Model-driven testing of security protocols with SecureMDD

Kuzman Katkalov, Nina Moebius, Kurt Stenzel, Marian Borek, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Katkalov, Kuzman, Nina Moebius, Kurt Stenzel, Marian Borek, and Wolfgang Reif. 2012. "Model-driven testing of security protocols with SecureMDD." In 2012 5th International Conference on New Technologies, Mobility and Security (NTMS), 7-10 May 2012, Istanbul, Turkey, edited by Albert Levi, Mohamad Badra, Matteo Cesana, Mona Ghassemian, Özgür Gürbüz, Nafaâ Jabeur, Marek Klonowski, Antonio Maña, Susana Sargento, and Sherali Zeadally, 1–5. Piscataway, NJ: IEEE. https://doi.org/10.1109/ntms.2012.6208678.

Model-Driven Testing of Security Protocols with SecureMDD

Kuzman Katkalov, Nina Moebius, Kurt Stenzel, Marian Borek and Wolfgang Reif
Institute for Software and Systems Engineering
University of Augsburg, 86135 Augsburg, Germany
{kuzman.katkalov,stenzel,borek,moebius,reif}@informatik.uni-augsburg.de

***Abstract*—Designing and executing test cases for security protocols is a tedious and technically complicated process. The SecureMDD approach allows intuitive, model-driven development of security-critical applications based on cryptographic protocols. With this paper we introduce a method which combines the model-driven approach used in SecureMDD with the design of functional and security tests. We construct and evaluate new modeling guidelines that allow the modeler to easily define such test cases during the modeling stage. We also implement model transformation routines to generate runnable tests for applications developed with SecureMDD.**

***Index Terms*—model-driven testing, security protocols, security tests, model-driven software development, unit tests**

I. Introduction

Developing a complex and secure system is not only a long and complicated process, but also very prone to errors. While a malicious user often only has to find one flaw in order to exploit an application, the developer has to cover all possible intrusion points and consider all possible attacking scenarios in order to make his system secure. Testing security-critical applications during development stages is essential to the quality of the final product, thus demanding easy and intuitive testing methods. Those methods should allow the modeler or an external tester to both test the functionality of the code as well as to emulate a malicious user in order to eliminate possible attacks on the implementation.

Using a model-driven approach to design complex and secure systems can simplify the development process significantly, thus explaining its popularity with developers. In this paper we introduce a method to test a security-critical application based on cryptographic protocols. Our approach is model-driven and allows the modeling of test cases with UML. Based on the UML model Java test code is generated fully automatically, which allows to run the tests on code level. The test framework is integrated into SecureMDD [1], which is a model-driven approach to develop security-critical systems.

Section II introduces the SecureMDD approach. Section III focuses on modeling guidelines for test cases in SecureMDD using a running example application “Copycard”. Section IV explains the process of code generation and shows the resulting test framework architecture for SecureMDD applications. Section V discusses some of the related work on model-driven testing, while section VI concludes this paper and provides an outlook on future development.

II. SecureMDD Approach

SecureMDD [1] offers a method to model an application in UML, supporting the design of secure cryptographic protocols between several components. Furthermore, the correctness of those protocols can be proved using formal methods. SecureMDD also considers the implementation of the system, providing a framework to generate executable and deployable code for smart cards, terminals (i.e. components that are connected to a smart card reader and communicate with smart cards) [2] as well as services.

A SecureMDD model of an application fully describes its static structure using class and deployment diagrams as well as dynamic behavior of modeled components using sequence and activity diagrams. It utilizes a SecureMDD UML profile, extending the models with stereotypes to model cryptographic aspects of an application, as well as a predefined set of security datatypes like cryptographic keys or nonces. Each sequence- and activity diagram models a (cryptographic) protocol between several components. While the sequences only give an overview over the exchanged messages, the activities fully describe the components’ behavior. To be able to completely model a system, SecureMDD defines a simple programming language called Model Extension Language (MEL). This is an object-oriented, UML-centric language with no support for loops; it does, however, provide all relevant cryptographic primitives and functions. MEL code can be parsed and analyzed within the context of the whole model in order to generate runnable Java or Java Card code for each component. Java Card hardware poses several limitations on card applet code [3] like limited available memory and lack of garbage collection, which is taken into account when generating code e.g. by reusing objects to avoid dynamic memory allocation.

Furthermore, the SecureMDD framework allows the automatic generation of a formal abstract state machine (ASM) model from UML, which can then be used to verify security properties using the interactive theorem prover KIV [4] [5].

III. Designing Test Cases

A. Test case modeling guidelines

We focus on testing code generated from a platform-independent application model by the SecureMDD framework to not only find protocol logic flaws, but also bugs in the implementation. To achieve this, we support functional tests to

verify the functionality of modeled protocols and component behavior. We also support security test cases by modeling attack sequences to emulate attacks on a protocol and find security vulnerabilities even prior to protocol verification. Such attacks can be executed in the real world by e.g. intercepting the communication between a smart card and a terminal using special hardware or implementing a fake terminal or smart card, which is used to communicate with their real counterparts. Until the test framework was introduced in SecureMDD, functional tests had to be coded by hand. Emulating an attacker was an infeasible task, since all communication is encoded and is hard to intercept, and has thus never been done before in SecureMDD. We support the interactive verification of security properties as well, but this is a very time-consuming task and the replaying of proofs (after finding a flaw in the protocol and fixing it) is very difficult. For this reason, it is very important to support the intuitive modeling of tests as well as the generation of test code.

Our test cases are modeled with activity diagrams using MEL and are placed in the model project of the application. Test cases are structured as follows: setup all needed components, run existing protocols of the model or attack protocols and check the results.

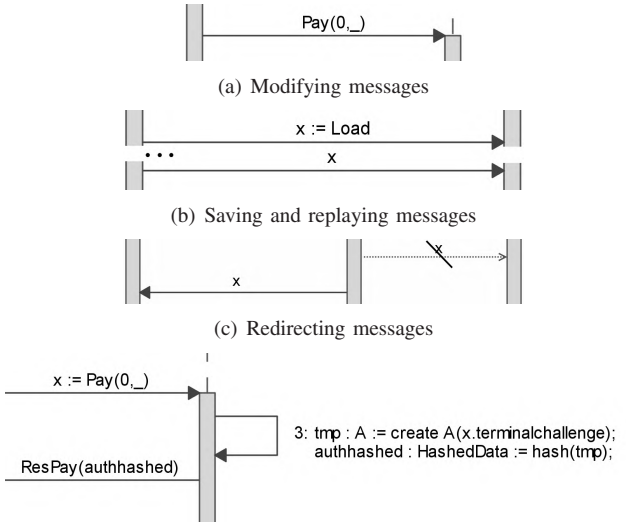
In order to run a modeled protocol or an attack on a protocol, the respective sequence diagram can be referenced from the test case diagram using the conventional Java/MEL method call convention; the name of the protocol or the attack is treated as a method name, while instantiated components are passed as parameters. Original protocol sequence diagrams are already part of SecureMDD modeling guidelines as simplified versions of corresponding activity diagrams; copies of those sequence diagrams can be modified to model protocol attacks. By applying UML constraints to flows or nodes it is possible to define checks like "does this smart card attribute ever exceed a certain value" in MEL on any of a component's attributes prior, after or even during a protocol run. Those checks and invariants can be used to test the correctness of a protocol run. If any of the constraint checks fail, the test is aborted and the tester is notified, e.g. via the JUnit Eclipse UI. Since component code can throw exceptions, e.g. when it can't understand an incoming message, the modeler can provide a test case exception handling routine by attaching a UML exception handler to the activity node which is calling a protocol or an attack.

Designing modeling guidelines for protocol attacks by a Dolev-Yao attacker [6] is a challenging task. Appropriate models have to be intuitively understandable, while also allowing the tester to easily create and modify attacks on any supported protocol.

Our modeling guidelines allow the modification, redirection, suppression and replaying of messages between components, without introducing an explicit attacker. This is done by renaming, adding or deleting messages from copies of sequence diagrams depicting application protocols.

To modify the contents of an intercepted message, we use the MEL activity diagram convention of treating messages like method calls, using message contents as parameters. By adding such parameters to a message name, the test modeler can provide new message content values as literals. If a value is to remain untouched, an underscore ("_") can be used as a parameter. Fig. 1(a) shows the modification of a `Pay` message. The first argument is set to zero, the second argument remains unchanged. The order of those parameters is defined in the class diagram, where messages are modeled as classes with attributes depicting their contents. Of course, not only the contents but the class of the message can be changed as well, essentially replacing the whole intercepted message; in this case, the entire name of the message needs to be modified to reflect the change.

Also, the attack sequence modeler is able to save any message into an implicitly defined variable as shown in Fig. 1(b). This variable can be used to implement replay attacks by inserting new messages into the sequence with the same name as the variable. In this figure, the message `Load` is stored in variable `x` and replayed later. Furthermore, it can be handled like any MEL-object of this message class.



(a) Modifying messages

(b) Saving and replaying messages

(c) Redirecting messages

(d) Executing custom MEL code

Fig. 1. Protocol attack model elements

Any message can be redirected to almost any lifeline, allowing the modeler to design elaborate attacks involving more than one card or terminal. To achieve this, the modeler can simply point the message to a different receiving lifeline (see Fig. 1(c)).

To suppress a message, the modeler simply has to delete it from the sequence diagram. Note that the corresponding response message has to be removed as well.

To allow even more complex attacks, we introduce MEL code in sequence diagrams beyond simple message storing in variables. By inserting to-self-messages which contain valid MEL-code, it is possible to manipulate intercepted messages or create new messages. This enables the modeler to create objects in order to assemble new messages and use crypto primitives like hashing, signing or en-/decrypting. The modeler

has access to previously stored messages and the attributes of the component which is represented by the lifeline. Fig. 1(d) shows an example for such custom code, which creates a new object of class `A` using parts of the previously saved `Pay` message and hashes it. The result is then being sent as the content of the `ResPay` message.

This section might beg the question why our test cases are not being generated automatically, without being modeled first. After all, it is possible to derive functional test cases from a model using additional model elements like constraints. Initial component state and attributes as well as user message values could be generated, looked up in a pool of predefined values or set by the modeler. However, modeling functional test cases manually still makes sense in certain scenarios, e.g. to test different pre- and postconditions for different protocol runs, initial component states or attribute values or to define more complex initialization routines; furthermore, criteria for automatically generating security test cases are not clear. While it would be possible to create generic, predefined protocol attacks to automatically assemble security test cases, most application specific attack scenarios could not be covered using this strategy. Thus, it seems plausible to create a test framework and modeling guidelines to allow the manual modeling of test cases, and to complement this modeling process by providing some automatically generated test case models as well as support for a library of predefined protocol attacks in the future.

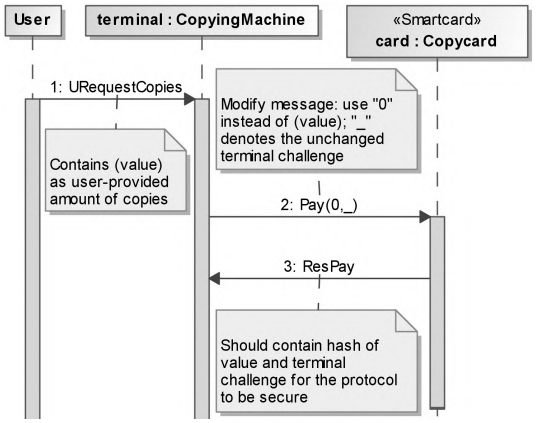
B. Testing the "Copycard" application

"Copycard" is an example application modeled in SecureMDD which allows depositing points on a smart card by inserting money into a terminal and to redeem those points at a copying machine. This application uses the protocols `pay`, `load` and `getBalance`. Protocol `pay` is initiated between a card and a terminal when the user wants to redeem his copycard points by requesting a copy and inserting his card into a copying machine. Protocol `load`, on the other hand, is used to load points on the copycard. In both protocols, the components generate a challenge and exchange a hashed secret passphrase to authenticate themselves.

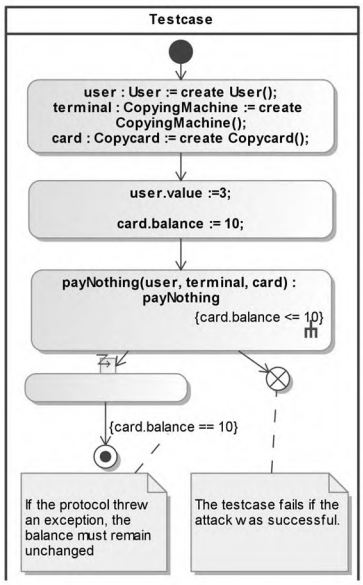
The developer has to ensure that no money is lost or generated by a malicious user (i.e. an attacker could try to program his own copycard or try to load money on his copycard by programming his own terminal) or a flaw in the system. Only a real, verified terminal can be used to load the copycard, and only a real copycard can be used to pay for copies. We are interested in testing the actual code implementation by running functional test cases or modifying the message flow between a terminal and a smart card, as an attacker would do.

Fig. 2(a) shows an attack on the `pay`-protocol; this is a modified copy of a sequence diagram describing the `pay`-protocol, which is already part of the model[1].

[1]A version of the Copycard model can be found at http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD



(a) `payNothing` attack on the `pay`-protocol to receive free copies



(b) Executing the attack

Fig. 2. Calling a protocol attack `payNothing` (a) from test case (b)

The message flow of this protocol between the terminal and the card can be learned from Fig. 2(a), as the shown attack only introduces modified contents of the `Pay` message. The protocol is initiated by the user (i.e. the GUI controlled by the user) by sending a `URequestCopies` message to the terminal, containing the amount of copies to be made. The terminal then sends a `Pay` message to the copycard containing the amount of points to be subtracted from the card's balance, as well as a randomly generated challenge. The card subtracts this amount from its internal balance and calculates a hash of a secret passphrase and the received challenge. This hash is sent to the terminal via the `ResPay` message, which also calculates a hash using the same data and compares it to the received message prior to printing the copies. This procedure is done to authenticate the copycard to the terminal (using the passphrase) and to prevent replay attacks (using the challenge).

As a tester we assume a flaw in the protocol, allowing a

Dolev-Yao attacker to modify the amount sent by the terminal so that no points are debited from the Copycard balance. We base this attack on the intuition (which is usually grounded on creativity, experience and knowledge about common protocol attacks) that the terminal might not check whether the card debits the correct amount of points. Since the `Pay` message consists of more information than just the amount, we can specify the other content to stay untouched with an underscore "_" as an argument (see section III-A). We can then prompt the test framework to run the `pay`-protocol using the modification specified in Fig. 2(a) by defining a test case pictured in Fig. 2(b). There we first initialize the three components `User`, `CopyingMachine` and `Copycard`. In the next step we initialize the attributes of the components. The attribute value of the user is set to three, the card's balance is set to ten. Then we call the `payNothing` attack protocol (see Fig. 2(a)) requesting three (the users's `value` attribute) copies from the copying machine. During the protocol run, the invariant `card.balance <= 10`, modeled as a constraint on the protocol run node is checked at all times, to exclude the scenario where the attacker can generate points by ejecting the card during the protocol run. As a modeler and a tester, we want the `pay`-protocol to be secure; the test case is modeled to fail, if the attack on the protocol was successful, which in this case would also mean that none of the components fail to process an incoming message. If one of the components doesn't understand an attacker message and aborts, or the card doesn't accept the modified amount of points, an exception will be thrown, which will be caught by the test case, since a UML exception handler has been attached to the protocol call and the balance of the card checked for changes using the constraint `card.balance == 10`. If this assertion check doesn't fail, the test case succeeds, which is modeled with an `Activity Final Node`. If no exception was thrown, then all components processed the attacker-modified messages successfully, in which case the attacker receives free copies; in this case, the test case is thus modeled to fail by employing a `Flow Final Node`.

The `pay`-protocol can be easily fixed by including the amount of points to be debited in the hash calculation. Then the terminal is able to detect when a copycard receives the incorrect amount of points and won't print any copies.

IV. Generating Code

SecureMDD uses model-to-model and model-to-text transformations to generate platform-specific UML models as well as deployable Java and Java Card[2]-code. After the UML model has been exported to the EMF-UML2 file format, we can easily invoke the transformations using our own Eclipse plugin. Several platform specific UML models are generated from the input model using QVTO[3] scripts: one for each component type respectively (terminal, smartcard and/or service) and one for test cases.

[2]http://www.oracle.com/technetwork/java/javacard/index.html

[3]Operational QVT Language for model to model transformations, http://www.eclipse.org/m2m/

Code for each component like terminal or smartcard is generated into an own independent package using XPand[4]. A modular plugin-system is needed for test cases to be able to interact with (emulated) components in order to check constraints at run-time or modify the protocol flow.

This is achieved by extending generated component code with generic interfaces for plugin-hooks. *Gate*-classes are introduced as a concept and are used to hook into the message-handling routines using those interfaces. Such gates are able to check any given list of constraints at every protocol step of each component, and also to read and modify their sent messages, emulating a Dolev-Yao-attacker. It is possible to hook and unhook gates at any given moment, allowing us to generate test case-independent and still deployable component-code.

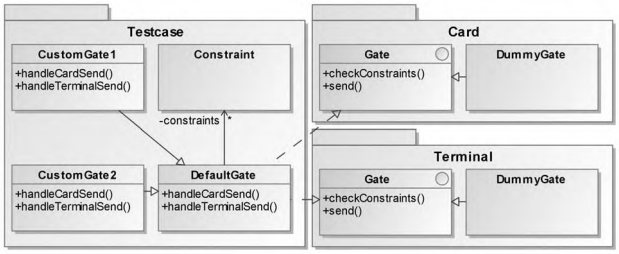


Fig. 3. Custom `Gate` class hierarchy

Fig. 3 shows a draft of the used gate class hierarchy. A `DefaultGate` implements each component's `Gate` interface and can be used instead of its default instance of `DummyGate`, a dummy class used only as a placeholder to forward component's sent messages to the intended receiver. `DefaultGate` optionally contains a list of constraints, is used by functional test cases and can be extended by custom gates, which are used to implement protocol modifications. Each `CustomGate` implements an attack on a protocol by providing methods to intercept and modify, inject or redirect messages as modeled in the corresponding attack sequence. Those methods extend the default `send`-methods of the components.

Each constraint in the test case models defining invariants on protocol runs is used to generate a separate constraint class. This class contains the condition check, implemented as a JUnit assert, which in turn is derived from the MEL statement inside the modeled constraint. Such invariants can reference component attributes as well as test case variables and are thus automatically initialized with the current test case context, i.e. they contain a copy of all variables declared prior to the protocol call.

MEL as a language as well as appropriate model transformations had to be extended to support test case specific features. Since test cases should be able to reference classes

[4]XPand: statically-typed template language for model to text transformations, http://www.eclipse.org/modeling/m2t/?project=xpand

from any component package (e.g. for initialization purposes), a Java-like package referencing has been implemented using component class names as implicit packages.

Test cases are generated into a test case package and can be run comfortably as JUnit-tests; smartcard code is run inside an emulator to speed up the testing process.

V. Related Work

Model-driven testing is a well-researched area with a variety of available implementations, none of which were found to be optimally compatible with SecureMDD. Bouquet et al. [7] introduce their own guidelines for modeling applications in UML and OCL, which allow test cases to be generated automatically from the application model. However, they do not consider security testing of protocols.

Sensler et al. [8] introduce an approach which focuses on designing and generating functional test cases for web applications. Although the underlying platform only considers functionality tests, some features of test case modeling like defining reusable subprocedures and keeping the test cases modular inspired our own test case design guidelines.

Fourneret et al. [9] focus on defining security properties for models of smart card applications, verifying those models using the UMLsec approach [10]. Those properties are then used to generate appropriate test cases for the application. This work, however, does not focus on modeling attacks on cryptographic protocols.

Jürjens et al. [11] (see also [12]) present work for tool-supported, systematic generation of security test cases for the implementations of formally specified security systems, but leave the development of a test case specification language such as presented in this paper as future work.

Notably, protocols and application-specific properties like "card balance can only be increased by the value of inserted money into the terminal" often employ arithmetics on integers. Thus, model-checking them may be infeasible due to the resulting large search space, even if supported by some model-checking tools like AVANTSSAR [13]. Generation of test cases based on the results of such tools as proposed in [14] or [15] would therefore suffer from the same limitation. This suggests the use of interactive verifiers as well as testing techniques such as proposed in this paper.

VI. Conclusion and Outlook

Defining and verifying security attributes of applications is an important part of the development process with SecureMDD, as shown in our previous work [16]. With the new modeling guidelines presented in this paper it is now easily possible to also test those attributes and to exploit found vulnerabilities by designing appropriate test cases. By testing the actual, deployable codebase we provide a testing environment close to the real world. Thus, by introducing and executing test cases in the early stages of application modeling, the modeler is able to detect errors and security vulnerabilities before the (optional) verification phase of application development with the SecureMDD approach. Since the verification itself can become a lengthy task depending on the size and complexity of the application even when using the tightly integrated interactive verifier KIV, it is important to allow quick test case modeling and execution and provide clear and understandable results.

In future, we plan to additionally support model checking tools in order to check standard security properties; it might then also be feasible to generate security test case models using model checking results, in order to generate testing code for the actual implementation.

References

[1] N. Moebius, K. Stenzel, H. Grandy, and W. Reif, "'SecureMDD: A model-driven development method for secure smart card applications'," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, march 2009, pp. 841 –846.

[2] ——, "Model-driven code generation for secure smart card applications," in *Software Engineering Conference, 2009. ASWEC '09. Australian*, april 2009, pp. 44 –53.

[3] Oracle, "Java card platform specification 2.2.2," available at http://java.sun.com/javacard/specs.html.

[4] N. Moebius, K. Stenzel, and W. Reif, "Formal verification of application-specific security properties in a model-driven approach," in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science, F. Massacci, D. Wallach, and N. Zannone, Eds. Springer Berlin / Heidelberg, 2010, vol. 5965, pp. 166–181. [Online]. Available: "http://dx.doi.org/10.1007/978-3-642-11747-3_13"

[5] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel, "KIV 3.0 for Provably Correct Systems," in *Proc. Int. Wsh. Applied Formal Methods*, ser. LNCS, D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, Eds., vol. 1641. Springer, 1999, pp. 330–337.

[6] D. Dolev and A. Yao, "On the security of public key protocols," *Information Theory, IEEE Transactions on*, vol. 29, no. 2, pp. 198 – 208, 03 1983.

[7] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," in *Proceedings of the 3rd international workshop on Advances in model-based testing*, ser. A-MOST '07. New York, NY, USA: ACM, 2007, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1291535.1291545

[8] C. Sensler, M. Kunz, and P. Schnell, "Test automation with model-driven test script development," *OBJEKTspektrum*, 03 2006.

[9] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jürjens, and P. Yousefi, "Model-based security verification and testing for smart-cards," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, aug. 2011, pp. 272 –279.

[10] J. Jürjens, *Secure Systems Development with UML*. Springer Berlin / Heidelberg, 2005. [Online]. Available: http://dx.doi.org/10.1007/b137706

[11] J. Jürjens and G. Wimmel, "Formally testing fail-safety of electronic purse protocols (extended abstract)," 2001.

[12] J. Jürjens, "Model-based security testing using umlsec: A case study," *Electr. Notes Theor. Comput. Sci.*, vol. 220, no. 1, pp. 93–104, 2008.

[13] D. von Oheimb and S. Mödersheim, "ASLan++ — a formal security specification language for distributed systems," in *Formal Methods for Components and Objects, FMCO 2010, Graz, Austria*, ser. LNCS, B. Aichernig, F. de Boer, and M. Bonsangue, Eds., vol. 6957. Springer, Dec. 2010, pp. 1–22, http://ddvo.net/papers/FMCO2010.html.

[14] M. Büchler, J. Oudinet, and A. Pretschner, "Security mutants for property-based testing," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, M. Gogolla and B. Wolff, Eds. Springer Berlin / Heidelberg, 2011, vol. 6706, pp. 69–77. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21768-5_6

[15] A. Armando, R. Carbone, L. Compagna, K. Li, and G. Pellegrino, "Model-checking driven security testing of web-based applications," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, april 2010, pp. 361 –370.

[16] N. Moebius, K. Stenzel, and W. Reif, "Pitfalls in formal reasoning about security protocols," in *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*. IEEE, 02 2010, pp. 248 –253.