacks with statecharts. In: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ISARCS, ACM, New York, NY, USA, QoSA-ISARCS '11, pp 123–132

Fernandez EB, Washizaki H, Yoshioka N, Kubo A, Fukazawa Y (2008) Classifying security patterns. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol 4976 LNCS, pp 342–347

Hamid B, Percebois C, Gouteux D (2012) A methodology for integration of patterns with validation purpose. In: Proceedings of the 17th European Conference on Pattern Languages of Programs, ACM, New York, NY, USA, EuroPLoP '12, pp 8:1–8:14

Howard M, LeBlanc D (2003) Writing Secure Code. Microsoft Press

Jhawar R, Kordy B, Mauw S, Radomirović S, Trujillo-Rasua R (2015) Attack trees with sequential conjunction. In: IFIP International Information Security Conference, Springer, pp 339–353

Kobashi T, Yoshizawa M, Washizaki H, Fukazawa Y, Yoshioka N, Okubo T, Kaiya H (2015) Tesem: A tool for verifying security design pattern applications by model testing. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp 1–8

Kordy B, Mauw S, Radomirović S, Schweitzer P (2012) Attack–defense trees. Journal of Logic and Computation p exs029

Kordy B, Kordy P, Mauw S, Schweitzer P (2013) Adtool: security analysis with attack–defense trees. In: International Conference on Quantitative Evaluation of Systems, Springer, pp 173–176

Marback A, Do H, He K, Kondamarri S, Xu D (2009) Security test generation using threat trees. In: 2009 ICSE Workshop on Automation of Software Test, pp 62–69

Marback A, Do H, He K, Kondamarri S, Xu D (2013) A threat model-based approach to security testing. Softw Pract Exper 43(2):241–258

Mitre corporation (2015) Common attack pattern enumeration and classification. URL `https://capec.mitre.org/`

Morais A, Martins E, Cavalli A, Jimenez W (2009) Security protocol testing using attack trees. In: 2009 International Conference on Computational Science and Engineering, vol 2, pp 690–697

Offensive Security (2017) Exploit database archive. URL `https://www.exploit-db.com/`

OWASP (2016) Owasp testing guide v3.0 project. URL `http://www.owasp.org/index.php/Category:OWASP\_Testing\_Project\#OWASP\_Testing\_Guide`

del Pilar Salas-Zárate M, Alor-Hernández G, Valencia-García R, Rodríguez-Mazahua L, Rodríguez-González A, Cuadrado JLL (2015) Analyzing best practices on web development frameworks: The lift approach. Science of Computer Programming 102:1–19

Regaigna L, Bouhours C, Salva S (2016) A systematic approach to assist designers in security pattern integration. In: International Conference on Advances and Trends in Software Engineering (SOFTENG 2016), Lisbon, Portugal

Regainia L, Salva S (2017) Security pattern classification, companion site. URL `http://regainia.com/research/companion.html`

Rojas JM, Fraser G, Arcuri A (2015) Automated unit test generation during software development: A controlled experiment and think-aloud observations. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2015, pp 338–349

Salva S, Regainia L (2017a) Using data integration for security testing. In: Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings, pp 178–194

Salva S, Regainia L (2017b) Using data integration to help design more secure applications. In: Proceedings of the 12th International Conference on Risks and Security of Internet and Systems, Springer-Verlag, Dinard, France

Schieferdecker I, Grossmann J, Schneider MA (2012) Model-based security testing. In: Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012., pp 1–12

Schumacher M (2003) Security Engineering with Patterns: Origins, Theoretical Models, and New Applications. Springer-Verlag New York, Inc.

Shahmehri N, Mammar A, Montes De Oca E, Byers D, Cavalli A, Ardi S, Jimenez W (2012) An advanced approach for modeling and detecting software vulnerabilities. Inf Softw Technol 54(9):997–1013

Slavin R, Niu J (2017) Security patterns repository. URL `http://sefm.cs.utsa.edu/repository/`

Torr P (2005) Demystifying the threat modeling process. IEEE Security Privacy 3(5):66–70

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

Yoder J, Yoder J, Barcalow J, Barcalow J (1998) Architectural patterns for enabling application security. Proceedings of PLoP 1997 51:31

Yoshizawa M, Kobashi T, Washizaki H, Fukazawa Y, Okubo T, Kaiya H, Yoshioka N (2014) Verifying implementation of security design patterns using a test template. In: 2014 Ninth International Conference on Availability, Reliability and Security, pp 178–183

Yskout K, Heyman T, Scandariato R, Joosen W (2006) A system of security patterns, technical report cw-469

Yskout K, Scandariato R, Joosen W (2015) Do security patterns really help designers? In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 292–302