Using Data Integration for Security Testing

Sébastien Salva[1] and Loukmen Regainia[2]

[1] LIMOS CNRS UMR 6158, Clermont Auvergne University,
sebastien.salva@uca.fr
[2] LIMOS CNRS UMR 6158, Clermont Auvergne University,
loukmen.regainia@uca.fr

Abstract. The explosion of digitisation makes a plethora of security data publicly available for developers. These numerous (often complex) documents expose them to the difficulty of choosing the most appropriate solution for securing their applications. We propose in this paper a method based upon data acquisition and integration, which assists developers in the Threat modelling stage and in the security test case execution. The method firstly helps devise Attack Defense Trees by means of a data-store. These trees show attacks, steps and defenses given under the form of security patterns, which are re-usable solutions to design more secure applications. These trees are then used for the test case generation. The data-store integrates test case stubs, which make this generation easier and developers more efficient. We evaluate our approach on 24 participants and show encouraging results on the use of data integration in software engineering.

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

1 Introduction

Since a decade, it is well admitted that software security is essential and has to be considered through all the software life cycle. Many developers, researchers and organisations have hence made security their hobby-horse and brought several improvements with the proposal of numerous digitised security bases and documents. These take security into consideration at different stages of the software life cycle and are presented with different viewpoints, abstraction levels or contexts. This plethora of diverse documents makes difficult the choices of security solutions and afterwards their validations. Indeed, developers cannot be experts in any field and they clearly lack guidance for conceiving and implementing both secure software and tests.

This work focuses on this issue and studies the possibility of using publicly available security resources for helping developers devise more secure applications. We propose a method, which aims at assisting developers in the Threat modelling stage and in the test case generation and execution. More precisely, the contributions of this paper are highlighted in the following points:

- we present a security data acquisition and integration method, which extracts data from various Web and publicly accessible sources to conceive a data-store storing relationships among attacks, security principles, security patterns and test case parts written with the Given When Then (GWT for short) template. *The security pattern intuitively relates countermeasures to threats and attacks in a given context* [8]. These security re-usable solutions often are presented textually or with UML schema and are characterised by a set of structural and behavioural properties;
- from the data-store, our method helps in the generation of Attack Defense Trees (ADTrees [3]) showing the attacker possibilities to compromise an application and the defenses that may be put in place to prevent attacks. We have chosen the ADTree model because it offers the advantage of being easy to understand even for novices in security. These ADTrees are composed of defenses given under the form of security pattern combinations;
- ADTrees serve here to the test case generation. These test cases aim to check whether an application is vulnerable against the attacks exposed in an ADTree and whether security pattern consequences are observed in the application behaviour. Pattern consequences are observable events resulting from the good contextualisation and implementation of the pattern in an application. From an ADTree, our method extracts attack scenarios, GWT test case stubs and related procedures composed of comments or blocks of code, which aim at guiding the developer in the test case completion. As ADTrees can be formalised with formal expressions, called ADTerms, we strictly define the test case generation and execution.

Besides, we concentrated our attention on quality criteria and on education while the design of this method. ADTrees are indeed constructed with concrete data extracted from the CAPEC base[3]. ADTrees also express security pattern combinations with respect to several criteria, i.e., Unambiguity, Navigability and Comprehensibility, which are quality criteria proposed in [1], the last two 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*. Test case stubs are also structured to ease Readability and Re-usability and to try to increase Effectiveness.

We have generated a data-store specialised to the context of Web applications (Web sites), which is composed of 215 CAPEC attacks, 26 security patterns and 669 test case parts. We employed this data-store to evaluate on 24 human subjects the benefits of using the notion of data acquisition and integration in the software life cycle. This evaluation shows encouraging results about Comprehensibility and Effectiveness.

The remainder of the paper is organised as follows: Section 2 presents some related work. The data integration step is shortly presented in Section 3. The next section shows how ADTrees are generated by means of the data-store. Section 5 addresses the test case generation and execution. We present our evaluation in Section 6 and finally conclude in Section 7.

[3] `https://capec.mitre.org/`

2 Related Work

A plethora of papers deals with model-based security testing. Due to lack of room, we only present some of them related to our work, which consider models not to describe the implementation behaviour but rather to express the attacker's goals or the vulnerability causes of the system [6, 5, 4, 9]. Some authors focused on trees (Attack trees, Security Activity Graphs, etc.), which express the treats or attacks or vulnerability causes that should be prevented in systems. From these models, test cases are then written to check whether attacks can be successfully executed. Morais et al. introduced a security testing approach specialised for protocols [6]. 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 [5], 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 allowing the generation of concrete values. These events are then replaced with blocks of code to produce test cases. In [4], test cases are generated from Threat trees. The latter are completed with parameters associated to regular expressions. Security scenarios are extracted from the Threat trees and are manually converted to executable test scripts. Shahmehri et al. proposed a passive testing approach to detect vulnerabilities [9]. The undesired vulnerabilities are modelled with models called SGMs, which are specialised DAGs showing security goals, vulnerabilities and eventually mitigations. Detection conditions are then semi-automatically extracted and given to a monitoring tool, which returns test verdicts.

These works take Threat models as inputs, which are manually written. If these lack of details (parameters, attack steps, etc.), the final test cases will be too abstract as well. Furthermore, these methods do not give any recommendation on how to write tests and on how to structure them. Hence, developers lack guidance to write tests and to reuse them. This paper proposes a method, which semi-automatically infer Attack Defense Trees, composed of attacks steps, techniques and defenses. To ease the understanding and readability of the generated test case stubs, we associate in our data-store every attack and defense step to some test case sections, which are classified w.r.t. an application context and to an attack step or security pattern.

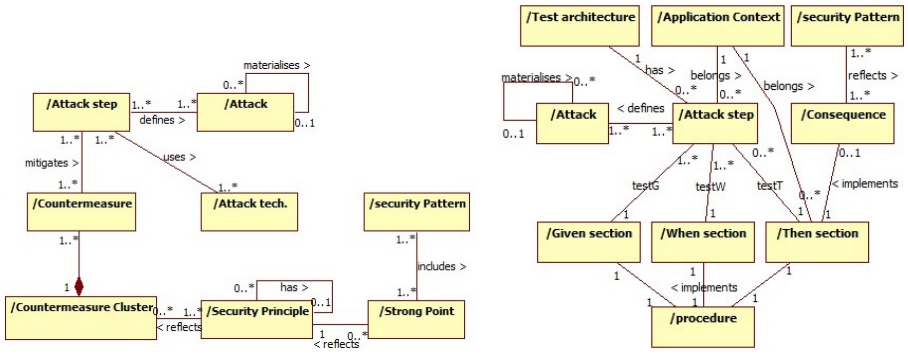
Few works tackled the testing of security patterns, which is another topic of this paper. Yoshizawa et al. introduced a method for testing whether behavioural and structural properties of patterns can be observed in the traces of instrumented applications [10]. Given a security pattern, two test templates (OCL expressions) are written, one to specify the pattern structure and another to describe its behaviour. Then, developers have to make templates concrete by writing Selenium scripts for experimenting the application. The latter returns traces on which the OCL expressions are verified. In contrast to the previous paper, our inferred ADTrees firstly help developers choose for every attack, the combinations of patterns that can be used as countermeasures. Then, our testing approach aims at testing the security pattern consequences. We do not check the

structure of the patterns. Hence, our approach is complementary to the previous one.

We also proposed a semi-automatic data integration method in [7], in order to extract a pattern classification. We took inspiration from this paper to infer pattern combinations. In contrast, the notion of data-store, its architecture, the considered security properties and the test case generation and execution are new contributions.

3 Data Integration

3.1 Data-store architecture presentation



(a) Data-store meta-model (b) Data-store extension

Fig. 1.

Figure 1(a) exposes the meta-model of the first part of the data-store used to integrate relationships among attacks, attack steps, techniques, security principles and security patterns. The entities of Figure 1(a) refer to these security properties. To increase the precision of the relations, we chose to decompose attacks into sub-attacks, and into attack steps. These steps are associated to countermeasures, allowing to prevent or counter attack steps. We also decompose security patterns into strong points, which are sub-properties expressing pattern key design features. Relying on a hierarchical organisation of security principles, the method maps countermeasure clusters to principles and strong points to principles. As countermeasures usually are detailed properties, we gather them into clusters (groups) to reach about the same abstraction levels as those of the security principles.

The meta-model of Figure 1(a) is extended with new entities and relations, which are required for the testing process. This extension is depicted in Figure 1(b). An attack step is also associated to a Test architecture and to one Application context. The context refers to an application family, e.g., Android applications, or Web sites. The "Test Architecture" entity refers to textual paragraphs explaining the points of observation and control, testers or tools required to execute the attack step on an application, which belongs to an Application

context. Next, we map attack steps onto GWT test case sections. For readability and re-usability purposes, we chose to consider the "Given When Then" pattern to break up test cases into several parts:

- the Given section aims at putting an application under test in a known state;
- the When section triggers some actions;
- the Then section is used to check whether the conditions of success of the test case are meet (assertions). In our context, we suppose that a Then section returns "$Pass_{st}$" if an attack step st has been successfully executed on an implementation and "$Fail_{st}$" otherwise.

Likewise, we map security pattern consequences onto Then sections to check whether the consequences of the pattern can be observed in the application traces. We assume that a Then section returns "$Fail_{sp}$" if a consequence of the security pattern sp is not observed from the implementation. For instance, if an application is conceived with the "Input Guard" pattern, then unexpected inputs should bring the application to a quiescent state (no output) or outputs reflecting errors should be observed.

Each test case section is linked to one procedure stored in the Procedure table of the data-store, which implements the section. A Given, When or Then section can be reused with several attack steps or security patterns. With regard to the meta-model given in Figure 1(b), a GWT test case section (and procedure) is classified according to one application context and one attack step or pattern consequence.

In some specific application contexts, the procedures can be completed with comments or with blocks of code to ease the test case development. When the procedure content can be reused with any application in a precise context, we call it *Generic procedure*:

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 C;*

The data-store must only contain Generic procedures related to an application context. It worth mentioning that an empty procedure is generic.

3.2 Security data acquisition and integration

This section summarises a data integration example for the Web application (Web sites) context. We chose to focus on the CAPEC base to extract information about security attacks. The CAPEC base offers an open and rich catalogue of attacks in a comprehensive schema. We conceived a tool for data acquisition and extraction, based on text mining and on the ELT (Extraction, Load, Transform) tool Talend [4]. With it, we automatically scanned all the CAPEC base (Version 2.8) and collected 215 attacks, 209 steps, 448 techniques and 217

[4] https://talend.com/

countermeasures, knowing that attacks can share steps, attack techniques and countermeasures. Among these, we gathered 75 attacks and 142 attack steps specialised for the context of Web sites.

As security patterns are described in an abstract manner with texts, we manually collected security patterns, their strong points and consequences from the catalogue given in [12]. We gathered 26 security patterns, 43 consequences and 36 strong points. We also integrated the inter-pattern relations given in [11]. We organised 66 security principles found in the literature into a hierarchy composed of four levels, from the most abstract to the most concrete principles.

The data integration of the GWT test case sections was automatically performed. For a given attack, the CAPEC base provides two text sections called "Attack Prerequisites" and "Resources Required". We automatically scanned these paragraphs and completed 209 procedures including comments composed of the two previous paragraphs. Each procedure is associated to one Given test case section (one section for each attack step). For every step *st*, we added one When test case section and one procedure composed of comments listing the techniques related to *st*. Still in the CAPEC documents, the paragraphs "Indicators" and "Outcomes" provide some directives and conditions on an attack step realisation. In the same way as previously, we automatically scanned these paragraphs and, for every attack step, we completed the data-store with one Then section associated to one procedure, itself composed of the two previous paragraphs of the CAPEC base given as comments. In this way, we generated 627 GWT test case sections. For every security pattern consequence found in the data-store, a Then test case section and its related procedure are also automatically inserted. The procedure is composed of comments listing the consequence, which have to be observed from the application traces. In the context of Web applications, we observed that several procedures can be completed with blocks of code calling penetration testing tools. We completed 32 procedures, which cover 43 attack steps. We used the tools Selenium and ZAProxy[5], which is a penetration testing tool covering various Web vulnerabilities.

This data-store is available here[6].

4 Threat Modelling

Before presenting how our method assists developers in threat modelling, we recall some notions about the ADTree model.

4.1 Attack Defense Trees (ADTrees)

ADTrees *are graphical representations of possible measures an attacker might take in order to attack a system and the defenses that a defender can employ to protect the system* [3]. As illustrated in Figures 3(a) and 3(b), ADTrees have

[5] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[6] `http://regainia.com/research/companion.html`

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 (like in Figure 3(a)) 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 [2]. This operator expresses the execution order 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 node "Attack Step" in Figure 3(b) is refined with a sequence of others steps. Alternatively, 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).

4.2 Attack Defense Tree generation

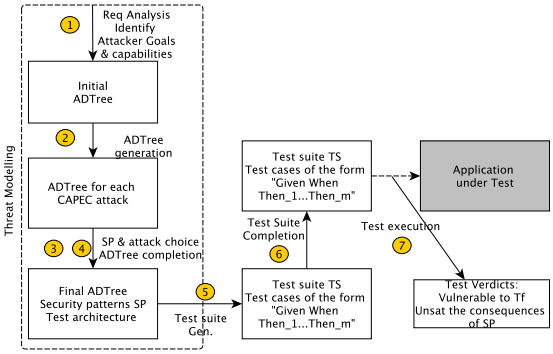


Fig. 2. Threat modelling and security testing steps

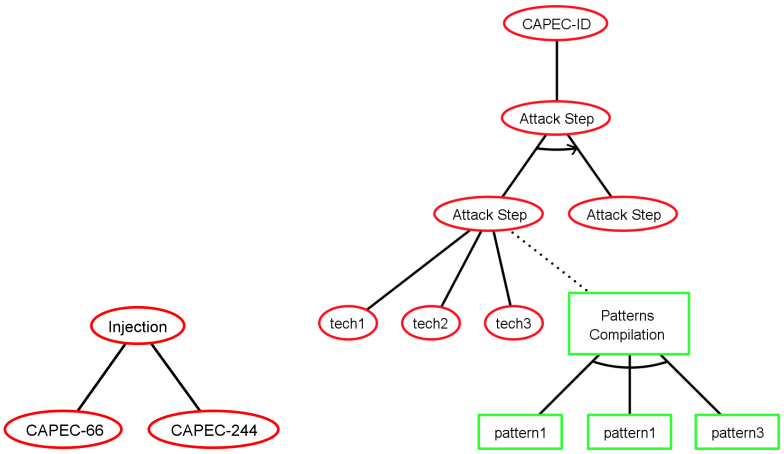
The first stage of our method takes place in the Threat modelling phase of the software life cycle, which occurs while the requirement analysis. Threat modelling is a process consisting in identifying and describing the attacker's goals and capabilities, as well as identifying the potential threats of an application. Different methods can be followed, e.g., DREAD, or STRIDE. Our method starts to semi-automatically generate an ADTree by means of the data-store. The ADTree generation is illustrated in the the fourth first steps of Figure 2. We present them below.

Step 1: Initial ADTree design

The developer initially establishes a first ADTree T whose root node represents the attacker's goal, which may be refined with child nodes. The ADTree T

describes attack combinations, which can be applied on the application. We here assume that T at least has leaves labelled by attacks kept in the data-store. Otherwise, a semantic alignment may be required to replace some labels by similar attack names.

Figure 3(a) depicts an ADTree example: the goal, given in the root node, is to inject malicious code into an application. This node is disjunctively refined with two children expressing two more concrete attacks, CAPEC-66: SQL Injection and CAPEC-244: Cross-Site Scripting via Encoded URI Schemes.



(a) Initial ADTree example (b) ADTree general form

Fig. 3.

Step 2: ADTree generation

Usually, yhe ADTree T does not include enough details on how the attack is sequenced and on the defenses expressing how the application can be protected. Implementing a secure application and deriving test cases from this kind of tree remains a tedious task. The data-store can be used to augment T. For every node labelled with an attack Att, we automatically generate an ADTree denoted $T(Att)$. The architecture of the data-store leads to the generation of ADTrees having the general form illustrated in Figure 3(b). The root of an ADTree $T(Att)$ is labelled by Att. This node may have children expressing more concrete attacks and so forth. The most concrete attacks have step sequences (edges connected with an arrow). These steps are connected to techniques with a disjunctive refinement. The lowest attack steps in the ADTree are also linked to defense nodes, which may be the roots of sub-trees expressing security pattern combinations whose purpose is to counteract the attack step.

We implemented the ADTree generation with a tool, which takes attacks of the data-store and yields XML files. These can be edited with the tool *ADTool* [3]. For instance, Figure 4 depicts the ADTree of the attack CAPEC-66, which was exported from ADTool. Each lowest attack step has a defense node express-

ing pattern combinations. Step 2.1, which identifies the possibilities to inject malicious code through the application inputs, requires more patterns than the other steps to filter these inputs. Some of them have relations: for instance "Application Firewall" can be replaced by "Input Guard" with "Output Guard".

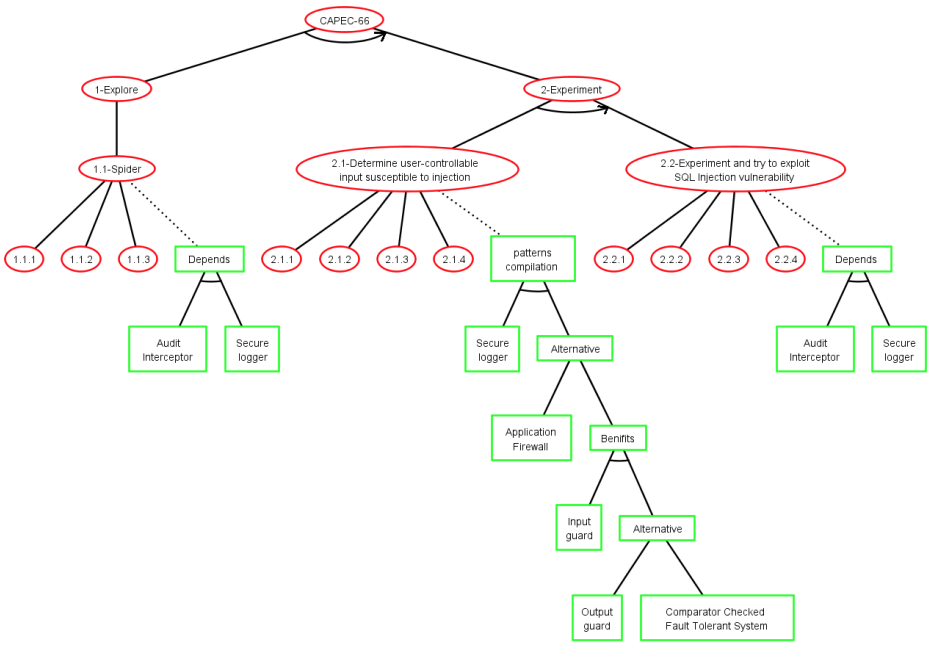


Fig. 4. ADTree of the Attack CAPEC-66

Step 3: Security pattern choice

The developer can now edit the ADTrees $T(Att)$ to keep or remove attack steps w.r.t. the application context. He or she also has to choose the security patterns that have to be contextualised and implemented in the application. After this step, we assume that a defense node either is labelled by a security pattern (it does not have children) or has a conjunctive refinement of nodes labelled by security patterns. The lowest nodes labelled by attack steps, must be linked to a defense node.

As a result of the steps 2 and 3, the generated ADTrees have specific forms and are expressed with specific ADTerms:

Proposition 1. *An ADTree $T(Att)$ achieved by the previous steps has an ADTerm $\iota(T(Att))$ having one of these forms:*

1. $\vee^p(t_1, \ldots, t_n)$ *with* $t_i(1 \leq i \leq n)$ *an ADTerm also having one of these forms:*
2. $\overrightarrow{\wedge}^p(t_1, \ldots, t_n)$ *with* $t_i(1 \leq i \leq n)$ *an ADTerm having the form given in 2) or 3);*
3. $c^p(st, sp)$*, with st an ADTerm expressing an attack step and sp an ADTerm modelling a security pattern combination.*

The first ADTerm expresses children nodes labelled by more concrete attacks. The second one represents sequences of attack steps. The last expression is composed of an attack step st refined with techniques, which can be counteracted by a security pattern combination sp. In the remainder of the paper, we call the last expression *Basic Attack Defence Step*, shortened as BADStep. These shall be particularly useful to build GWT test case stubs:

Definition 2 (Basic Attack Defence Step (BADStep)). *A BADStep b is an ADTerm of the form $c^p(st, sp)$, where st is an ADTerm modelling an attack step and sp an ADTerm of the form sp_1 or $\wedge^o(sp_1, \ldots, sp_m)$ modelling a security pattern conjunction.*
$\text{defense}(b) = \{sp_1 \mid b = c^p(st, sp_1)\} \cup \{sp_1, \ldots, sp_m \mid b = c^p(st, \wedge^o(sp_1, \ldots, sp_m))\}$

Step 4: Final ADTree generation

In the initial ADTree T, each attack node labelled by Att is now automatically replaced with the ADTree $T(Att)$. This can be done by substituting every term Att in the ADTerm $\iota(T)$ by $\iota(T(Att))$. We denote T_f the resulting ADTree. It depicts a logical breakdown of the various options available to an adversary and the defences, materialised with security patterns, which have to be inserted into the application model.

In this step, we also extract from the data-store a description of the test architecture required to run the attacks on the application under test and to observe its reactions.

5 Test Suite Generation

The semantics of an ADTree can be defined in terms of attack-defense scenarios. In general terms, a 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 scenario, then the root goal will not be achieved. The semantics of an Adtree T_f, i.e. its scenario set, can be extracted from its ADTerm $\iota(T_f)$:

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

An attack scenario s of $SC(T_f)$ is an ADTerm over BADSteps. $BADStep(s)$ denotes the set of BADSteps of s. We also denote $SP(s)$ the security pattern set found in s: $SP(s) = \{sp \mid \exists b \in BADStep(s) : sp \in \text{defense}(b)\}$. By extension, $BADStep(T_f)$ stands for the set of BADSteps found in $\iota(T_f)$; $SP(T_f)$ is the security pattern set of $\iota(T_f)$, found in all its scenarios.

Step 5: Test suite generation

Let us consider a security scenario $s \in SC(T_f)$. Given a BADStep $b = c^p(st, sp) \in BADStep(s)$, we generate the GWT test case $TC(b)$, which aims at checking whether the application under test I is vulnerable to the attack

step st and whether the consequences of the security patterns in defense(b) can be observed from I. $TC(b)$ is assembled from the data-store by means of the following steps:

1. the data-store provides for st, with the relations $testG$, $testW$ and $testT$, one Given section, one When section and one Then section, each related to one procedure. The Then section aims to assert whether the application is vulnerable to the attack step st;
2. the data-store provides from the security pattern set defense(b) a set of other Then sections, each related to procedures. These Then sections aim to check whether the security pattern consequences can be observed from the application behaviours;
3. all these sections are assembled to make up the GWT test case stub $TC(b)$.

By applying these steps on all the scenarios of $SC(T_f)$, we obtain the test suite TS with $TS = \{TC(b) \mid b = c^p(st, sp) \in BADStep(s) \text{ and } s \in SC(T_f)\}$.

We implemented these steps to yield GWT test case stubs compatible with the Cucumber framework[7], which supports a large number of languages. Figure 5 gives a test case stub example obtained with our tool from the first step of the attack CAPEC-66 depicted in Figure 4. The test case lists the Given When Then sections in a readable manner. Every section is associated to a Generic procedure stored into another file. The procedure related to the When section is given in Figure 6. The comments comes from the data-store and are extracted from the CAPEC base. This procedure includes a generic block of code; the "getSpider()" method relates to the call of the ZAProxy tool, which crawls a Web application to get its URLs. In this example, it only remains for the developer to complete the initial URL before testing whether the application can be explored.

```
Feature: CAPEC-66: SQL Injection
#1. Explore
Scenario: Step1.1 Survey application
#The attacker first takes an inventory of the functionality exposed by the application.
Given   a new scanning session
When    spider the application
Then    the application is spidered
#assertions for security pattern testing
Then  Output Guard security pattern is present
Then   Input Guard security pattern is present
```

Fig. 5. A GWT test case example

Step 6: Test case stub completion

Now, the developer has to complete the previous GWT test case stubs. We believe that the decomposition of the test case and its link to the ADTree T_f (associations among steps, security patterns and procedures) make this step easier. In addition, the Generic procedures, composed of comments or blocks of code

[7] https://cucumber.io/

```
@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 with application using a utility such as WireShark.
getSpider().setMaxDepth(10);
url = "URL to be scanned";
try { spider(url);
} catch (InterruptedException e) {e.printStackTrace();}
waitForSpiderToComplete();}
```

Fig. 6. The procedure related to the When section of Figure 5

should make him or her more effective in the test case writing.

Step 7: Test suite execution

Once the GWT test case stubs are completed, these can be executed on the application under test I. The test architecture allowing the experimentation of I is described in the report provided by Step 4.

After the execution of one test case $TC(b)$ on I, denoted $TC(b)||I$, we obtain sets of verdict messages of the form "$Pass_{st}$", "$Fail_{st}$" or "$Fail_{sp}$", resulting from its assertions (see Section 3.1) :

Definition 4 (Test verdict sets). *Let I be an application under test, $b = c^p(st, sp) \in BADStep(T_f)$ with* defense$(b) = \{sp_1, \ldots, sp_m\}(m > 0)$ *and* $TC(b) \in TS$ *be a test case. The execution of $TC(b)$ on I leads to a verdict set, denoted* Verdict$(TC(b)||I)$*, which can be:*

- $\{Fail_{st}\}$ *(resp.* $\{Pass_{st}\}$*) means I is (resp. does not appear to be) vulnerable to the attack step st and that the consequences of the security patterns are observed;*
- $\{Pass_{st}, Fail_{sp_1}, ..., Fail_{sp_k}\}$ *means I does not appear to be vulnerable to the attack step st but some consequences of the security patterns $sp_1, \ldots, sp_k$ are not observed;*
- $\{Fail_{st}, Fail_{sp1}, ..., Fail_{sp_k}\}$ *means I is vulnerable to the attack step st and that some consequences of the security patterns $sp_1, \ldots, sp_k$ are not observed.*

From these verdict messages, we define two first relations. The first relation *vulnerable* defines that an application I is vulnerable to a BADStep b if the message $Fail_{st}$ belongs to the verdict set Verdict$(TC(b)||I)$. The relation $unsat^c$ defines that I does not satisfies the consequences of the pattern sp if the message $Fail_{sp}$ belongs to Verdict$(TC(b)||I)$:

Definition 5 (Test case verdicts). *Let I be an implementation, T_f be an ADTree, $b = c^p(st, sp) \in BADStep(T_f)$ and $TC(b) \in TS$ be a test case.*

1. *I vulnerable $b = true$ if $\{Fail_{st}\} \in$* Verdict$(TC(b)||I)$*, $false$ otherwise;*
2. *I $unsat^c$ sp if $sp \in$* defense(b) *and $Fail_{sp} \in$* Verdict$(TC(b)||I)$*.*

We now define the relation *effective* on a scenario $s \in SC(T_f)$, composed of the BADSteps $b_1, \ldots, b_n$ and on I to formally state whether s detects vulnerabilities on I. The relation *effective* is evaluated by substituting every BADStep term b_i with the evaluation of I vulnerable b_i. These relations help define and evaluate the final implementation relations.

Definition 6 (Implementation relations). *Let I be an implementation, T_f be an ADTree, and $s \in SC(T_f)$, with $BADStep(s) = \{b_1, \ldots, b_n\}$.*

1. *$\sigma : BADStep(s) \rightarrow \{true, false\}$ is a substitution $\{b_1 \rightarrow (I \text{ vulnerable } b_1), \ldots, b_n \rightarrow (I \text{ vulnerable } b_n)\}$;*
2. *s effective I, if the evaluation of the result $s\sigma$ of applying σ to s returns true;*
3. *I vulnerable $T_f \Leftrightarrow_{def} \exists s \in SC(T_f)$ s effective I;*
4. *I unsat^c $SP(T_f) \Leftrightarrow_{def} \exists sp \in SP(T_f), I$ unsat^c sp.*

6 Evaluation

We empirically studied two scenarios on 24 participants to assess whether developers can take profit of our approach. The duration of each scenario was set 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.

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: show that both Web sites are vulnerable to the two attacks, show that the application behaviours do not include at least one consequence of the security pattern "Input Guard" and at least one consequence of "Output Guard".

In the first scenario, denoted Part 1, we supplied the CAPEC base, two concrete attack examples, the detailed steps showing how to manually perform them along with the expected outcomes and the security pattern catalogue given in [12]. In the second scenario, denoted Part 2, we also supplied the ADTrees of the two attacks (Figure 4 is one of them) along with the generated GWT test case stubs for each attack step. At the end of each scenario, the students were invited to fill in a form listing ten questions. Due to lack of room, we only present the questions and results concerning the test case generation:

- Q7: Was it easy to write test cases?
- Q8: How long did you take for writing test cases?
- Q9: How confident are you about your test cases?
- Q10: Provide your test cases (or suites).

These questions was asked in order to evaluate the following criteria:

- C1: Comprehensibility: does our method ease the test case development?
- C2: Effectiveness: can the test cases detect defects?
- C3: Efficiency: does our method help reduce the time needed to write tests?

6.1 Experiment results

We extracted the following results from the forms returned by the participants (available here[8]). We collected the answers of Question Q7, proposing this four-valued scale: *easy, fairly easy, difficult, very difficult*. Figure 7 depicts the distribution of the participant opinions.

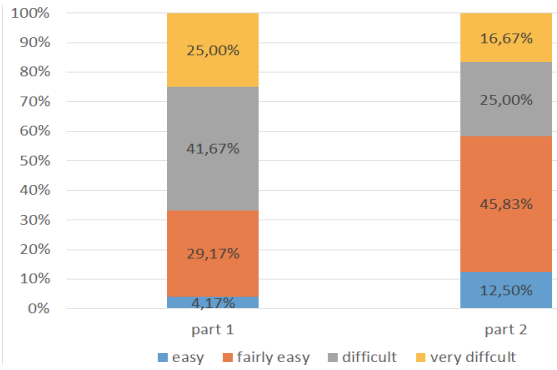


Fig. 7. Response rates for Question Q7

We collected the time spent by the participants for writing test cases. The participants needed 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 Part2. The levels of confidence of the participants is estimated with Question Q9. The possible answers were for both scenarios: *very sure, sure, fairly sure, not sure*.

We finally analysed the test cases given by the participants 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 8 presents the number of participants who meet these aspects.

6.2 Result interpretation

C1 Comprehensibility: we chose this criteria to evaluate whether our method makes testing easier. Figure 7 shows that one quarter of the students found easier the test case writing with our test case stubs. After discussion, it turned out that the test case structure with the GWT template made test cases more readable and that the links between test case sections and Attack steps helped students

[8] `http://regainia.com/research/companion.html`

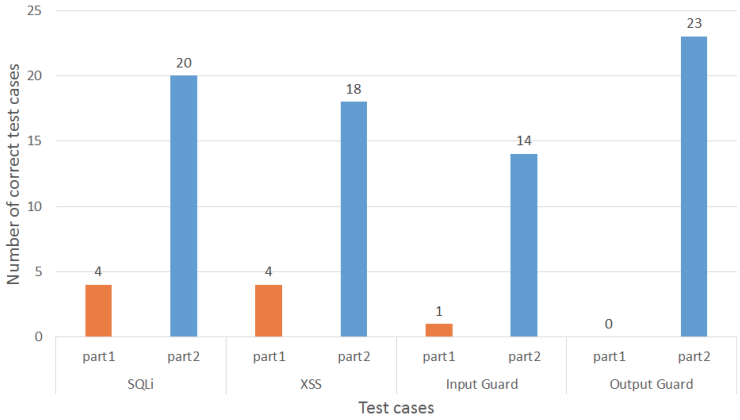


Fig. 8. Test case correctness (Question Q10)

understand what to develop. In the meantime, Question Q9 reveals that the confidence level of the participants about their test cases increases by 20,83%.

C2 Effectiveness: Figure 8 depicts the results about the test case correctness. The columns "SQLi" and "XSS" provide the number of test cases allowing to reveal that the attacks can be successfully executed on the applications. In Part 1, few participants developed complete test cases despite the detailed steps we provided (assertions were missing or incorrect in most of the test cases). The number of correct test cases strongly increases in Part 2 thanks to the comments the participants found in the procedures. The columns "Input Guard" and "Output Guard" give the number of Then sections (and procedures) allowing to show that the consequences of these security patterns are not observed from the application behaviours. This task was much more difficult for the students as they are not yet expert in security patterns. Hence, it is not surprising to see that only one student was able to write at least an assertion showing that the Input Guard consequences are not present. The number of correct Then sections rises to 14 (58,3%) in Part 2. With the pattern "Output Guard", the number of correct Then sections rises from 0 to 23 in Part 2. We can conclude that the test case correctness strongly increases with our approach.

C3 Efficiency: on average, the participants took 46 minutes for writing test cases from scratch and 60 minutes with the use of our method. 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 deduced that they took more time to follow and analyse the ADTrees, to read the comments in procedures, etc. As a result, almost all the test cases are correct in Part 2 (more assertions, etc.).

7 Conclusion

We have presented a method taking advantage of data integration for guiding developers devise more secure applications from the Threat modelling stage to

the testing one. The method generates ADTrees and test case stubs allowing to check whether an application is vulnerable to attacks and whether security pattern consequences are observable from the application behaviour. We conducted an evaluation of the method, which shows it makes the participants more effective on security testing. But, several issues remain open. For instance, our method does not take into consideration the size of the ADTrees in the Threat modelling stage. This is a strong limitation since large trees are usually unreadable, which contradicts the method purposes. The ADTree reduction could be a first solution on this problem. But, the literature does not yet provide a generic method for this kind of reduction. Besides the tree structure, the node meaning must be taken into account in the node aggregating process, which must preserve the semantics of the ADTree.

References

1. Alvi, Aleem, K., Zulkernine, M.: A Comparative Study of Software Security Pattern Classifications. 2012 Seventh International Conference on Availability, Reliability and Security pp. 582–589 (2012)
2. Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., Trujillo-Rasua, R.: Attack trees with sequential conjunction. In: IFIP International Information Security Conference. pp. 339–353. Springer (2015)
3. Kordy, B., Mauw, S., Radomirović, S., Schweitzer, P.: Attack–defense trees. Journal of Logic and Computation p. exs029 (2012)
4. Marback, A., Do, H., He, K., Kondamarri, S., Xu, D.: A threat model-based approach to security testing. Softw. Pract. Exper. 43(2), 241–258 (Feb 2013)
5. Marback, A., Do, H., He, K., Kondamarri, S., Xu, D.: Security test generation using threat trees. In: ICSE Workshop on Automation of Software Test. pp. 62–69 (May 2009)
6. Morais, A., Martins, E., Cavalli, A., Jimenez, W.: Security protocol testing using attack trees. In: International Conference on Computational Science and Engineering. vol. 2, pp. 690–697 (Aug 2009)
7. Regainia, L., Salva, S.: A methodology of security pattern classification and of attack-defense tree generation. In: Proceedings of the 3nd International Conference on Information Systems Security and Privacy (ICISSP). SciTePress, Porto, Portugal (feb 2017)
8. Schumacher, M.: Security Engineering with Patterns: Origins, Theoretical Models, and New Applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
9. Shahmehri, N., Mammar, A., Montes De Oca, E., Byers, D., Cavalli, A., Ardi, S., Jimenez, W.: An advanced approach for modeling and detecting software vulnerabilities. Inf. Softw. Technol. 54(9), 997–1013 (Sep 2012)
10. Yoshizawa, M., Kobashi, T., Washizaki, H., Fukazawa, Y., Okubo, T., Kaiya, H., Yoshioka, N.: Verifying implementation of security design patterns using a test template. In: 9th International Conference on Availability, Reliability and Security. pp. 178–183 (Sept 2014)
11. Yskout, K., Heyman, T., Scandariato, R., Joosen, W.: A system of security patterns (2006)
12. Yskout, K., Scandariato, R., Joosen, W.: Do security patterns really help designers? In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. pp. 292–302. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015)