Security Mutants for Property-Based Testing

Matthias Büchler, Johan Oudinet, and Alexander Pretschner

{buechler,oudinet,pretschner}@kit.edu
Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany

Abstract. The last decade has witnessed impressive progress in terms of dedicated approaches to formally analyzing security properties of models. However, related approaches to generating tests generally rely on purely syntactic test selection criteria. In this paper, we consider models of protocols and describe an approach to generate tests from security properties. Security-specific mutation operators are defined and used to introduce potential security-specific leaks into the model. Then, if the leak is confirmed by a model analyzer, a test case for the security property is generated. We present examples for security-relevant mutants at the model level and show how they correspond to security-flawed implementations, thus providing evidence that model-level mutants are indeed useful for doing security testing.

Keywords: Property-based testing, Security protocols, Mutation, Test generation

1 Introduction

One of the main challenges in testing systems is the selection of test cases. Ideally, test cases unveil potential and actual failures, are cheap to generate and run, make it easy to track down underlying faults, and make it possible to assess their quality. Accordingly, there is a plethora of test selection criteria. Among them random criteria as well as criteria that reflect coverage of the implementation's or model's structure are particularly appealing because they lend themselves to the automated generation of test cases. Both kinds of criteria usually do not take into account properties of the system (see Section 3 for a discussion of exceptions). This situation is not entirely satisfying when it comes to characteristics of a system that pertain to a system's performance or security.

Many security properties of protocols can today be formally verified at the model level. Once a model has been verified w.r.t. confidentiality, integrity, or availability properties, the natural next step is to derive tests from this correct model for some implementation. In terms of structural coverage criteria, one approach to doing so is to pick a set of properties Φ that reflects the coverage criterion (e.g., *it is possible to reach state* σ, for all states σ), and use a model checker to generate a test case per property (that drives the model into state σ). Unfortunately, this simple approach does not work for non-structural universal properties like the aforementioned security properties. Let ψ be such a property. It is then reasonable to assume $\mathcal{M} \models \psi$—the model should satisfy the property. Using a model checker to verify $\mathcal{M} \models \neg\psi$ then yields *all* traces of the model as counter examples (test cases); no discrimination between more or less "interesting" traces takes place. In this paper, we tackle the problem of automatically generating test cases that are "interesting" when it comes to testing a specific security property according to potential implementation vulnerabilities. Indeed, we provide evidence that our generated test cases are correlated to targeted flaws at the source code level, which is not addressed by related work that focus on security properties only and leave out the implementation level.

Hence, rather than simply negating the property, we follow a different approach (that others have followed as well, see Section 3). Intuitively, we modify the *model* in a way that the security property is violated in a specific way, related to a common mistake at the implementation level. More specifically, we describe the following process. We start from a model, $\mathcal{M}$, of the system in which a specified property, φ, holds: $\mathcal{M} \models \varphi$. Then, we apply mutations to the model in order to get a new model $\mathcal{M}'$ in which the property doesn't hold anymore: $\mathcal{M}' \not\models \varphi$. In this way, we obtain one or several traces of $\mathcal{M}'$ that are counter-examples for φ, also called attack traces.

An attack trace describes a sequence of messages that an intruder should be able to play entirely in order to prove the presence of a real flaw at the implementation level. If an attack trace can be reproduced on the implementation, then we have found a real security flaw. Moreover, this flaw is related to a specific security property and a specific vulnerability at the implementation level, according to the process used to generate these test cases.

In sum, the problem that we tackle is the derivation of test cases for specific universal security properties. Our solution is based on a set of mutation operators at the model level that are related to possible errors at the source code level. We consider our main contribution in providing premature evidence for the correlation between High Level Protocol Specification Language (HLPSL) mutants and implementation-level vulnerabilities. We complement the procedure for testing non-structural universal properties [5,23], in a sense that if a flaw is found, we can boil it down to a specific vulnerability related to the mutant involved.

In the remainder of this paper, we will focus on mutation operators for models of protocols described in HLPSL [19]. In Section 2, we present two mutation operators and explain what kind of real implementation-level security problems they reflect. In Section 3, we review related work and conclude in Section 5.

2 HLPSL Mutation Operators

The Avispa tool [3] is used for automated validation of Internet security protocols and applications. It provides a dedicated language for protocol specifications called High Level Protocol Specification Language (HLPSL) which is used to specify protocols and their security properties. A model checker checks if the protocol model satisfies the specified properties. If security properties are violated the model checker outputs an attack trace that violates the property. To find attack traces for implementations on the basis of a correct model, we use mutation-based testing. The selection criterion for such mutants is the positive correlation with common errors at the implementation level. Before describing the mutant operators, we give a short description of the HLPSL input language and the output of the Avispa tool.

HLPSL Input Language. A HLPSL example specification of the NSPK-fix protocol[1] can be found in Appendix A. A security protocol specified in HLPSL is based on the A-B notation and is role based. It usually consists of several sections: (1) basic role specifications for protocol participants (line 1-37), (2) a session section where multiple participant roles are instantiated (line 40-46), (3) an environment section where sessions are combined and the intruder knowledge is defined (line 50-63), and (4) a goal section for security properties (line 67-71). In general a basic role defines transitions (line 10-17,30-36) that usually describe the receipt of a message and the sending of a reply. To do so it specifies preconditions and actions that have to be executed when the preconditions are true.

Avispa Tool Output. The Avispa tool defines an output format OF which is described as follows: The result of the tool is either *conclusive* or *inconclusive* where in the first case either $safe$ or $unsafe$ is reported. If the tool finds an attack (the tool reports $unsafe$) parts of the output consists of the name of the violated security property and an attack trace which shows the exchange of messages between participants in order to violate the property. For example, in the NSPK protocol the attack trace looks like as follows:

```
ATTACK TRACE:
    i -> a: start
    a -> i: {Na(1).a}_ki
    i -> b: {Na(1).a}_kb
    b -> i: {Nb(2).Na(1)}_ka
    i -> a: {Nb(2).Na(1)}_ka
    a -> i: {Nb(2)}_ki
```

It is a sequence of messages, such that X sends m to Y is represented by:

```
X -> Y: m
```

The agent i is a special agent — the intruder — that behaves as defined by the attacker model. The Avispa tool only defines the Dolev Yao intruder model, specified as a channel parameter (e.g. line 1 or 21 in Listing 1.1).

2.1 Agent Identifier Mutant

A HLPSL specification specifies a protocol and security properties like secrecy or authentication. Security flaws are often based on man-in-the-middle attacks where a message from a session can be used in another session and therefore violate

[1] This example comes from `http://avispa-project.org/library/NSPK-fix.html`

specific security properties of the protocol. By modifying agent IDs in the HLPSL specification, one may produce test cases that are related to man-in-the-middle attacks in particular and the violation of the secrecy property in general.

The HLPSL language supports send and receive statements for messages. To generate such test cases, we consider variables in receive statements like the following: $RCV(\{A.B'\}_K)$. It specifies that received messages are encrypted with key K and consist of two concatenated values A and B. If a message is received, the primed variable B is bound to the corresponding value in the message. The non-primed variable A already has been bound to a value and operates as a selector value. E.g. using the above receive statement at the left side of a transition, the receive statement is only triggered if the incoming message matches with the value of variable A. Therefore primed and non-primed variables can allow or prevent man-in-the-middle attacks at the HLPSL level.

Violation of Authentication and Secrecy. We apply the previous idea to the first message of the Needham-Schroeder Public-Key (NSPK) protocol. Correctly, Bob only accepts $\{Na'.A\}_{Kb}$ from Alice in a session if A corresponds to the intended sender of that session — due to the unprimed selector variable A in $RCV(\{Na'.A\}_{Kb})$ and the sharing of the agent identifier A when a session between Alice and Bob is defined. To invalidate this check, the mutant primes variable A so that A in $RCV(\{Na'.A'\}_{Kb})$ now adapts its value to the value in the received message. This means, an intruder can successfully use message $\{Na.A\}_{Kb}$, originated from a session Alice$\leftrightarrow$Bob, in a session intruder$\leftrightarrow$Bob.

Checking the modified model the Avispa tool returns the following attack trace. An expression $X \rightarrow Y : m$ means that X sends m to Y. The partial trace shows that in the third step, the intruder i can forward the message to Bob. Because Bob doesn't check the agent ID, it accepts the message in the session intruder$\leftrightarrow$Bob and sends back an answer encrypted with key k_i, instead of k_a.

```
ATTACK TRACE:
    i -> a: start
    a -> i: {Na(1).a}_kb
    i -> b: {Na(1).a}_kb
    b -> i: {Na(1).Nb(2).b}_ki
```

2.2 Nonce Mutant

In the NSPK protocol both Alice and Bob are creating nonces which are sent to the other partner. Nonces are generally used to guarantee the freshness of messages and that an agent is present in a session. Both are security properties and the latter can be specified with the keyword *authentication_on* together with (w)*request* and *witness* in a HLPSL specification. Therefore modifying nonces may affect the *authentication* property. The following mutant modifies the HLPSL model in such a way that the generated attack trace exactly addresses the part of the source code that deals with the *authentication_on* security property.

Violation of Authentication. Alice uses the nonce Na in the first message to verify that Bob participates in the current session. She expects that the first reply from Bob contains Na too. The mutant in this section replaces $RCV(Na.Nb'.B)$ by $RCV(Na'.Nb'.B)$ that means that Alice does not check the received nonce Na anymore. The Avispa tool indeed confirms that the mutant affects the authentication property of the protocol:

```
ATTACK TRACE:
    i -> a: start
    a -> i: {Na(1).a}_kb
    i -> a: {x238.x239.b}_ka
    a -> i: {x239}_kb
```

This attack trace for the modified protocol shows that an intruder was able to finish the protocol with Alice although Alice thinks she is talking to Bob.

2.3 Mutant-Implementation Error-Correspondence

To show that the described mutants reflect common mistakes at source code level we consider the C implementation given in [11]. The security of the NSPK protocol is based on different checks. E.g. when Bob receives the first message from Alice, he has to check if variable A in message $\{Na.A\}_{Kb}$ is set correctly. Similarly when Bob replies to Alice, she needs to check if Na in message $\{Na.Nb.B\}_{Ka}$ is correct. It's crucial that these checks are performed at the implementation level as well. In our implementation Bob performs the above check if the sender ID is correct with an if statement given as follows:

```
if(strncmp(alice_msg.id,ALICE_ID,sizeof(ALICE_ID))) {...} else {...}
```

Similarly, Alice executes the following if statement to check Na:

```
if(strncmp(alice_msg.nonceA,nonceA,sizeof(nonceA))) {...} else {...}
```

Therefore applying the above mutants at the HLPSL level corresponds to the case where the software developer has either (1) forgotten to implement the if statements, (2) has misplaced the if statements and therefore has made them ineffective, or (3) has messed up the conditions in the if statements.

3 Related Work

Our work is closely related to mutation testing [7,12]. The goal of mutation testing is to assess the effectiveness of test suites (or test selection criteria) which is done by introducing small syntactic changes into a program and then see if the test suite detects these changes. Two hypotheses underlie the generalizability of results obtained with mutation testing: the competent programmer hypothesis (programmers develop programs that are close to the correct version) and the coupling effect (small syntactic faults correlate with major faults). These assumptions have been subject to quite some validation research that is summarized in [12, Section II.A]. The idea of manually injecting real world faults into systems to the end of assessing the quality of quality assurance is common practice [20] for instance in the telecommunication industries; and so is fault-based testing where systems are checked for the occurrence of specific errors, e.g., stuck-at-1 errors in circuit designs. Security-related mutation testing has also been successfully performed at the level of access control policies [13,15,16,17]; we differ in that we consider protocol models rather than access control policies as a basis.

Test assessment criteria can also be understood as test selection criteria [24]. In our context, this means that mutation testing can also be used to generate rather than assess tests, an idea that was, among others, successfully applied for specification-based testing from AutoFocus or SMV models in the security context [23,1] and in the context of HLPSL [5]. These three papers are, in terms of related work, closest to our approach. Our work differs from them in that we provide evidence for the correlation between model-level mutants and implementation-level faults, instead of just a discussion about why implementations can go wrong. Model checkers have been used for test case generation since at least 1998 [2], in a variety of contexts that has been surveyed elsewhere [9]. Most of this work concentrates on generating tests that satisfy structural criteria such as state coverage, transition coverage, MC/DC coverage, etc., on the model. In this context, coverage criteria have also been successfully applied to temporal logic properties [21,8,22]. Our work differs in that we rely on a domain-specific fault model.

Formal models and model checking for security properties have been used by others [6,10,18,4,14]. They rely either on dedicated security rules that are used for test case generation, or are concerned with functional testing of security-critical subsystems. In contrast, our work is based on a dedicated fault model.

In practice, security testing is today usually performed with penetration testing tools (e.g., `http://sectools.org/`). These tools are different in that they do not rely on protocol models to perform the tests, and do not use model checking technology for the generation of tests.

4 Acknowledgments

This work was partially supported by the FP7-ICT-2009-5 Project no. 257876, “Secure Provision and Consumption in the Internet of Services” (`http://www.spacios.eu`).

5 Conclusion

We describe mutants at the HLPSL level that are closely related to implementation-level security faults. Actually, we showed on a C implementation which lines of code are addressed by the described mutants. One drawback of generating mutants at a higher-level language like HLPSL is that the number of generated mutants is rather small (see for example the experimental results in [5]). We are currently working on producing mutants at a lower-level intermediate language (i.e., Intermediate Format (IF)) in order to introduce more subtle changes that cannot be described at a higher level.

To be consistent with practices in security testing, we must facilitate the use of our test cases in a penetration testing tool. We are currently working on translating our test cases as exploits in Metasploit (`http://metasploit.com`). The main difficulties for this translation are: intercepting exchanged messages, filtering those that are relevant to the attack trace, building and sending messages from the intruder that are accepted by the honest agents.

References

1. P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *International Conference on Engineering of Complex Computer Systems*, pages 212–221. IEEE, 2001.
2. P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM*, pages 46–54. IEEE, 1998.
3. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *CAV*, volume 3576 of *LNCS*, pages 135–165. Springer, 2005.
4. A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In *TestCom/FATES*, pages 103–118, 2008.
5. F. Dadeau, P.-C. Héam, and R. Kheddam. Mutation-based test generation from security protocols in HLPSL. In *ICST*. IEEE, March 2011. To appear. 9 pages.
6. V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Test generation for network security rules. In *TestCom*, pages 341–356, 2006.
7. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program Mutation: A New Approach to Program Testing. In *Infotech State of the Art Report, Software Testing*, pages 107–126, 1979.
8. G. Fraser and F. Wotawa. Complementary criteria for testing temporal logic properties. In *TAP*, pages 58–73, 2009.
9. G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab.*, 19(3):215–261, 2009.
10. J. García-Alfaro, F. Cuppens, and N. Cuppens-Boulahia. Towards filtering and alerting rule rewriting on single-component policies. In *SAFECOMP*, pages 182–194, 2006.
11. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.
12. Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing, 2011. To appear in IEEE TSE.
13. Y. LeTraon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *ISSRE*, pages 93–102, 2007.
14. W. Mallouli, G. Morales, and A. Cavalli. Testing security policies for web applications. In *Proc. 1st workshop on security testing*, 2008.
15. F. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. 16th Intl. Conf. on the World Wide Web*, pages 667–676, 2007.
16. T. Mouelhi, Y. LeTraon, and B. Baudry. Mutation analysis for security tests qualification. In *Proc. 3rd Workshop on Mutation Analysis*, pages 233–242, 2007.

17. A. Pretschner, T. Mouelhi, and Y. LeTraon. Model-based tests for access control policies. In *ICST*, pages 338–347, 2008.
18. D. Senn, D. A. Basin, and G. Caronni. Firewall conformance testing. In *TestCom*, pages 226–241, 2005.
19. The AVISPA Team. *AVISPA User Manual*, 1.1 edition, 2006. `http://www.avispa-project.org/package/user-manual.pdf`.
20. J. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors.* John Wiley & Sons, 1997.
21. M. Weiglhofer, G. Fraser, and F. Wotawa. Using coverage to automate and improve test purpose based testing. *INFSOF*, 51(11):1601–1617, 2009.
22. M. W. Whalen, A. Rajan, M. Per Erik Heimdahl, and S. P. Miller. Coverage metrics for requirements-based testing. In *ISSTA*, pages 25–36, 2006.
23. G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. In *ICFEM*, pages 471–482, 2002.
24. H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29:366–427, December 1997.

A HLPSL model for the corrected version of Needham-Schroeder Public-Key authentication protocol

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))
played_by A def=
  local State : nat,
        Na, Nb: text

  init State := 0

  transition

    0.  State  = 0 /\ RCV(start) =|>
        State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)
                   /\ secret(Na',na,{A,B})
                   /\ witness(A,B,bob_alice_na,Na')

    2.  State  = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
        State':= 4 /\ SND({Nb'}_Kb)
                   /\ request(A,B,alice_bob_nb,Nb')
end role


role bob(A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))
played_by B def=
  local State : nat,
        Na, Nb: text

  init State := 1

  transition

    1.  State  = 1 /\ RCV({Na'.A}_Kb) =|>
        State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'.B}_Ka)
                   /\ secret(Nb',nb,{A,B})
                   /\ witness(B,A,alice_bob_nb,Nb')

    3.  State  = 3 /\ RCV({Nb}_Kb) =|>
        State':= 5 /\ request(B,A,bob_alice_na,Na)
end role


role session(A, B: agent, Ka, Kb: public_key) def=
  local SA, RA, SB, RB: channel (dy)

  composition
        alice(A,B,Ka,Kb,SA,RA)
     /\ bob  (A,B,Ka,Kb,SB,RB)
end role



role environment() def=
    const a, b          : agent,
          ka, kb, ki    : public_key,
          na, nb,
          alice_bob_nb,
          bob_alice_na  : protocol_id

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition
        session(a,b,ka,kb)
     /\ session(a,i,ka,ki)
     /\ session(i,b,ki,kb)
end role



goal
  secrecy_of na, nb
  authentication_on alice_bob_nb
  authentication_on bob_alice_na
end goal


environment()
```
Listing 1.1. HLPSL model of NSPK-fix