A Practical Way of Testing Security Patterns

Loukmen Regainia and Sébastien Salva
LIMOS - UMR CNRS 6158
University Clermont Auvergne, France
email: loukmen.regainia@uca.fr, sebastien.salva@uca.fr

***Abstract*—We propose an approach for helping developers devise more secure applications from the threat modelling stage up to the testing one. This approach relies on a Knowledge base integrating varied security data to perform these tasks. It firstly assists developers in the design of Attack Defence Trees (ADTrees) expressing the attacker possibilities to compromise an application and the defences that may be implemented. These defences are expressed by means of security patterns, which are generic and re-usable solutions to design secure applications. ADTrees are then used to guide developers in the generation of test cases and Linear Temporal Logic (LTL) specifications. The latter encoding properties about security pattern behaviours. Test verdicts show whether an application is vulnerable to the attack scenarios and if the security pattern properties hold in the application traces.**

Keywords-Security pattern ; Security Testing ; Attack Defence Tree ; Test Case Generation.

I. Introduction

Preventing attackers from exploiting software defects, in order to compromise the security of applications or to disclose and delete user data, may be considered as the main motivations for software security. It is well admitted that the choice of security solutions and the audit of these solutions are two tasks of the software life cycle requiring time, expertise and experience. Unfortunately, developers lack resources and guidance on how to design or implement secure applications and test them. Furthermore, different kinds of expertise are required, e.g., to represent threats, to choose the most appropriate security solutions w.r.t. an application context, or to ensure that the latter are implemented as expected.

Several digitalised security bases, documents and papers have been proposed to guide developers in these activities. For instance, the Common Attack Pattern Enumeration and Classification (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., [1], list 176 reusable solutions for helping developers design more secure applications. The security pattern, which is a topic of this paper, *intuitively relates countermeasures to threats and attacks in a given context* [2]. This profusion of documents makes developers drown in a sea of recommendations taking security with different viewpoints (attackers, defenders, etc.), abstraction levels (security principles, attack steps, exploits, etc.) or contexts (system, network, etc.). In this paper, we focus on this issue and propose an approach to assist developers devise more secure applications from the threat modelling stage up to the testing one. The originality of this approach resides in the facts that it relies on a Knowledge base to automate some steps and it does not require that developers have skills in (formal) modelling.

Brief review of related work, and contributions: Numerous papers proposed methods for generating test cases from models to check the security of systems, protocols or applications. Among them, several papers, e.g., [3], [4], focused on models not to describe the implementation behaviour but rather to express attacker goals or vulnerability causes of a system. 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.

We proposed in [5] a preliminary approach for helping developers devise more secure applications with a guided test case generation approach. It is based on a first Knowledge base integrating security data. The approach firstly queries the Knowledge base to help developers write an Attack Defense Tree (ADTree) encoding the attack scenarios that may be performed by an adversary and the defences, materialised with security patterns, which have to be integrated and implemented into the application. Thereafter, the approach helps generate security test cases to check whether the application is vulnerable to these attacks. However, it does not assist developers to ensure that security patterns have been correctly implemented in the application. This work supplements our early study by covering this part.

Few works dealt with the testing of security patterns, which is the main topic of this paper. Yoshizawa et al. introduced a method for testing whether behavioural and structural properties of patterns may be observed in application traces [6]. Given a security pattern, two test templates (Object Constrant Languauge (OCL) expressions) are manually written, one to specify the pattern structure and another to encode its behaviour. Then, developers have to make templates concrete by manually writing tests for experimenting the application. The latter returns traces on which the

OCL expressions are verified. Our approach requires neither complete models nor formal properties. It generates ADtrees to help developers choose security patterns. Furthermore, with our approach, developers do not need to have a good knowledge and skills on the writing of formal properties. Instead, we propose a practical way to generate them. Intuitively, after the choice of security patterns, our approach provides UML sequence diagrams, which can be modified by the developer. From these diagrams, we generate LTL properties, which capture the cause-effects relations among pairs of method calls. After the test case execution, we check if these properties hold in application traces. The developer is hence not aware of the LTL formula processing.

Paper Organisation: Section II introduces the Knowledge base used by our approach. Section III gives the first three steps of the approach, which aim at generating threat models, proposing security patterns and providing UML sequence diagrams. Section IV addresses the generation of test cases and LTL properties, which are used to return test verdicts. We finally conclude in Section V.

II. Knowledge base overview

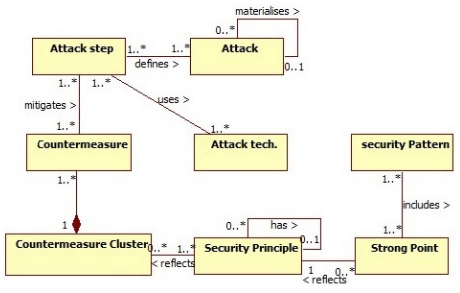


Figure 1. Knowledge Base meta-model part 1

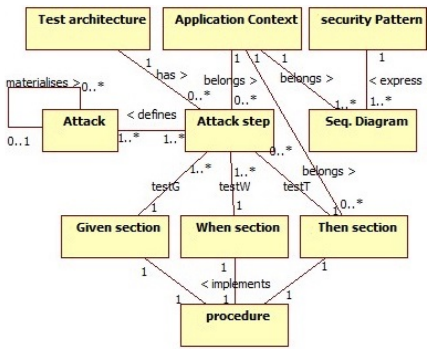


Figure 2. Knowledge Base meta-model part 2

Our approach relies on a Knowledge base to automate some of its steps. Its meta-model is depicted in Figures 1 and 2. We summarise its architecture in the following but we refer to [5] for a complete description and for the data integration. The meta-model associates attacks, techniques, security principles, security patterns, test cases and UML sequence diagrams. 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 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.

In Figure 2, an attack step is also associated to one Application context and one Test architecture. The context refers to an application family, e.g., 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 Given When Then (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 met (assertions). We suppose that a Then section returns the message "$Pass_{st}$" if an attack step st has been successfully executed, "$Inconclusive_{st}$" if some test case procedures have not been executed due to various problems (e.g., incomplete test architecture, network issues, etc.) or "$Fail_{st}$" otherwise.

A test case section is linked to one procedure stored in the Procedure table, which implements the section. These procedures may be completed with comments or with blocks of code to ease the test case development. But, they must remain generic, i.e., re-usable with any application in a precise context.

For this paper, we have updated the Knowledge base in such a way that a security pattern is also associated to UML sequence diagrams, themselves adapted to application contexts. Indeed, security pattern catalogues often provide UML sequence diagrams expressing the security pattern behaviours. These diagrams show how to implement a security pattern with regard to an Application context.

As a proof of concept, we generated a Knowledge base specialised to Web applications (The paper [5] details some data acquisition and integration steps). It includes information about 215 attacks (209 attack steps, 448 techniques), 26 security patterns, 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 and which observations reveal that the application is vulnerable. We manually completed 32 procedures, which cover 43 attack steps. We used the testing framework Selenium and the penetration testing tool ZAProxy [7], which covers varied Web vulnerabilities. This Knowledge base is available here [8].

III. Security and security pattern testing

Figure 3 depicts an overview of the six steps of our approach, beginning by the construction of a threat model and ending with the generation of test verdicts expressing whether an Application Under Test (AUT) is vulnerable to the attack scenarios encoded in the threat model and whether security pattern behaviours hold in the AUT traces. Before describing these steps, we briefly recall some notions about the ADTree model.

A. 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* [9]. ADTrees have two 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 (as in Figure 4) or conjunctive. The latter is graphically distinguishable by connecting the edges between a root node and its children with an arc. We extend these two refinements with the sequential conjunctive refinement of attack nodes. This operator expresses the execution order of child attack nodes. Graphically, a sequential conjunctive refinement is illustrated by connecting the edges, going from a node to its children, with an arrow.

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).

B. Model Generation, Security Pattern Choice (Step 1 to 3)

Step 1: Initial ADTree Design

The developer establishes an initial ADTree T whose root node represents the attacker's goal. This node may be refined with several layers of children to refine this goal. We suppose that T at least has leaves labelled by attacks available in the Knowledge base. Otherwise, a semantic alignment may be required to replace some attack labels.

An example is given in Figure 4: the goal, given in the root node of the ADTree, refers to the injection of malicious code into an application. This goal is disjunctively refined by two children expressing two more concrete attacks, described in

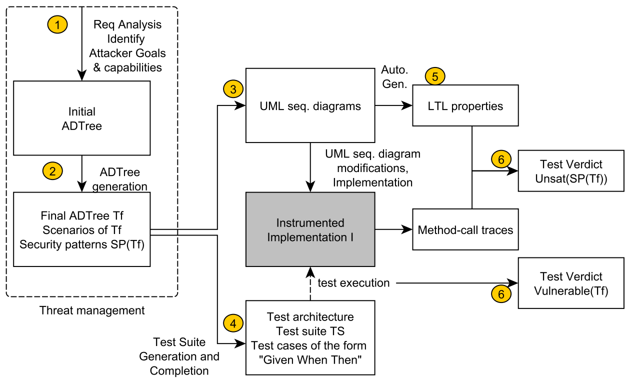


Figure 3. Overview of the 6 steps of the approach

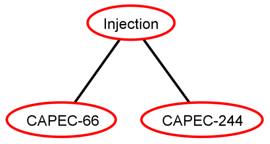


Figure 4. Initial ADTree example

the CAPEC base: CAPEC-66: SQL Injection and CAPEC-244: Cross-Site Scripting via Encoded URI Schemes.

Step 2: Detailed ADTree Generation

The Knowledge base is now queried to automatically complete T with more details about the attack proceeding and with defense nodes labelled by security patterns. We summarise this step in the following:

For every node labelled with an attack Att, the Knowledge base is called to automatically generate an ADTree denoted $T(Att)$. This ADTree has a specific form satisfying the meta-model of the Knowledge base. More precisely, the root of $T(Att)$ 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 purposes are to counteract the attack steps. The developer can now edit the ADTrees $T(Att)$ to edit some 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. These generated ADTrees have a specific form, which are encoded by these 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 ADTerm is composed of an attack step st refined with techniques, which can be counteracted by a security pattern combination $sp = \wedge^o(sp_1, \ldots, sp_m)$. We call this ADTerm a Basic Attack Defence Step $c^p(st, sp)$, shortened BADStep. $BADStep(T_f)$ denotes the set of BADSteps of T_f (final ADTree).

Figure 5 depicts a part of the ADTree of the attack CAPEC-66. Each lowest attack step has a defense node expressing 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 the pattern "Application Firewall" can be replaced by "Intercepting Validator" or "Output Guard".

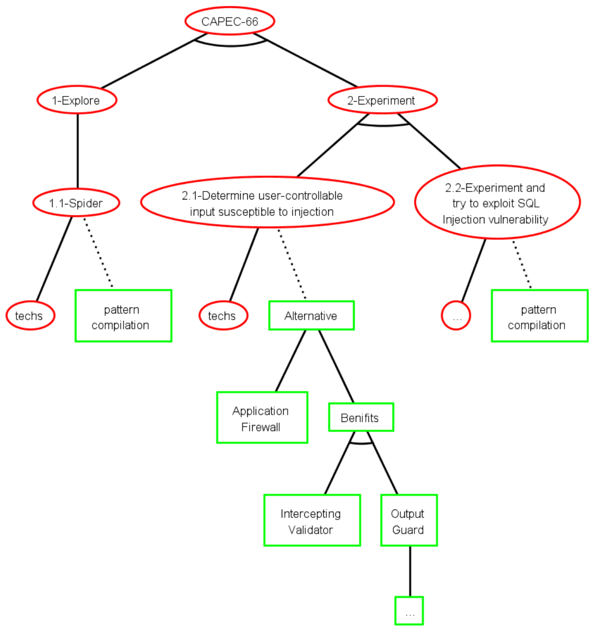


Figure 5. Part of the ADTree of the Attack CAPEC-66

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, and $SP(T_f)$ the security pattern set found in $\iota(T_f)$.

In this step, we finally extract from the Knowledge base a description of the test architecture required to run the attacks on the application under test and to observe its reactions.

Step 3: UML Sequence Diagram Extraction

For every security pattern found in T_f, we extract a list of UML sequence diagrams from the Knowledge base, each being related to the application context. These show the behavioural activities of the patterns. We now suppose that the developer implements every security pattern in the application. At the same time, he/she can choose to modify the generic class and method names labelled in the UML sequence diagrams. In this case, we assume that the sequence diagrams are annotated to point out these modifications.

As example, Figure 6 illustrates the UML sequence diagram of the security pattern "Intercepting Validator", whose purpose is to control the compliance of user requests with regard to a specification. The validation must be performed in the server side. For instance, while the pattern implementation, if the name of the method "process" has to be modified by "send", the new label must be of the form "process/send" to express the substitution.

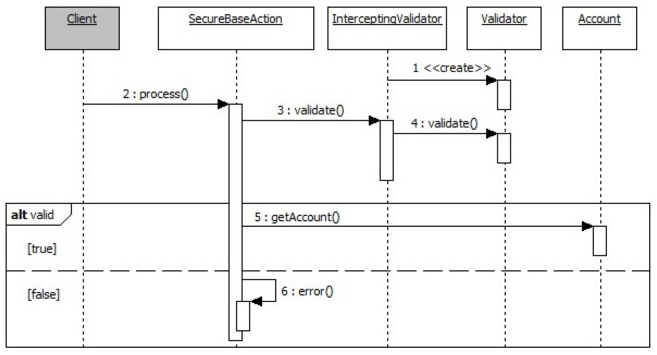


Figure 6. Intercepting Validator sequence diagram

IV. Attack and Security Pattern Testing

At this stage, an ADTree encodes the notion of attack scenarios over BADSteps, a scenario being is a minimal combination of events leading to the root attack.

Definition 2 (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 is still an ADTerm over BADSteps. $BADStep(s)$ denotes the set of BADSteps of s.

Step 4: 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 AUT is vulnerable to the attack step st. $TC(b)$ is constructed by extracting from the Knowledge base, for the attack step st, 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; these sections are assembled to make up the GWT test case stub $TC(b)$.

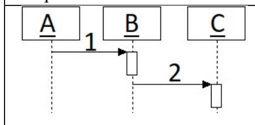
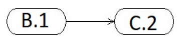
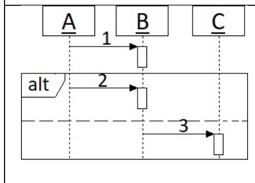
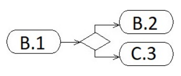
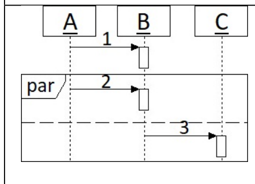
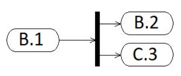
After having iteratively applied this test case construction on the scenarios of $SC(T_f)$, we obtain the test suite TS with $TS = \{TC(b) \mid b = c^p(st, sp) \in BADStep(s)$ and $s \in SC(T_f)\}$.

Step 5: Security Pattern LTL Property Generation

Our approach aims at checking whether security pattern behavioural properties hold in the AUT. Instead of asking the developer to write these properties, we automatically generate them from UML sequence diagrams. This step analyses sequence diagrams, recognises behavioural characteristics and translates them into LTL properties.

Given a security pattern sp and its UML sequence diagram, the latter is firstly transformed into a UML activity diagram. We propose 5 transformation rules whose three are depicted in Table I. Intuitively, these rules transform each method call in the sequence diagram by an action state in the activity diagram. We took inspiration from the transformations of UML sequence diagrams to state machines proposed in [10]. This transformation allows us to use the mapping of UML activity diagrams to LTL properties proposed by Muram et al. [11]. The transformation rules are based upon the Response Property Pattern [12], which describes the cause-effect relations among method calls. Three examples of transformations are given in Table I. At the end of this transformation sequence, we have a set of LTL properties $P(sp)$ for every security pattern $sp \in SP(T_f)$. Although the LTL properties of $P(sp)$ do not necessarily cover all the possible behavioural properties of a security pattern sp, this process offers the advantages to not require generic LTL properties modelling pattern behaviours, and to not ask developers to instantiate these generic LTL properties to match the application model or code.

TABLE I
TRANSFORMATION RULES

Sequence	Activity	LTL properties
		$\square(B.1 \longrightarrow \Diamond C.2)$
		$\square(B.1 \longrightarrow (\Diamond B.2) xor (\Diamond C.3))$
		$\square(B.1 \longrightarrow (\Diamond B.2) and (\Diamond C.3))$

From the example of UML sequence diagram given in Figure 6, four LTL properties are generated. Table II lists them. These capture the cause-effect relations of every pair of methods found in the UML sequence diagram.

TABLE II
LTL PROPERTIES OF THE PATTERN INTERCEPTING VALIDATOR

p_1	$\square(Controller.SecureBaseAction.process \longrightarrow \Diamond InterceptingValidator.Validator.create)$
p_2	$\square(InterceptingValidator.Validator.create \longrightarrow \Diamond interceptingValidator.InterceptingValidator.validate)$
p_3	$\square(InterceptingValidator.InterceptingValidator.validate \longrightarrow \Diamond InterceptingValidator.Validator.validate)$
p_4	$\square(InterceptingValidator.Validator.validate \longrightarrow (\Diamond model.Account.getAccount) xor (\Diamond Controller.SecureBaseAction.error))$

Step 6: Test Verdict generation

Once the GWT test case stubs are completed by the developer, these are executed on AUT. The test architecture allowing the experimentation of AUT is described in the report provided by Step 2. The execution of a test case $TC(b)$ on AUT, leads to a local verdict denoted $\text{Verdict}(TC(b)||AUT)$, which takes as value a test case assertion message. Furthermore, we consider that the AUT is instrumented with a debugger or similar tool to collect the methods called in the application while the execution of the test cases of TS. After the test case execution, we hence have a set of method call traces of AUT denoted $Traces(AUT)$. With a model-checking tool, e.g., Declare2LTL [13], our approach can now detect the non-satisfiability of LTL properties of a security pattern sp on $Traces(AUT)$. The predicate $Unsat^b(sp)$ defines this detection:

Definition 3 (Local Test Verdicts) *Let AUT be an application under test, $b = c^p(st, sp) \in BADStep(T_f)$, sp_1 a security pattern in sp, $TC(b) \in TS$ be a test case.*

1) $\text{Verdict}(TC(b)||AUT) =$
-$Fail_{st}$ (resp. $Pass_{st}$) means AUT is (resp. does not appear to be) vulnerable to the attack step st;
-$Inconclusive_{st}$ means that various problems occurred while the test case execution.
2) $\text{Unsat}^b(sp_1) =_{def}$ *true if* $\exists p \in P(sp_1), \exists t \in Traces(AUT), t \nvDash p$*; otherwise,* $\text{Unsat}^b(sp_1) =_{def} false$*;*

Subsequently, we define the final test verdicts with regard to the ADTree T_f. These verdicts are given with the predicates $\text{Vulnerable}(T_f)$, $\text{Unsat}^b(SP(T_f))$ and $\text{Inconclusive}(T_f)$ returning boolean values. The predicate $\text{Vulnerable}(b)$ is also defined on a BADStep b to later apply a substitution $\sigma : BADStep(s) \rightarrow \{true, false\}$ on an attack-defense scenario s. A scenario s holds if the evaluation of the substitution σ to s (i.e., replacing every BADStep term b with the evaluation of $\text{Vulnerable}(b)$) returns true. When a scenario of T_f holds, then the threat

TABLE III
TEST VERDICT SUMMARY AND SOLUTIONS

Vulnerable(T_f)	$\mathrm{Unsat}^b(SP(T_f))$	Incon (T_f)	Corrective actions
False	False	False	No issue detected
True	False	False	At least one scenario is successfully applied on AUT. Fix the pattern implementation. Or the chosen patterns are inconvenient.
False	True	False	Some pattern behavioural properties do not hold. Check the pattern implementations with the UML seq. diag. Or another pattern conceals the behaviour of the former.
True	True	False	The chosen security patterns are useless or incorrectly implemented. Review the ADTree, fix AUT.
T/F	T/F	True	The test case execution crashed or returned unexpected exceptions. Check the Test architecture and the test case codes.

modelled by T_f can be achieved on AUT. This is defined with the predicate $\mathrm{Vulnerable}(T_f)$. $\mathrm{Unsat}^b(SP(T_f))$ is true as soon as a security pattern property does not hold on $Traces(AUT)$. Table III informally summarises the meaning of some test verdicts and some corrections that may be followed in case of failure.

Definition 4 (Final Test Verdicts) *Let AUT be an application under test, T_f be an ADTree, $s \in SC(T_f)$ and $b = c^p(st, sp) \in BADStep(T_f)$.*

1) $\mathrm{Vulnerable}(b) =_{def}$ *true if* $\mathrm{Verdict}(TC(b)||AUT) = Fail_{st}$*; otherwise,* $\mathrm{Vulnerable}(b) =_{def} false$*;*
2) $\sigma : BADStep(s) \rightarrow \{true, false\}$ *is a substitution* $\{b_1 \rightarrow \mathrm{Vulnerable}(b_1), \ldots, b_n \rightarrow \mathrm{Vulnerable}(b_n)\}$*;*
3) $\mathrm{Vulnerable}(T_f) =_{def}$ *true if* $\exists s \in SC(T_f) : eval(s\sigma)$ *returns true; otherwise,* $\mathrm{Vulnerable}(T_f) =_{def} false$*;*
4) $\mathrm{Inconclusive}(T_f) =_{def}$ *true if* $\exists s \in SC(T_f)$, $\exists b \in \mathrm{BADStep}(s)$*:* $\mathrm{Verdict}(TC(b)|| AUT) = Inconclusive_{st}$*; otherwise,* $\mathrm{Inconclusive}(T_f) =_{def} false$.
5) $\mathrm{Unsat}^b(SP(T_f)) =_{def}$ *true if* $\exists sp \in SP(T_f), \mathrm{Unsat}^b(sp) = true$*; otherwise,* $\mathrm{Unsat}^c(SP(T_f)) =_{def} false$*;*

V. CONCLUSION

This paper proposes an approach taking advantage of a Knowledge base to assist developers in the implementation of secure applications through six steps covering threat modelling, the choice of security patterns, security testing and the verification of security pattern behavioural properties. It guides developers in the generation of ADTrees and test cases. In addition, it automatically generates LTL properties, encoding security pattern behaviours. As a consequence, the approach does not require developers to have skills in (formal) modelling or in formal methods. We have implemented this approach in a tool prototype [8]. We briefly summarise its features in this paper: it generates ADTrees stored into XML files, which can be edited with *ADTool* [9]. Our tool also builds GWT test case compatible with the Cucumber framework [14], which supports a large number of languages. These test cases can be imported with the IDE Eclipse. The verification of LTL properties is performed with the Declare2LTL model checker. We started to perform some experiments on Web applications to assess the user benefits. An evaluation will be presented in a future work.

REFERENCES

[1] R. Slavin and J. Niu. (retrieved: 07, 2018) Security patterns repository. [Online]. Available: http://sefm.cs.utsa.edu/repository/

[2] M. Schumacher, *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[3] A. Morais, E. Martins, A. Cavalli, and W. Jimenez, "Security protocol testing using attack trees," in *2009 International Conference on Computational Science and Engineering*, vol. 2, Aug 2009, pp. 690–697.

[4] N. Shahmehri et al., "An advanced approach for modeling and detecting software vulnerabilities," *Inf. Softw. Technol.*, vol. 54, no. 9, pp. 997–1013, Sep. 2012.

[5] S. Salva and L. Regainia, "Using data integration for security testing," in *Proceedings 29th International Conference, ICTSS 2017*, 10 2017, pp. 178–194.

[6] Y. Masatoshi et al., "Verifying implementation of security design patterns using a test template," in *2014 Ninth International Conference on Availability, Reliability and Security*, Sept 2014, pp. 178–183.

[7] OWASP. (retrieved: 07, 2018) Zap proxy project. [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[8] R. Loukmen and S. Sbastien. (retrieved: 07, 2018) Zap proxy project. [Online]. Available: http://regainia.com/research/companion.html

[9] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack–defense trees," *Journal of Logic and Computation*, p. exs029, 2012.

[10] R. Grønmo and B. Møller-Pedersen, "From sequence diagrams to state machines by graph transformation," in *Theory and Practice of Model Transformations*, L. Tratt and M. Gogolla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 93–107.

[11] F. U. Muram, H. Tran, and U. Zdun, "Automated mapping of UML activity diagrams to formal specifications for supporting containment checking," in *Proceedings FESCA 2014, Grenoble, France, 12th April 2014.*, 2014, pp. 93–107.

[12] L. Patterns. (retrieved: 07, 2018) response ltl pattern. [Online]. Available: http://patterns.projects.cs.ksu.edu/documentation/patterns/response.shtml

[13] M. M. Fabrizio et al. (retrieved: 07, 2018) Declare toolkit. [Online]. Available: http://www.win.tue.nl/declare/

[14] Cucumber. (retrieved: 07, 2018) Cucumber website. [Online]. Available: https://cucumber.io/