Generating Security Tests in Addition to Functional Tests*

Jacques Julliand, Pierre-Alain Masson, Régis Tissot
LIFC — Laboratoire d'Informatique de l'Université de Franche–Comté
16, route de Gray F–25030 Besançon, France
{julliand, masson, tissot}@lifc.univ-fcomte.fr

ABSTRACT

This paper is about generating security tests, in addition to functional tests previously generated by a model-based testing approach. The method that we present re-uses the functional model and the adaptation layer developed for the functional testing, and relies on an additional security model. We propose to compute the tests by using some test purposes as guides for the tests to be extracted from the models. We see a test purpose as the combination of a security property and a test need issued from the know-how of a security engineer. We propose a language based on regular expressions for the expression of such test purposes. We illustrate our approach with experiments on IAS.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Security

Keywords

Model Based Testing, Test Purposes, Security Properties, Test Needs

1. INTRODUCTION

Generating smart tests for security policies is a challenging task, which is not fully addressed by nowadays test generation techniques. We consider in this paper some security properties expressed w.r.t. a system, and our intention is to ensure that these security properties are specifically tested. We focus on access control properties.

*This work is partially funded by the French National Research Agency ANR (ANR-05-RNTL-01001) and the Région Franche-Comté.

AST'08, May 11, 2008, Leipzig, Germany.

We are in a model based testing (MBT) framework [10]. A formal functional model of the system is written, from which tests are extracted by means of a *selection criterion*. As the model is an abstraction of the implementation under test (IUT), the tests are concretized by means of an adaptation layer which translates each operation call of the model into an executable script on the IUT. The verdict of the tests is obtained by comparing the results of the IUT with the ones predicted by the model. These tests directly activates the targeted functionality. They are *functional* tests. Testing the security of a system requires more than functional testing. The system must be proved to remain secure against tortuous scenarios of use or attack.

Our contribution in this paper is twofold. Writing a formal model and an adaptation layer is an important effort. We show that they can be re-used with little supplementary effort to compute additional *security* tests. Additionally, our security tests rely on some *test needs* expressed by security engineers w.r.t. security properties. They describe tortuous situations in which security could be violated.

We propose to write an additional model dedicated to security, which is much simpler to write than the full functional model, as it concentrates only on the security requirements. The tests are computed form the security model by using a *test purpose* as a selection criterion. We see test purposes as sequences of operation calls, that correspond to scenarios that exercise a security property according to a test need. The tests issued from the security model are then "replayed" on the functional model, to bring them to the same abstraction level as the functional tests. This allows re-using the existing adaptation layer to concretize the tests.

We formally describe a test purpose as a *test pattern*, and we present in this paper a language dedicated to the expression of such test patterns.

Our approach have been experimented in the framework of the french RNTL POSE project. We have generated tests for access control properties of IAS, a smart card platform.

The IAS is presented in Sec. 2. Section 3 describes our security testing process. Our language for describing test patterns is presented in Sec. 4. We compare our approach to related works and conclude in Sec. 5.

2. OVERVIEW OF IAS

This work was done in the framework of the french RNTL POSE[1] project, that brings together industrial (GEMALTO, LEIRIOS, SILICOMP/AQL) and academic (LIFC/INRIA

[1] `http://www.rntl-pose.info`

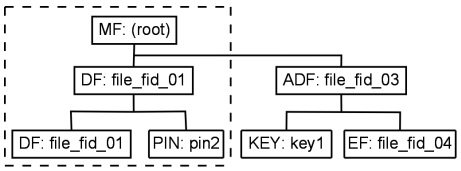

Figure 1: A sample IAS tree structure

CASSIS project, LIG) partners. The problematic is the conformity validation of a system to its security policy, especially for smart cards.

Experiments have been made with a real size industrial application, the IAS platform. Prior to the project, a functional model in B had been written by the LIFC and Leirios, from which functional tests had been computed and executed on an IAS implementation by Gemalto. We have completed these tests with security ones.

IAS stands for *Identification, Authentication and electronic Signature*. It is a standard for Smart Cards developed as a common platform for e-Administration in France, and specified [3] by GIXEL[2]. IAS provides services to the other applications running on the card.

IAS conforms to the ISO 7816 standard. The file system of IAS is illustrated with an example in Fig. 1. Files in IAS are either *Elementary Files* (EF), or *Directory Files* (DF), such as `file_fid_01` and `file_fid_02` in Fig. 1 . The file system is organized as a tree structure whose root is designed as MF (*Master File*). An application is assigned a specific DF on the card, which is called an ADF (*Application Directory File*).

The *Security Data Objects* (SDO) are objects of an application that contain highly sensible data such as PIN codes (see for example `pin2` in Fig. 1) or cryptographic keys, that can be used to restrict the access to some of the data of the application.

The services provided by the IAS module can be invoked by means of various APDU[3] commands.

Commands of IAS are for creating objects on the card, changing the life cycle state of these objects, setting the values of some attributes or navigating through the file system.

IAS responds to a command by means of a *status word* (i.e. a codified number), which indicates if the APDU command has executed correctly. If not, the status word codifies the error that occurred.

In addition to the existing B functional model of IAS, we have written a security model in B, focusing on the access control mechanisms. The functional model was approximately 15500 lines long, whereas our security model was 1000 lines long.

3. SECURITY PROPERTY BASED TESTING PROCESS

We illustrate in this section the concepts of security property, test need and test purpose, mentioned in Sec. 1. Then we describe our process for generating security tests.

[2]`http://www.gixel.fr` - it is the trade association in France for electronic components industries

[3]*Application Protocol Data Unit* - it is the communication unit between a reader and a card; its structure conforms to the ISO 7816 standards

3.1 Test Needs and Test Purposes

An access control *security property* for IAS states for example that to write inside a DF, a given access condition has to be true, otherwise the writing is refused. The functional tests will exercise the property in two separate situations: one where the writing succeeds, and one where it fails (possibly due to the access condition).

Security engineers want to test the property in other situations. For example, they think of the case when the access rule is first true and then becomes false. The *test need* is that the previous true value for the access rule has no side effect at the moment of writing.

A *test purpose* corresponding to this test need is to: *reach a state where the access rule is true; perform the writing operation*[4]*; reach a state where the access rule is false; perform the writing operation.*

This example illustrates that one often wants to express a test purpose as both states to be reached and operations to perform. We present in Sec. 4 a language for expressing test purposes by means of states and actions. The formalization of a test purpose is a *test pattern*.

In our process, we use a test pattern as a selection criterion to compute abstract test cases from the security model. An *abstract test* is a sequence of parameterized operation calls from a (functional or security) model. The parameters are instantiated and the test indicates the expected result of each operation call, thus providing an oracle for the concrete tests executed on the IUT.

We need a test generation tool able to compute sequences of operation calls with instantiated parameters for either reaching a particular state of the model, or enabling a particular operation of the model. We have used the LTG test generation solution [4] from Leirios, and we have used the test patterns as guides from which the tests have been computed.

3.2 Process for Generating Security Tests

Our process for generating security tests uses an operational security model as an oracle and a test purpose as a selection criterion. The process is made of four steps as shown in Fig. 2:

1. *generation of a set of symbolic abstract security tests SST* from a test purpose TP and the Security Policy Model SPM. The security parameters are instantiated, while the purely functional parameters remain abstracted.

2. *functional valuation of the symbolic abstract security tests into SAVT* from SST and the Functional Model FM. The conformance between the functional and security status words is checked w.r.t. a mapping R.

3. *concretization of the tests from SAVT into SCT* thanks to an Adaptation Layer AL which maps the operations and data of the model to the operations and data of the IUT.

4. *execution of each concrete test of SCT on the IUT.* The verdict is given by comparing the outputs from the IUT and the ones predicted from the model.

[4]this is for making sure that before the loss of the right to write, the writing operation was indeed possible, and not refused for any other reason.

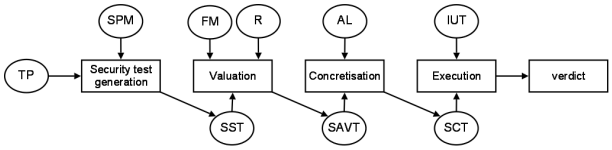

Figure 2: Security tests generation process

This process completes the model-based generation of the functional tests. We re-use the FM, the AL and the execution ground installation of the concrete tests. The security engineer has to design the SPM and the test purposes TP. He is only concerned by the security policy specification, and does not need to know the remainder of the functional specification.

4. LANGUAGE FOR TEST PATTERNS DESCRIPTION

In this section, we introduce the language that we have designed to formally express the tests purposes as test patterns. It is structured as three different layers: *model*, *sequence*, and *test generation directive*.

The *model layer* makes the language generic w.r.t. the system modelling language, by describing the operation calls and the state properties in the terms of the SPM. The *sequence layer* is based on regular expressions and allows to describe test scenarios as sequences of operation calls leading to target states. The *test generation directive layer* allows to specify some coverage criteria for the test generation tool.

4.1 Syntax of the Model Layer

It is given in Fig. 3. The rule `SP` describes access conditions as state predicates over the state variables of the SPM (expressed directly in the modelling language). The rule `OP` allows to describe the operation calls:

```
OP     ::= operation_name
         | $OP
         | $OP "\{"OPLIST"}"
OPLIST ::= operation_name
         | operation_name","OPLIST
SP     ::= state_predicate
```

Figure 3: Syntactic Rules for the Model Layer

- either by indicating which operation is called,
- or by the token `$OP` meaning "any operation is called",
- or by `$OP\{OPLIST}` meaning "any operation is called but one from the list `OPLIST`".

4.2 Syntax of the Test Generation Directive Layer

This part of the language is given in Fig. 4. It allows to specify guidelines for the test generation step. We propose two kinds of directives.

The rule `CHOICE` introduces two operators denoted as | and ⊗ for covering the branches of a choice. Let S_1 and S_2 be two test patterns. The pattern $S_1 \mid S_2$ tells the test generator to generate tests for both the pattern S_1 and the pattern S_2. $S_1 \otimes S_2$ tells the test generator to generate tests for either the pattern S_1 or the pattern S_2.

```
CHOICE ::="|" | "⊗"
OP1    ::= OP | "["OP"]"
```

Figure 4: Syntactic Rules for the Test Generation Directive Layer

The rule `OP1` tells the test generator to cover one of the behaviours[5] of the operation `OP`. It is the default option. The test engineer can also ask for the coverage of all the behaviours of the operation by surrounding its call with brackets.

4.3 Syntax of the Sequence Layer

This part of the language is given in Fig. 5. The rule `SEQ` is for describing a sequence of operation calls as a regular expression.

```
SEQ    ::= OP1 | "("SEQ")" | SEQ"⇝("SP")"
         | SEQ "." SEQ
         | SEQ REPEAT
         | SEQ CHOICE SEQ
REPEAT ::= "*" | "+" | "?"
         | "{"n"}" | "{"n",}" | "{,"n"}" | "{"n","m"}"
```

Figure 5: Syntactic Rules for the Sequence Layer

A step of a sequence is either an operation call as denoted by `OP1` (see Fig. 4) or an operation call that leads to a state satisfying a state predicate, as denoted by `SEQ ⇝(SP)`.

Sequences can be composed by the concatenation of two sequences, the repetition of a sequence or the choice between two sequences. We use the usual regular expression repetition operators, augmented with bounded repetition operators (exactly n times, at least n times, at most n times, between n and m times).

4.4 Test Pattern Example

We exhibit one of the test patterns written for the experimentation of our approach. The property to be tested is "*to access an object protected by a PIN code, the PIN must be authenticated*". The test need is "*we want to test this property after all possible ways to lose an authentication over a PIN*".

The test pattern has been written in two stages: the initialization stage and the core testing stage. The initialization stage simply aims at building the data structure required on the card to run the test. We don't give it here.

Figure 6 shows the core testing stage, describing the test purpose that combines the security property and the test need in three steps. First, the pattern describes all the possible ways for losing the authentication (for instance, a failure of the `VERIFY` command or a reset of the retry counter) over the PIN `pin2`[6]. The aim of the second step

[5]Every branch of an operation described as a control structure (such as a conditional structure) is called a *behaviour* of the operation.

[6]The B variable `pin_authenticated_2_df` (∈ `PIN_ID` ↔

is to select the DF `file_fid_02` (with the command `SELECT_FILE_DF_CHILD`) in order to apply an access command inside of it. The final step of the test pattern describes the application of the access commands inside the `file_fid_02` DF (for instance for creating a new DF with the command `CREATE_FILE_DF`) to test the access conditions.

```
(VERIFY | CHANGE_REFERENCE_DATA
| (RESET . SELECT_FILE_DF_CHILD) | RESET_RETRY_COUNTER
| (SELECT_FILE_DF_PARENT . SELECT_FILE_DF_CHILD))
  ⇝(current_DF = file_fid_01 ∧ file_fid_01 ∉ pin_authenticated_2_df[{pin2}])
. SELECT_FILE_DF_CHILD
  ⇝(current_DF = file_fid_02)
.[ CREATE_FILE_DF|DELETE_FILE | ACTIVATE_FILE | DEACTIVATE_FILE
| TERMINATE_FILE_DF | PUT_DATA_OBJ_PIN_CREATE ]
```

Figure 6: Example of a test pattern

5. CONCLUSION

We have presented in this paper a method for generating security tests in a model based testing context. We re-use existing functional model and adaptation layer. We write another model dedicated to security.

A test purpose, combining a security property and a test need, is used for extracting interessant test cases from the security model. The tests are then automatically replayed on the functional model and concretized by means of the adaptation layer.

We also have presented a language for expressing test purposes, which relies on regular expressions.

The method have been experimented on a real size industrial application, the IAS platform for smart cards. We have experimented three different test patterns, which gave a total of 183 tests that have been run on the IAS implementation. The three patterns were for testing: the different ways to lose the authentication over a PIN object (our example in this paper); the interpretation of the access conditions based on PIN authentication; the effect of life cycle changes on the authentication over a PIN object.

The results of the experimentation are encouraging, since the execution of the tests revealed some known differences of interpretation of the specification by the developers and the writers of the model. Moreover, the tests that we have generated are not redundant with the tests computed from the FM without the test patterns.

The TGV approach [5], and works from the Vertecs project[7] [9, 2] use explicit test purposes to extract tests from specifications, both given as Input/Output Symbolic Transition Systems (IOSTS). Our approach is methodologically different. Our intention is to re-use existing material, produced previously for using model-based testing. Despite the strong hypothesis that this material already exists, this makes our approach a ready to use industrial methodology when it is the case. The language we use to express the test purposes can be instantiated with various modelling languages. We have experimented it with B and in UML/OCL.

In [6], the authors show how tests dedicated to exercise a given security policy can be obtained by reusing functional tests. In comparison, we do not reuse the existing functional tests, but we augment them with security tests, independent from the functional ones. What we reuse is the existing functional material (i.e. the functional model and the adaptation layer). Our approach fits in what they call an independent strategy.

Also, as a difference with the above cited approaches, we have showed in a previous work [8] how the test purposes can be automatically computed, by modelling the test needs as syntactic transformation rules that transform regular expressions. The tool Tobias [7], that unfolds in a combinatorial way tests expressed as regular expressions, could be used to unfold our test patterns.

We are currently working at identifying and writing such transformation rules, based on the IAS case study. This work needs to be developed by studying many other case studies, in order to produce rules sufficiently generic to be applicable to a variety of examples. Rules could also be automatically deduced from the syntactic expression of a property, as suggested by [1] for properties expressed in JTPL, a temporal logic for JML.

`DF_ID`) associates to a pin identifier the DF identifiers where the PIN object is authenticated.

[7] http://www.irisa.fr/vertecs/

6. ACKNOWLEDGEMENTS

We would like to thank all the partners of the POSE ANR project, namely Gemalto, Leirios, Silicomp/AQL, LIG and INRIA for their great role in this work.

7. REFERENCES

[1] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 225–239. Springer, 2006.

[2] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, Aug. 2007.

[3] GIXEL. *Common IAS Platform for eAdministration*, Technical Specifications, 1.01 Premium edition, 2004. http://www.gixel.fr.

[4] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated test generation from B models. In *B'2007*, volume 4355 of *LNCS*, pages 277–280. Springer, 2007.

[5] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfert*, 7(1), 2005.

[6] Y. Le Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *ISSRE'07*, pages 93–102, 2007.

[7] Y. Ledru, F. Dadeau, L. Du Bousquet, S. Ville, and E. Rose. Mastering combinatorial explosion with the TOBIAS-2 test generator. In *ASE'07*, pages 535–536. ACM, 2007.

[8] P.-A. Masson, J. Julliand, J.-C. Plessis, E. Jaffuel, and G. Debois. Automatic generation of model based tests for a class of security properties. In *A-MOST'07*, pages 12–22. ACM Press, 2007.

[9] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verification to safety testing. In *TestCom'04*, volume 2978 of *LNCS*. Springer, 2004.

[10] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.