SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1531

Model-based mutation testing from security protocols in HLPSL

Frédéric Dadeau[1,*,†], Pierre-Cyrille Héam[1,2], Rafik Kheddam[3], Ghazi Maatoug[2] and Michael Rusinowitch[2]

[1]*Femto-ST institute/INRIA CASSIS, 16 route de Gray, 25030 Besançon, France, France*
[2]*INRIA Nancy Grand Est et LORIA, 615, rue du Jardin Botanique, 54602 Villers les Nancy, CEDEX, France*
[3]*Univ. Grenoble Alpes, LCIS, 26900 Valence,France*

SUMMARY

In recent years, important efforts have been made for offering a dedicated language for modelling and verifying security protocols. Outcome of the European project AVISPA, the high-level security protocol language (HLPSL) aims at providing a means for verifying usual security properties (such as data secrecy) in message exchanges between agents. However, verifying the security protocol model does not guarantee that the actual implementation of the protocol will fulfil these properties. This article presents a model-based testing approach, relying on the mutation of HLPSL models to generate abstract test cases. The proposed mutations aim at introducing leaks in the security protocols and represent real-world implementation errors. The mutated models are then analysed by the automated validation of Internet security protocols and applications tool set, which produces, when the mutant protocol is declared unsafe, counterexample traces exploiting the security flaws and, thus, providing test cases. A dedicated framework is then used to concretize the abstract attack traces, bridging the gap between the formal model level and the implementation level. This model-based testing technique has been experimented on a wide range of security protocols, in order to evaluate the mutation operators. This process has also been fully tool-supported, from the mutation of the HLPSL model to the concretization of the abstract test cases into test scripts. It has been applied to a realistic case study of the Paypal payment protocol, which made it possible to discover a vulnerability in an implementation of an e-commerce framework. Copyright © 2014 John Wiley & Sons, Ltd.

Received 4 September 2012; Revised 20 December 2013; Accepted 20 March 2014

KEY WORDS: mutation testing; security protocols; HLPSL; AVISPA; test generation

1. INTRODUCTION

In recent years, security has become an important challenge in software engineering. Security properties such as *secrecy*, *authentication* and data *integrity* are now more and more required in the systems and need extensive verification and validation efforts. Numerous approaches have been designed to formalise various security aspects, allowing their verification and validation, at the model level. This work is focused on security protocols that represent exchanges of messages (possibly encrypted, hashed, signed etc.) between actors. Such protocols aim at establishing a trustful communication link between actors. The automated validation of Internet security protocols and applications (AVISPA) European project[‡] (2003–2006) aimed at the automated verification of security protocols [1]. These protocols are described using a dedicated specification language named high-level protocol specification language (HLPSL) [2] and verified using four possible back-end tools, using various techniques (e.g. Constraint-Logic-based Attack Searcher (CL-AtSe) [3] uses a model checking approach, Tree Automata based on Automatic Approximations for the Analysis of

*Correspondence to: Frédéric Dadeau, Femto-ST institute/INRIA CASSIS, 16 route de Gray, 25030 Besançon, France.
[†]E-mail: frederic.dadeau@femto-st.fr
[‡]http://www.avispa-project.org.

Security Protocols (TA4SP) [4] uses tree automata and rewriting techniques), which make it possible to exhibit possible attacks on the protocol leading to the violation of a security property (exposure of a secret data, impersonation etc.). When an attack is found, the protocol is declared *unsafe*, and an attack trace leading to the violation of the security property is returned.

Even though security protocol models have been formally verified and proved correct, their implementation may still be faulty and may lead to severe damages. Recently, the OpenSSL package used in Debian-based operating systems (e.g. Ubuntu)[§] contained a bug in the random generator used to generate SSL and SSH keys. Such a vulnerability can be exploited using a man-in-the-middle attack that targets an SSH stream. The consequences were heavy, because all SSH certificates generated between September 2006 and May 2008 had to be revoked and regenerated after correcting the bug to prevent any attack. Another severe security flaw was discovered [5] in the SAML-based implementation of the single-sign-on protocol for Google Applications [6]: thinking it was useless, the developers chose to omit a field in the implementation. More recently, it is shown by Focardi *et al*. [7, 8] that PKCS#11 [9], a standard application programming interfaces (API) cryptographic device, may have security flaws, depending on how it is implemented. The importance of validating the code, and not only the model of the protocol, is thus crucial.

The approach presented in this article proposes a model-based testing technique that relies on the use of mutations of formal models to generate abstract test cases for security testing. Mutation testing [10] is a technique that consists in introducing faults into a programme or a model. Mutation testing is often used to evaluate the quality of a test suite [11], when introducing faults into a programme, but it can also be used to generate test cases [12], when introducing faults into a model. The principle of model-based mutation testing is to consider that the test generator has to produce a test that is able to reveal a particular defect illustrated by a given mutant. Mutation testing is thus a relevant technique for including faults (such as those mentioned previously) into security protocols.

When performing a model-based testing approach [13], it is mandatory to be able to animate (or explore) it, in order to compute test cases from the model. Thus, each execution trace extracted from the model produces a test case that can then be run on the system under test (SUT) after a possible adaptation of the abstract test case. In this work, the use of a model checker is considered, in order to produce the test cases. Model checking is an old technique [14] that consists in enumerating the states of a model in order to check the satisfaction or the violation of a given property. When the property is violated, the model checker is able to exhibit a counterexample, namely, a trace of the model execution that leads to an incorrect state. In the context of model-based testing, such a capability is used to produce test cases by providing test targets in terms of states to reach; the model checker computes a trace that can be used as a test case. Because the considered models are deterministic, a trace is sufficient for applying this technique.

The technique proposed here consists in performing mutations of security protocols written in HLPSL and aims at generating model-based test cases for validating implementations of security protocols. It relies on the principle of introducing realistic faults into a security protocol, using relevant mutation operators and, then, generating model-based test cases, using a dedicated model checker. The test generation approach is depicted in Figure 1. First, an HLPSL file, representing a safe protocol, is given as input to a mutant generator (named jMuHLPSL) producing a set of mutants. These latter are then analysed by the AVISPA tool in order to check if the mutation introduced a vulnerability. In that case, the AVISPA tool computes an attack trace that represents a test case. If the mutation keeps the protocol safe or produces an incoherent protocol, then no test case is produced.

The contributions of this paper are the following. First, a set of mutation operators dedicated to the mutation of security protocols are presented and discussed. Second, an extensive evaluation of these mutation operators is reported on a bench of 50 protocols, including a discussion on their relevance w.r.t. real implementation faults and an assessment of their combination. Third, a framework for the concretization and the execution of the abstract test cases is presented. Finally, the approach is illustrated on a real-world payment protocol, in order to show its relevance.

[§]http://www.debian.org/security/2008/dsa-1571.

DOI: 10.1002/stvr

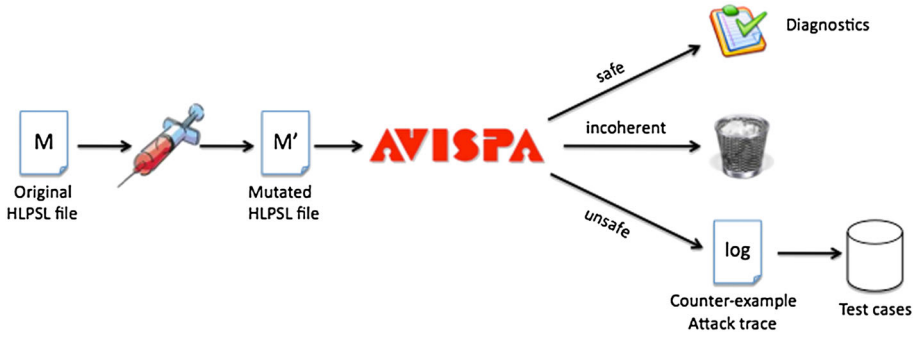

Figure 1. Test generation approach using mutation of models.

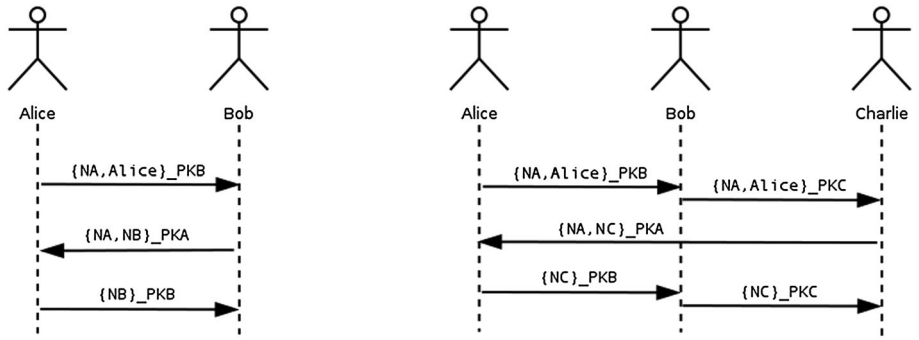

Figure 2. The NSPK protocol (left-hand side) and Lowe's attack on NSPK (right-hand side.)

The content of the article is organised as follows. Section 2 presents the basic principles of security protocols before introducing the HLPSL formalism for their description in Section 3. Then, the mutation operators are presented in Section 4. Section 5 describes the experimental assessment of the approach, namely, the tool that has been designed to produce the mutants, the application of the approach to a large set of existing security protocols and the evaluation of higher-order mutation (HOM). Then, Section 6 explains the correlation between the proposed mutation operators and real faults that can be found in an implementation. The concretization of the test cases is presented in Section 7. A case study, illustrating the full approach on the Paypal payment protocol, is described in Section 8. Finally, Section 9 presents the related work, and Section 10 concludes and presents the future work.

2. SECURITY PROTOCOLS

First some basic definitions and notations are defined. Participants to a protocol are called *agents*. Each execution of a protocol is called a *session*. An attacker is a particular agent called the *intruder*. Used random numbers are called *Nonce*. A message X encrypted by a key Y is denoted $\{X\}_Y$ or `{X}_Y`.

In a distributed communication environment, the security of communications lies on both cryptographic functions and security protocols. Cryptography is the science of producing robust, fast and safe encryption algorithms. Security protocols represent the way cryptographic primitives are exploited in practice to obtain safe communications in an open network. Lowe [15] showed that security protocols may be unsafe when the intruder is active on the network, even if cryptographic functions are unbreakable. One can classically illustrate it using the two agents NSPK protocol depicted in Figure 2 (left-hand side) in a sequence diagram[¶]. Agents are `Alice` and `Bob`, whose

[¶]In the rest of the article, these diagrams will be replaced by an Alice–Bob notation, that is, `A -> B : M` indicating that agent `A` sends to agent `B` a message `M`.

DOI: 10.1002/stvr

respective public keys are PKA and PKB. This is an authentication protocol: Alice [respectively, Bob] wants to be sure to communicate with Bob [respectively, Alice]. First, Alice sends to Bob a message containing its identity and a random number NA (a *Nonce*) and encrypted with the public key of Bob. When Bob receives this message, he can decrypt it using its private key. He composes a message with NA and a random number NB, encrypts it with the public key of Alice and sends it. Alice can decrypt it, checks whether the first number corresponds to NA and sends back NB to Bob. To finish, Bob checks whether this number corresponds to the one he sent during the second step. The idea of the protocol is the following: because the nonce NA is encrypted with the public key of Bob, only Bob can decrypt it, and when Alice receives it back, she knows that she is communicating with Bob. Similarly, when Bob receives NB, he knows he is communicating with Alice.

All messages transiting on the network are either encrypted by Alice's public key or Bob's public key. If the cryptography is safe, an intruder watching messages cannot obtain any information on neither NA nor NB. But assume now that Bob has malicious intentions and wants to impersonate Alice when discussing with another agent, Charlie. When Alice starts a protocol session with him, Bob can open a protocol session with Charlie, to break the security, in the way depicted in Figure 2 (right-hand side). When he receives the first message from Alice, Bob decrypts it, encrypts it with Charlie's public key and sends the encrypted message to Charlie. For Charlie, this message looks like a first message from Alice. Consequently, Charlie continues the protocol and sends the message {NA,NC}_PKA to Alice. Note that, at this stage, Bob does not know the value of NC. Alice receives this message from Charlie but thinks it was composed by Bob because it contains her nonce NA. Thus, Alice continues her session and sends {NC}_PKB to Bob. Now, Bob knows NC and can send it, encrypted with Charlies's public key PKC, to Charlie. At the end, Alice thinks she is talking with Bob (that is true), but Charlie thinks he is talking with Alice (that is untrue). This kind of protocol flaw, using interleaving of sessions, is called a *man in the middle attack*.

During last years, hundreds of papers were dedicated to the analysis of security protocols. The reader interested in *man in the middle attacks* and related questions can refer to the survey of Cortier *et al*. [16].

Notice that the present work does not claim that cryptographic functions are unbreakable. The goal is to test logical flaws in protocol implementations. Thus, the question of testing whether encryption schemes are safe or not is not targeted.

3. HIGH-LEVEL PROTOCOL SPECIFICATION LANGUAGE MODELS AND THE AVISPA TOOL SET

The HLPSL [2] is a specification language dedicated to the design of security protocols [17]. It is used by the AVISPA verification tool [1].

3.1. The high-level protocol specification language

The HLPSL is a TLA-like [18] state-transition-based specification language. The two main elements of HLPSL specifications are *basic roles* and *composed roles*. HLPSL is described in this section using the following example from the HLPSL tutorial [17], with three agents A, B and S:

```
A -> S : {Kab}_Kas
S -> B : {Kab}_Kbs
```

During the first step of the protocol A sends to S a key Kab encrypted with the symmetric key Kas. Next, S sends to B the received key Kab encrypted with the symmetric key Kbs. This is a simple protocol where S (the server) is a trusted party used for exchanging keys. The key Kas [resp. Kbs] is a secret key shared by A and S [respectively, B and S].

Each basic role describes a state-transition system corresponding to the behaviour of each agent. More precisely, each basic role describes which information the participant can use initially (parameters), how it starts (its initial state) and ways in which the state can change (transitions). For

DOI: 10.1002/stvr

```
role server(A, B, S : agent,
            Kas, Kbs : symmetric_key,
            SND, RCV : channel (dy))
played_by S def=
   local
      State: nat, Kab: symmetric_key
   init
      State := 0
   transition
      step1.
         State = 0 /\ RCV({Kab'}_Kas) =|>
         State':= 1 /\ SND({Kab'}_Kbs)
end role
```

```
role session(A,B,S: agent,
      Kas, Kbs : symmetric_key)

def=
  local SNDA, RCVA, SNDB, RCVB,
          SNDS, RCVS: channel (dy)

composition
  alice(A,B,S,Kas,SNDA,RCVA) /\
  bob(B,A,S,Kbs,SNDB,RCVB) /\
  server(S,A,B,Kas,Kbs,SNDS,RCVS)

end role
```

Figure 3. Example of basic (left column) and composed (right column) role.

```
role environment()  def=

    const A, B, S : agent, Kas, Kbs, Kis : symmetric_key

    intruder_knowledge = {A, B, S, Kis}

    composition
       session(A,B,S,Kas,Kbs) /\ session(A,I,S,Kas,Kis) /\ session(I,B,S,Kis,Kbs)

end role
```

Figure 4. Example of environment role.

instance, the role given in Figure 3 (left column) specifies the behaviour of the server. The parameter of this role are agent's names (`A`, `B`, `S`) describing who is playing the protocol and useful symmetric keys for the server (`Kas` and `Kbs`). Parameter `SND` and `RCV` describes the channels used to respectively send and receive messages (and `dy` means that theses channels are *Dolev-Yao* channels [19], which is totally unsafe channels). Several other parameter can be defined as *hash functions*. Then, the transition system representing the server behaviour is described. There are only two states, 0 and 1. When `S` is in state 0 (beginning of the protocol) and if it receives a message of the form $\{K_{ab}\}_{K_{as}}$ (denoted by `RCV(Kab'_Kas)`), then `S` passes to state 1 and sends the message $\{K_{ab}\}_{K_{bs}}$ to `B` (denoted by `SND(Kab'_Kbs)`). When describing transition systems, it is also possible to specify that a message is obtained using the exclusive-or operator and the notation `xor(_,_)`. Other features are possible, but they will not be used in the proposed mutation operators. For the described example, two other roles, for Alice and Bob, have to be defined.

Composed roles instantiate one or more basic roles, describing how they work together for a session of the protocol (and such roles are often called `role session`). Similarly to basic roles, composed roles have parameters (agents names, used keys) and local variables (channels names) but no description of a state-transition system. In composed roles, there is `composition` section in which basic roles are instantiated. In the example, it describes how Alice, Bob and the server's roles work together in parallel (Figure 3, right column).

Finally, there is a special composed role called `environment` specifying how the intruder interacts with other agents, describing which session he can play with other agents. This role, as shown in Figure 4, contains a special section `intruder_knowledge` specifying the initial knowledge of the intruder. In the example, the intruder knows the names of agents `A`, `B`, `S` and shares a key `Kis` with the server. Moreover, Alice, Bob and the server play a fair session (first part of the composition session). The intruder also plays a session with Alice and the server (playing Bob's roles) and a session with Bob and the server (playing Alice's roles).

Because it is useless for the work presented here, no other HLPSL features are described, especially how to specify secrecy/authentication properties. The interested reader can refer to the HLPSL reference manual [2] for more details.

3.2. The AVISPA tool set

The AVISPA toolkit [1] is composed of four back-ends dedicated to the analysis of security protocols. The tool is based on a first translation of the original HLPSL file into the intermediate

DOI: 10.1002/stvr

format (IF) language [20], which is then analysed using various techniques implemented in the underlying tools.

The on-the-fly model checker (OFMC) [21] performs protocol falsification and bounded verification by exploring the transition system described by an IF specification on demand. OFMC implements several correct and complete symbolic techniques. It supports the specification of algebraic properties of cryptographic operators and typed and untyped protocol models.

The CL-AtSe [3] applies constraint solving, with simplification heuristics and redundancy elimination techniques. CL-AtSe is built in a modular way and is open to extensions for handling algebraic properties of cryptographic operators. It supports type-flaw detection and handles associativity of message concatenation. This tool makes it possible to produce attack traces.

The SAT-based Model-Checker (SATMC) [22] builds a propositional formula encoding a bounded unrolling of the transition relation specified by the IF, the initial state and the set of states representing a violation of the security properties. The propositional formula is then fed to a state-of-the-art SAT solver, and any model found is translated back into an attack trace.

The TA4SP back-end [4] approximates the intruder's knowledge by using regular tree languages and rewriting. For secrecy properties, TA4SP can show whether a protocol is flawed (by under-approximation) or whether it is safe for any number of sessions (by over-approximation).

As the objective of this work is not only to analyse security protocols but also to produce attack traces, the TA4SP tool will not be considered in this approach, as it does not support this feature.

3.3. Counterexample generation

Usually, model checkers take as input a model and a property that one wants to verify on the system. They perform the exploration of the model (possibly up to a given depth) and check the satisfaction of the property in each state. If a violation of the property is discovered, the model checker returns an execution trace of the model, which does not satisfy the property.

The AVISPA back-ends that are considered here work similarly. They perform the exploration of the protocol, by constructing the transition system associated to each role, accordingly to the composition defined in the model. The properties appear in a dedicated clause of the HLPSL model, called `goal`. These properties can be either temporal properties (expressed in LTL) or security macros: `secrecy_of` X and (`weak`) `authentication_on` Y, in which X and Y are identifiers referring to on-the-fly verifications to be performed at a given step of a given role and of the following forms:

- `secret`(E, id, S), which declares information E to be secret between the agents in set S;
- `witness`(A, B, id, E), which declares, for a weak authentication of A by B on E, that agent A is witness for the information E;
- `[w]request`(B, A, id, E), which declares, for a strong [or weak] authentication of A by B on E, that agent B requests a check of the value E.

Figure 5 illustrates the declaration of a security goal. When reaching state 1 in the server role, the model checker will have to ensure the secret property specifying that data `Kab'` are secret between agents A and B. If, at some point, this verification fails, the model checker will return a trace representing the exchanges of messages between the agents leading to this particular state.

```
role server(...)
played_by S def=
     ...
      step1.
         State = 0 /\ RCV({Kab'}_Kas) =|>
         State':= 1 /\ SND({Kab'}_Kbs) /\ secret({Kab'},sec1,{A,B})
end role

goal
     secrecy_of sec1
end goal
```

Figure 5. Example of a security goal in high-level protocol specification language.

DOI: 10.1002/stvr

4. MUTATION OPERATORS FOR HIGH-LEVEL PROTOCOL SPECIFICATION LANGUAGE MODELS

This section introduces several mutations for HLPSL specifications. These mutations are motivated by known flaws on protocols (see survey of Cortier *et al*. [16, 23] for more information and pointers), which will be explained along with the considered mutation operators. The relevance of the mutation operators will be discussed in Section 6.

4.1. XOR mutation

It is frequent to encrypt a message M with a key K using the exclusive-or operator, that is, $\{M\}_K = M \oplus K$. This method is known as the *simple XOR cipher* and is used, for instance, in the wired equivalent privacy protocol [24]. Several man-in-the-middle attacks on protocols can be performed using particular properties of the `xor` operator. Consider, for instance, the three pass Shamir protocol [23] given in the following text:

```
Alice   -> Bob   :  M xor KA
Bob     -> Alice : (M xor KA) xor KB
Alice   -> Bob   :  M xor KB
```

Alice wants to secretly send a message `M` to Bob but does not share any secret with him. First, she sends `M xor KA` to Bob (who does not know `KA`). Bob computes `(M xor KA) xor KB` using his own key `KB`. Now, Alice computes `((M xor KA) xor KB) xor KA= M xor KB` (because `xor` is commutative and idempotent) and sends her message back to Bob. In the general case (without xor-encryption), several attacks are known on this protocol. But, using xor-encryption, some new ones can be pointed out. For instance, a passive intruder who just observes the network knows `M1 = M xor KA`, `M2 = (M xor KA) xor KB` and `M3 = M xor KB`. He can compute `M1 xor M2 xor M3`, which is equal to `M`, the secret message!

There are several other examples of protocols that are safe with a general encryption scheme but become flawed while using `xor` and its commutative/nilpotence properties. Therefore, the first mutation that is considered consists in replacing in a protocol specification each encryption by a xor-encryption. For instance, Figure 6 shows how the server role (initially shown in Figure 3, left column) is mutated using the xor-encryption mutation operator.

4.2. Homomorphism mutation

The *homomorphism property* characterises an encryption function satisfying, for every $X_1, X_2, K, \{X_1.X_2\}_K = \{X_1\}_K.\{X_2\}_K$: the encryption of the concatenation of two blocks is equivalent to the concatenation of the encryptions of the blocks. For instance, the *Electronic Code Book* (ECB) [23] has this property.

Consider for example the modified version of NSPK proposed by Lowe [15] and given in Figure 7, left column. The modification consists in adding Bob identity in the second message. It is possible to prove that there is no man-in-the-middle attack for this protocol. However, if the encryption scheme has the homomorphism property, the protocol becomes unsafe as shown in Figure 7, right column.

```
role server(A, B, S : agent,
            Kas, Kbs : symmetric_key,
            SND, RCV : channel (dy))
played_by S def=
  local
    State: nat, Kab: symmetric_key
  init
    State := 0
  transition
    step1.
      State = 0 /\ RCV({Kab'}_Kas) =|>
      State':= 1 /\ SND({Kab'}_Kbs)
end role
```

```
role server(A, B, S : agent,
            Kas, Kbs : symmetric_key,
            SND, RCV : channel (dy))
played_by S def=
  local
    State: nat, Kab: symmetric_key
  init
    State := 0
  transition
    step1.
      State = 0 /\ RCV(xor(Kab',Kas)) =|>
      State':= 1 /\ SND(xor(Kab',Kbs))
end role
```

Figure 6. Example of the server basic role with the `xor` mutation.

DOI: 10.1002/stvr

```
Alice -> Bob   :  {NA,Alice}_PKB
Bob   -> Alice :  {NA,NB,Bob}_PKA
Alice -> Bob   :  {NB}_PKB
```

```
Alice   -> Bob     : {NA,Alice}_PKB
Bob     -> Charlie : {NA,Alice}_PKC
Charlie -> Bob     : {NA,NC,Charlie}_PKA
Bob     -> Alice   : {NA,NC,Bob}_PKA
Alice   -> Bob     : {NC}_PKB
Bob     -> Charlie : {NC}_PKC
```

Figure 7. The NSPK-Lowe protocol (left column) and an attack on it using homomorphism (right col.)

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

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

Figure 8. NSPK-Lowe: Alice role and its mutation using homomorphism.

```
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
```

```
role environment() def=
  const a, b         : agent,
    ka, ki   : public_key,
    na, nb,
    alice_bob_nb,
    bob_alice_na : protocol_id
  intruder_knowledge =
      {a, b, ka, ki, inv(ki)}
  composition
    session(a,b,ka,ka) /\
    session(a,i,ka,ki) /\
    session(i,b,ki,ka)
end role
```

Figure 9. Environment role of NSPK-Lowe and its mutation using the public key operator.

The attack is similar to the one on NSPK where Bob is the intruder. The homomorphism property is used after the third line. Once Bob receives {NA,NC,Charlie}_PKA, he can decompose it into {NA,NC}_PKA and {Charlie}_PKA using the homomorphism property. Thus, computing {Bob}_PKA he can forge the message {NA,NC,Bob}_PKA by a simple concatenation. Figure 8, left column, describes Alice's role in HSPSL.

In order to model the homomorphism property, all encrypted messages are decomposed as much as possible. For example, Alice's role is modified according to Figure 8, right column.

4.3. Public key mutation

Protocols may be unsafe if both participants use the same public key. This kind of situation may happen, for instance, after a bad manipulation of `.ssh` files. The goal of the public key mutation is to detect this kind of sensitivity.

The mutation can be easily implemented on the HLPSL specification by modifying the environment role. For instance, Figure 9 shows the environment of the NSPL-Lowe protocol, initially given in Figure 4. One can express that both `a` and `b` have the same public key, it suffices to replace each occurrence of `kb` by `ka` in the `composition` section. We obtain the environment specification given in Figure 9, right column.

DOI: 10.1002/stvr

```
Alice -> Bob   :  {NA,Alice}_PKB              Alice -> Bob   :  {NA,Alice}_PKB
Bob   -> Alice :  {NA,NB,Bob}_PKA      ⇒      Bob   -> Alice :  {NA,NB,X}_PKA
Alice -> Bob   :  {NB}_PKB                    Alice -> Bob   :  {NB}_PKB
```

Figure 10. The NSPK-Lowe protocol and its mutated version.

```
role alice (A, B: agent,                    role alice (A, B: agent,
            Ka, Kb: public_key,                         Ka, Kb: public_key,
            SND, RCV: channel (dy))                     SND, RCV: channel (dy))
played_by A def=                            played_by A def=
  local State : nat,                          local State : nat,
        Na, Nb: text                                Na, Nb: text, X : agent
  init State := 0                             init State := 0
  transition                                  transition
    0.  State  = 0 /\ RCV(start) =|>            0.  State  = 0 /\ RCV(start) =|>
        State':= 2 /\ Na' := new() /\               State':= 2 /\ Na' := new() /\
        SND({Na'.A}_Kb)                             SND({Na'.A}_Kb)
    2.  State  = 2 /\                           2.  State  = 2 /\
        RCV({Na.Nb'.Bob}_Ka) =|>                    RCV({Na.Nb'.X}_Ka) =|>
        State':= 4 /\ SND({Nb'}_Kb)                 State':= 4 /\ SND({Nb'}_Kb)
end role                                    end role
```

Figure 11. NSPK-Lowe: Alice role and its mutation using the substitution mutation.

4.4. Substitution

Consider the safe modified version of NSPK given in Figure 10, left column. In order to send the third message of the protocol, Alice does not need to check, in the second message, whether it really comes from Bob. This verification is a semantic security condition that can be forgotten, resulting in the protocol described in Figure 10, where Alice just checks if the second message is well formed, that is, that `X` is present and/or well typed (if the protocol analysis tool supports typing in the protocols).

In this case, Lowe's attack still works on this version of the protocol. In order to catch this kind of bad implementation, a specific mutation is considered, that replaces computational useless information by a number `X`. For instance, Alice's role given in Figure 11, left column, is transformed into the role shown in the right column.

4.5. Hash functions mutation

In cryptography, hash functions are one-way compression functions: if h is a hash function, then for every x, $h(x)$ is smaller (encoding size) than x, and from $h(x)$, it is practically impossible to compute back x.

Hash functions are, for instance, frequently used for numerical signatures, as in the SSL protocol. They are also useful for numerical registrations: in order to safely register a source code c (without providing it) to a trust authority, one just has to register $h(c)$. This situation is illustrated with the following toy protocol, inspired by Zhou and Gollman [25].

```
Alice -> Server : Alice, {h(c)}_{PriKA}
Server -> Alice : {h({c})_{PriKA},T}_{PriKS}
```

Alice sends to a trust server the message `{h(c)}_PriKA`, where `PriKA` is Alice's private key. Because this message is hashed, it has a small size, and `c` remains secret from the server. The server concatenates this message with a time stamp `T`, encrypts it with its private key and sends it back to Alice. This message is Alice registry. Alice can now prove to a judge that code `c` is hers: she can provide the registry to the judge who can obtain `{h(c)}_PriKA,T` using server's public key. The judge can verify the time stamp if it is needed, and can compute `h(c)` from `c` (provided by Alice), and compare it with `{{h(c)}_PriKA}_KA`, which has to be equal to the computed `h(c)`.

In security protocols, hashing is not only a compression feature but also a guarantee of security: an implementation that does not hash information may produce security flaws. Therefore, the mutation

DOI: 10.1002/stvr

```
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)
    2.  State  = 2 /\
        RCV({Na.Nb'.Bob}_Ka) =|>
        State'= 4 /\ SND({Nb'}_Kb)
end role
```

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

Figure 12. NSPK-Lowe: Alice role with permutation.

that is proposed here consists in omitting hash functions. From a practical point of view, it is easy to perform this mutation: it suffices to suppress hash function in the HLPSL specification of the protocol.

4.6. Permutation mutation

In some context, the order of information blocks in a message is important. For instance, it has been shown by Millen and Shmatikov [26] that, in the modified version of NSPK, sending `{NA,Alice}_KB` is not, from a security view point, equivalent to send `{Alice,NA}_KB`. Such a man-in-the-middle flaw is quite rare. However, these kinds of permutations can be a very common non-conformance between the protocol specification and its implementation.

If a message is the concatenation of two blocks, this mutation acts as a commutation. A syntactic analysis has to be performed to find the corresponding `RCV` in others roles. If the message is the concatenation of n blocks, the number of possible permutations is $n!$, which can produce a huge number of mutants. For the experimentations, it has been decided to place the last block of a message to the first place. Because there are many possible messages, this still provides many mutants.

To perform this mutation, a simple permutation of the order of the messages blocks is made. Consider for instance Alice's role specification reminded in Figure 12, left column. The permutation mutation may consist in permuting the order of Alice's identity and the nonce in the first message. The obtained mutated role is described in Figure 12, right column.

Note that Bob's role also has to be modified: the `RCV({A.Na'}_Kb)` corresponding to the reception of the first message has to match the new one. It is thus modified into `RCV({Na'.A}_Kb)`.

5. EXPERIMENTS

This section presents the experimental results that were obtained. First, the mutant generation tool is presented. Then, the application of the mutations on a set of existing security protocols is presented. Finally, the use of HOM is experienced and reported.

5.1. Tool support

For the need of the experiments, a prototype tool, named *jMuHLPSL*[‖], has been designed. It is implemented in Java and displays a simple graphical interface, which makes it possible to select an HLPSL file (or a directory containing HLPSL files) to be given as input and to generate all possible mutants from it, by applying on demand the mutations described in Section 4, in order to generate new HLPSL protocols. A screenshot of the tool is given in Figure 13.

[‖]Freely available at http://disc.univ-fcomte.fr/home/~fdadeau/tools/jMuHLPSL.jar.

DOI: 10.1002/stvr

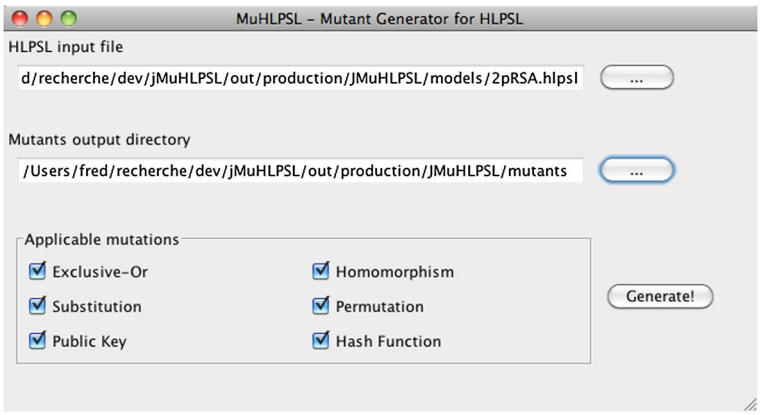

Figure 13. Screenshot of the jMuHLPSL tool.

Mutation Permut.	Mutation Pub. Key	Mutation Hach	Mutation Substi.	Mutation XOR	Mutation Homom.
680	81	92	680	46	46

Figure 14. Number of mutants for each mutation operator.

5.2. *Application on a bench of protocols*

The selected bench of protocols originates from the AVISPA project. During this project, 50 HLPSL models, representing realistic protocols, were designed and analysed by the different back-ends. The mutation operators presented in this article were applied on these 50 protocols, producing a total of 1625 mutants with the repartition given in Figure 14.

All these mutants were analysed using three of the AVISPA back-ends, namely, CL-AtSe (both with and without the `notype` option), OFMC (both with and without the `untyped` option) and SATMC, with the following restrictions:

- The XOR-mutated protocols were not analysed by SATMC because this tool does not support this operator.
- A 1 min time-out was set for each analysis.

Results are provided in Figures 15 and 16: for each protocol and each mutation operator, the three numbers, respectively, represent the number of SAFE verdicts, the number of UNSAFE verdicts (that will provide test cases using the attack trace returned by the tool) and the number of INCONCLUSIVE verdicts (the mutation makes the protocol inexecutable). Empty cells are displayed when the mutation is not applicable for the considered protocol.

For instance, for the 2pRSA protocol, 14 mutants were generated (four for the permutation and substitution operators, one for the XOR and homomorphisms operators and two for the hash and public key operators). For each permutation mutant, each back-end returns that the protocol is safe except SATMC, which did not answer in time. Therefore, a total of 16 SAFE verdicts are reported.

When a mutant protocol is declared UNSAFE, each of the considered back-ends returns an attack trace in terms of exchanged messages between the intruder and the agents participating to the protocol. In general, the described attack depends on the back-end, providing different possible test cases for the same unsafe mutant. An example of an attack trace returned by the CL-AtSe back-end is given in Figure 17. The concretization of this abstract test case then consists in implementing an intruder that attempts to (re)play the attack with the participants. This part is beyond the scope of the work presented here, although there exists some works presenting semi-automated translation of attack traces into executable test cases [27].

Notice that, during this phase, all the considered back-ends came to the same verdicts regarding the analysis of the protocols (except in case of time-outs).

DOI: 10.1002/stvr

Protocol	Nb. Mut.	Mutation Permut.	Mutation Pub. Key	Mutation Hach	Mutation Substi.	Mutation XOR	Mutation Homom.
2pRSA	14	16 - 0 - 0	8 - 0 - 0	9 - 0 - 0	8 - 0 - 10	0 - 4 - 0	4 - 0 - 0
8021x-Radius	109	212 - 0 - 0		0 - 4 - 0	200 - 0 - 15	0 - 0 - 4	4 - 0 - 0
APOP	5	10 - 0 - 0		0 - 4 - 0	0 - 4 - 5		
ASW	111	212 - 0 - 0	12 - 0 - 0		200 - 0 - 15	0 - 4 - 0	4 - 0 - 0
CHARPv2	3	4 - 0 - 0		0 - 4 - 0	0 - 4 - 0		
CRAM-MD5	5	4 - 0 - 0		0 - 4 - 10	4 - 0 - 0		
CTP	19	20 - 8 - 0		6 - 6 - 0	17 - 3 - 10	0 - 4 - 1	0 - 3 - 0
DHCP delayed-auth	7	9 - 0 - 0		4 - 8 - 0	0 - 4 - 5		
EAP-AKA	20	20 - 0 - 0		24 - 8 - 0	12 - 0 - 10	4 - 0 - 1	4 - 0 - 0
EAP-Archie	30	43 - 13 - 0		3 - 5 - 0	37 - 12 - 5	2 - 1 - 0	3 - 1 - 0
EAP-IKEv2	18	24 - 0 - 0	8 - 0 - 0	8 - 0 - 0	16 - 0 - 10	2 - 0 - 0	4 - 0 - 0
EAP-SIM	32	52 - 0 - 0		16 - 8 - 0	36 - 0 - 20		
EAP-TLS	73	128 - 0 - 0	12 - 0 - 0	12 - 4 - 0	116 - 4 - 10	0 - 4 - 0	0 - 4 - 0
EKE2	7	8 - 0 - 0		0 - 4 - 0	0 - 4 - 5	4 - 0 - 0	4 - 0 - 0
Geopriv	68	0 - 24 - 0	1 - 2 - 0	1 - 2 - 0	0 - 10 - 35	0 - 4 - 0	0 - 3 - 0
Geopriv password	23	4 - 5 - 0	3 - 9 - 0	0 - 2 - 0	5 - 1 - 15	0 - 0 - 4	0 - 3 - 0
Geopriv pervasive	17	9 - 0 - 0	10 - 4 - 0		0 - 8 - 10	0 - 0 - 4	0 - 0 - 5
HIP	24	32 - 0 - 0	8 - 4 - 0	8 - 4 - 0	32 - 0 - 0	0 - 4 - 0	4 - 0 - 0
IKEv2-CHILD	16	24 - 0 - 0		8 - 0 - 0	16 - 0 - 10	4 - 0 - 1	2 - 0 - 0
IKEv2-DSx	22	8 - 8 - 0	4 - 2 - 0	2 - 2 - 0	6 - 5 - 15		0 - 3 - 0
IKEv2-EAP Archie	43	30 - 0 - 0	11 - 0 - 0	13 - 0 - 0	23 - 0 - 15		
IKEv2-MACx	20	13 - 8 - 0		1 - 5 - 0	7 - 4 - 15	2 - 1 - 0	1 - 1 - 0
ISO2	9	8 - 2 - 0	8 - 7 - 0		0 - 4 - 5	0 - 4 - 0	0 - 4 - 0
ISO4	16	15 - 5 - 0	11 - 7 - 0		12 - 8 - 0	0 - 4 - 0	0 - 4 - 0
Kerberos Cross-Realm	70	102 - 34 - 0			66 - 22 - 60	0 - 0 - 4	0 - 4 - 0
Kerberos Forwardable	80	78 - 39 - 0			32 - 30 - 45	0 - 0 - 4	0 - 4 - 0
Kerberos PKINIT	40	51 - 17 - 0	0 - 15 - 0	3 - 1 - 0	36 - 16 - 20	0 - 0 - 4	0 - 4 - 0
Kerberos preauth	36	51 - 17 - 0	0 - 0 - 0		33 - 11 - 30	0 - 0 - 4	0 - 4 - 0
Kerberos basic	36	51 - 17 - 0	0 - 0 - 0		33 - 11 - 30	0 - 0 - 4	0 - 4 - 0
Kerberos Ticket-Cache	36	51 - 17 - 0			33 - 11 - 30	0 - 0 - 4	0 - 4 - 0

Figure 15. Results—Part 1.

Discussion

This evaluation aimed at two purposes. First, it was meant to check the ability of generating automatically test cases for security protocols. Second, it was meant to check if the mutation operators that are proposed in this article were relevant, in terms of resulting potential attack traces (test cases), which can be produced from their application.

From a practical point of view, this test generation technique made it possible to generate 788 UNSAFE verdicts (leading to test cases) out of 50 protocols. The analysis of the results shows that it was not possible to create at least one test case for each protocol, even if numerous mutants could be made from these protocols: no UNSAFE verdicts were provided for EAP-IKEv2, IKEv2-CHILD and IKEv2-EAP-Archie. Conversely, 73 UNSAFE verdicts were produced for Kerberos-Forwadable, and for the Geopriv protocol, at least one unsafe verdict was obtained for each mutation operator.

There is a total of 1625 mutated protocols. For each one, except the 46 XOR mutated protocols, there are five possible analysis (four for the XOR). Therefore, there is a total of 8079 potential analysis. Sometimes, the OFMC back-end, running with the option `untyped` returns *There occurred an error in substitutions. This bug may be caused by a normalisation of an untyped variable. Please try with option1 -nonorm or adding type information/not using option -untyped.* Moreover, the experimental results were computed with a 1 min time limit. With these limits, 3798 analysis were

DOI: 10.1002/stvr

Protocol	Nb. Mut.	Mutation Permut.	Mutation Pub. Key	Mutation Hach	Mutation Substi.	Mutation XOR	Mutation Homom.
KLIPKEY-SPKM unknown-initiator	13	8 - 0 - 0	12 - 0 - 0	4 - 0 - 15	4 - 4 - 0	0 - 2 - 0	0 - 2 - 0
LPD-IMSR	8	10 - 0 - 0	10 - 0 - 0		0 - 8 - 0	0 - 4 - 0	0 - 3 - 0
NSPK-fix	8	10 - 0 - 0	10 - 0 - 0		0 - 8 - 0	0 - 4 - 0	0 - 3 - 0
NSPK-KS-fix	13	0 - 0 - 20	3 - 0 - 0		1 - 0 - 10	0 - 4 - 0	0 - 3 - 0
PBK-fix weak-auth	10	4 - 5 - 0	0 - 15 - 0	4 - 0 - 0	8 - 0 - 0	2 - 2 - 0	0 - 4 - 0
QoS-NSLP	13	10 - 2 - 0	8 - 5 - 0		8 - 5 - 0	0 - 0 - 4	0 - 2 - 0
RADIUS-RFC2865	105	204 - 0 - 0		0 - 4 - 0	200 - 0 - 5	0 - 4 - 1	4 - 0 - 0
SET-purchase honest-payment gateway	54	66 - 3 - 0	24 - 8 - 0		56 - 0 - 20	0 - 4 - 0	3 - 0 - 0
SIMPLE	6	1 - 0 - 0		2 - 0 - 0	1 - 0 - 0		0 - 2 - 0
SIP	232	460 - 0 - 0		4 - 4 - 0	429 - 0 - 35		
S-KEY	1			0 - 2 - 0			
SPEKE	4	4 - 0 - 0			4 - 0 - 0	3 - 1 - 0	0 - 4 - 0
SPKM-LIPKEY	13	8 - 0 - 0	12 - 0 - 0	4 - 0 - 15	0 - 8 - 0	0 - 4 - 0	0 - 4 - 0
SRP	7	8 - 0 - 0		0 - 4 - 0	0 - 0 - 10	4 - 0 - 0	1 - 0 - 0
SSH-transport	6	2 - 0 - 0		3 - 4 - 0	0 - 0 - 5	1 - 3 - 0	2 - 0 - 0
TESLA	8	8 - 0 - 0	4 - 0 - 0	0 - 4 - 0	4 - 0 - 5	4 - 0 - 1	4 - 0 - 0
TLS	92	148 - 0 - 0	24 - 0 - 0	24 - 8 - 0	124 - 4 - 25	3 - 4 - 0	4 - 4 - 0
TSIG	7	8 - 0 - 0		4 - 0 - 0	0 - 0 - 10	3 - 0 - 0	0 - 4 - 0
TSP	9	7 - 1 - 0	4 - 0 - 0	8 - 0 - 0	4 - 0 - 5	4 - 0 - 0	0 - 4 - 0
UMTS-AKA	7	4 - 0 - 0		4 - 8 - 0	4 - 0 - 0	4 - 0 - 0	1 - 0 - 0

Figure 16. Results—Part 2.

completed as described in Figure 18 (the last row points out the percentage of performed analysis. Note that the time limit makes SATMC answering rarely. SATMC was run using the option `solver=sim`, that is, using the SIM solver. Other options allowing to use the zChaff or the mChaff solvers do not work on the version of AVISPA used for the experiments. It was also possible to use the SATO solver, but this experimentation has not been done. After 20 min on a permuted version of 2pRSA, SATMC did not finish to analyse the protocol and used about 2 GB of memory.

Considering the whole set of mutation operators, their efficiency in producing unsafe mutants is relatively weak (10% of the all possible analysis, 21% of the completed analysis). Even if this percentage may seem relatively low, it is important to notice that the considered protocols were mainly real-world protocols that are robust to various kinds of attacks or generally prevent possible implementation errors.

The homomorphism mutation and the exclusive-or mutation are applicable on the same set of protocols with a similar success in introducing security flaws (40% when applicable). Hash functions mutation also produce efficiently unsafe mutant protocols, producing 25% of security flaws when applicable. On the opposite, the permutation and substitution mutations produced numerous mutants (both 680) but resulted in a very weak percentage of induced faults (about 6%). These last two mutations are indeed very combinatorial. Notice that the substitution mutation tends to produce a large number of incoherent mutants, which can be explained by the fact that in a large majority of cases, the agent that receives a message does actually reuse/check all the elements of the message. Because these elements are supposed to have a specific meaning, this mutation is very unlikely to produce a coherent mutant when applied.

5.3. On the use of higher-order mutation

Higher-order mutation [28] consists in performing not only one but two mutations at the same time. In order to evaluate the relevance of HOM in this approach, an experiment has been designed that consisted in reusing the protocols declared as ‘safe’ after the application of a first mutation operator and apply on them a second mutation operator that also leaves the protocol safe when applied independently.

Figure 19 displays the different combination of mutations that were considered and the number of resulting mutants for each protocol. An empty cell means that the combination of mutations has not

DOI: 10.1002/stvr

```
SUMMARY
  UNSAFE

DETAILS
  ATTACK_FOUND
  TYPED_MODEL

PROTOCOL
  PBK-fix-weak-auth_permut_2.if

GOAL
  Authentication attack on (a,a,msg,tag2)

BACKEND
  CL-AtSe

STATISTICS

  Analysed   : 246 states
  Reachable  : 106 states
  Translation: 0.00 seconds
  Computation: 0.00 seconds


ATTACK TRACE
 i -> (a,12):  start
 (a,12) -> i:  i.{tag1.n21(Msg)}_(inv(pk_a)).{pk_a}_f
               & Witness(a,a,msg,n21(Msg));

 i -> (a,3):  start
 (a,3) -> i:  b.{tag1.n1(Msg)}_(inv(pk_a)).{pk_a}_f
              & Witness(a,a,msg,n1(Msg));

 i -> (a,12):  tag1
 (a,12) -> i:  {tag1.tag2}_(inv(pk_a))

 i -> (b,4):  b.{tag1.tag2}_(inv(pk_a)).{pk_a}_f
 (b,4) -> i:  n5(Nonce)

 i -> (a,3):  n5(Nonce)
 (a,3) -> i:  {n5(Nonce).tag2}_(inv(pk_a))

 i -> (b,4):  {n5(Nonce).tag2}_(inv(pk_a))
 (b,4) -> i:  ()
              & WRequest(a,a,msg,tag2);
```

Figure 17. CL-AtSe analysis log.

	CL-ATSE (typed)	CL-ATSE (untyped)	OFMC (typed)	OFMC (untyped)	SATMC
Nb.ofAnalysis	1460	1440	1341	1361	171
Efficiency	90%	88%	83%	84%	11%

Figure 18. Number of analysis for each back-end.

been applied for the considered protocol. Only a subset of the original set of protocols is displayed, representing the protocols on which a couple of mutations leaving them safe can be performed (e.g. the Authenticated Post Office Protocol for which there was only one applicable mutation that could not be used for this experiment). Notice that there is no specific order in the application of the two mutations.

The mutant protocols were analysed using the CL-AtSe back-end. None of them was declared unsafe by the tool.

DOI: 10.1002/stvr

Pe = Permutation, PK = Public Key, Ho = Homomorphism, Su = Substitution, Ha = Hash Function, Xo = XOR

Protocol	Pe PK	Pe Ha	Pe Ho	Pe Su	PK Ha	PK Ho	Ha Ho	Su Ho	Su Ha	Xo Pe	Xo Ho	Xo Ha	Xo PK	Xo Su
2pRSA	8	8	4		4	2	2							
8021x			53											
ASW	159		71			3								
CRAM-MD5				1										
EAP-AKA			5											
EAP-IKE	12	12	13		4	2	2							
EAP-TLS	96													
EKE2			2							2	1			
HIP			8											
IKEv2-CHILD		12	8				2							
IKEv2-EAP Archie	60	105	25		28	4	7							
KLIPKEY-SPKM unknown-initiator	6													
LPD-IMSR	4													
NSPK-fix	4													
PBK-fix									2					
RADIUS-RFC2865			51											
SPEKE				1										
SPKM-LIPKEY	6													
SRP			2							2				
SSH-transport			1											
TESLA	2		4			1								
TLS	111													
TSIG		2								2		1		
TSP					2							1	1	
UMTS-AKA			1	1				1		1	1			1
Total	372	139	248	3	38	12	13	1	2	7	2	3	1	1

Figure 19. Results of higher-order mutant generation.

6. RELEVANCE OF THE MUTATION OPERATORS

It is important to notice that the mutation operators that are proposed in this article are motived by actual implementation choices (notably, for the use of the xor-encryption scheme and the use of an encryption operator that displays the homomorphism property) or faults that can be made when implementing the protocol. This part discusses the implementation errors that can be related to the remaining four mutation operators.

6.1. Public key mutation

In a security context, it is often required to not use the same public keys on different systems or devices. For instance, the commercial tool Unfuddle (Subventurate LLC, Honolulu, Hawai) (based on Git) dedicated to document sharing does not allows the use of identical public keys. The description of the *X.509 Internet Public Key Infrastructure—Online Certificate Status Protocol* by the Internet Engineering Task Force, under RFC 2560 requires explicitly that *Two CAs will never, however, have the same public key...* (Section 4.1.2). However, several unaware users sometimes use the same public key (for instance, by copying their .ssh files) for different accounts in order to easily manage the communications in a secured network. Moreover, a recent work [29] shows that many public keys are identical because of bad pseudo-random generation. One can also cite the bug in an OpenSSL package leading to a lot of identical public keys[**]. It seems therefore important to check whether a protocol is sensitive to the use of same public keys by both agents and whether it is detected during an execution.

[**]http://www.debian.org/security/2008/dsa-1571.

DOI: 10.1002/stvr

6.2. Hash mutation

The hash mutation operator can be used to detect security flaws if the hashing function was omitted. Of course, this is a straightforward error, but the goal of this mutation is also to detect whether the protocol is sensitive when non-cryptographic hash functions were used. If the mutated protocol is analysed as unsafe, it means that it must be checked whether the involved hash function is cryptographically secured (for instance, it shall not be a simple modulus). It means that if, from a message `h(m)`, the intruder can forge `m`, then there is a security leak in the protocol. In this case, the attack trace is irrelevant. Nevertheless, the important point is rather that the protocol is unsafe.

In the theory of secure hashing functions, the main security properties used are *Preimage resistant*, 2^{nd} *Preimage resistant* and *Collision resistant*. The proposed mutation tests something such as *if the hash function is not preimage resistant, then there is a security leak*. It would be possible to define more complex mutations to detect the sensitivity to other properties.

6.3. Substitution and permutation mutations

These last two mutation operators focus on message blocks. To illustrate the correlation of these mutation operators with real programming mistakes, consider the pieces of Java code given in Figures 20 and 21. These codes are extracted from a free implementation of the NSPK that can be easily found on the Internet. Figure 20 shows how Bob forges the message that is sent in response to Alice's first contact. The second excerpt in Figure 21 shows the behaviour of Alice, who just

```
89  ...
90  currentDataList = new ArrayList<Data>();
91  currentDataList.add(BSent);
92  currentDataList.add(NAReply);
93  currentDataList.add(NBSent);
94  Data NANBBKSent = new Data(currentDataList, encrypter);
95  ...
```

Figure 20. Excerpt of Bob implementation.

```
76  ...
77  Data BNANBKReply = Data.read(serverIn);
78  System.out.println("BNANBK Reply:");
79  System.out.println(BNANBKReply);
80
81  currentDataList = BNANBKReply.unnest(decrypter);
82
83  if(currentDataList.size() == 3)
84  {
85      Data BReply = currentDataList.get(0);
86      Data NAReply = currentDataList.get(1);
87      Data NBReply = currentDataList.get(2);
88
89      System.out.println("B Identity Reply:");
90      System.out.println(BReply);
91
92      System.out.println("NA Reply:");
93      System.out.println(NAReply);
94
95      System.out.println("NB Reply:");
96      System.out.println(NBReply);
97
98      //check that the nonce you sent bob is the one he sent back
99      //also check that its actually bob
100     if((NASent.equals(NAReply))&&(new String(BReply.getData()).equals("Bob")))
101     {
102         currentDataList = new ArrayList<Data>();
103         currentDataList.add(NBReply);
104         Data NBKSent = new Data(currentDataList, encrypter);
105         ...
```

Figure 21. Excerpt of Alice implementation.

DOI: 10.1002/stvr

received Bob's message and has to check that this latter contains the nonce she sent before and Bob's identity.

These pieces of code are relevant to illustrate how these two mutation operators can be related to classical implementation errors.

First, consider Bob's code in Figure 20. It is easy to imagine that the programmer may choose (willingly or not) to change the order in which the different blocks of the message appear w.r.t. original protocol specification. In this case, he will most certainly refer to his own code to write the extraction of the message blocks in the corresponding code for Alice (see lines 85–87 of Figure 21), thus introducing a permutation similar to the one presented in this paper.

Second, consider line 100 of Alice's implementation in Figure 20, in which the programme checks that the blocks composing Bob's message correspond to Alice's nonce and Bob's identity. It is easy to imagine that the second predicate can be omitted, illustrating the substitution operator.

7. CONCRETIZATION OF THE ABSTRACT TEST CASES

Bridging the gap between an abstract attack scenario derived from a specification and a penetration test on real implementations of a protocol is still a non-trivial issue. This section proposes an architecture for automatically generating abstract attacks and converting them to concrete tests on protocol implementations. In particular, the objective is to improve previously proposed black-box testing methods in order to discover automatically new attacks and vulnerabilities. A related approach to test concretization has been published independently [27]. The authors proposed an ad hoc message construction procedure to compile their attack scenarios. The present approach directly reuses a previously existing procedure that was designed for the AVANTSSAR platform [30]. It is also important to notice that this first method [27] does not exploit mutation techniques and has been applied to a different protocol.

7.1. Platform architecture

As discussed before, the attack trace produced by a model checker is rather abstract. In order to be able to detect real attacks that affect protocol implementations, it is mandatory to provide a platform that performs both messages format conversion, from a formal level to the implementation level, and real communications with the SUT. This platform's architecture is now described, in terms of components, with their functionalities and interactions. It displays three main components, each with a specific role:

1. *Attack trace compiler*: identifies agents, message types and elementary operations;
2. *Scenario execution engine*: generates (respectively, retrieves) outgoing (respectively, incoming) messages;
3. *Attack simulator*: simulates the scenario on real communication channels.

As shown in Figure 22, the testing environment takes as inputs the attack trace and the mutated model of the considered protocol and returns an indication whether the considered attack on the considered implementation exists or not.

7.2. Attack trace compiler

The *Attack trace compiler* transforms an abstract attack trace into an executable attack scenario. The component collects intruder initial knowledge from the HLPSL protocol and follows the attack trace instructions to build the attack scenario. The latter describes in detail the actions that should be performed by the intruder when executing the attack. Hence, it is structured into steps and elementary operations, each step corresponding to an abstract attack trace instruction. A data structure is initialized with intruder *Initial knowledge* (keys or agent's identities) and is updated each time the system evolves to a new state.

DOI: 10.1002/stvr

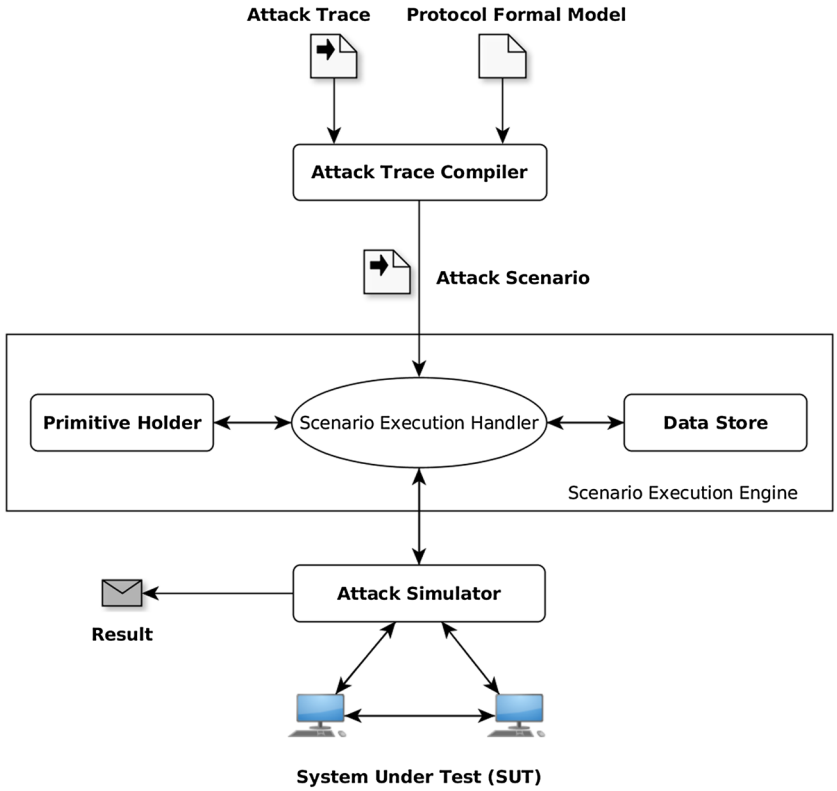

Figure 22. Platform architecture.

Scenario generation algorithm

In order to generate an executable attack scenario, for each step $I \rightarrow B : M$ in the attack trace, where I is an agent controlled by the intruder, one needs to check whether the intruder can compose M from the pieces of knowledge at this step. Therefore, a procedure must be used to verify whether a message t can be composed from a set of messages E by the intruder simulator. The proposed method provides a sequence of operations that allows one to construct a given message from a given set of messages. When several sequences are possible, the choice does not matter for the remaining of the process. The method is complete in the sense that if the message can be constructed, (at least) one sequence of operations is derived.

Algorithm 1 is a variant of a well-known algorithm in protocol analysis, and it is implemented in AVANTSSAR platform as part of the orchestration module. An optimised version has been designed for this purpose. The algorithm provides the sequence of operations to apply in order to generate t from E, whenever it is possible. This sequence of operations is represented as a first-order term $\bar{t}$. The proposed method constructs t from knowledge set E by checking which sub-term in $E \cup t$ can be constructed. Several passes are necessary, because after constructing some keys, one can decrypt more messages. Hence, iteratively, more and more messages get constructible. At the end of the main loop, one checks whether the message t is in the set of constructible messages K_1. The notation $\{l\}_r$ indicates that the plain text l is encrypted by key r. The inverse key of r is denoted by r^{-1}. P_1 (respectively, P_2) are symbols that denote the left (respectively, right) projection of a pair.

7.3. Scenario execution engine

This module is responsible of translating the attack scenario, from the formal level to the implementation level. Because the execution environment is designed for an implementation level, the exchanged messages are real network messages. As seen in the previous sections, messages in the

DOI: 10.1002/stvr

INPUT E: set of messages
OUTPUT t: message
Let $Sub(E,t)$ be the set of submessages of $E \cup \{t\}$
$K_1 \leftarrow E$
$K_0 \leftarrow Sub(E,t) \backslash E$
repeat
  for all $e \in Sub(E,t)$ **do**
    if $e \in K_0$, and $l, r \in K_1$, with $e = \{l\}_r$ **then**
      $K_0 \leftarrow K_0 \backslash \{e\}$; $K_1 \leftarrow K_1 \cup \{e\}$; $\overline{e} \leftarrow crypt(\overline{l}, \overline{r})$
    end if
    if $e \in K_0$, and $l, r \in K_1$, with $e = pair(l,r)$ **then**
      $K_0 \leftarrow K_0 \backslash \{e\}$; $K_1 \leftarrow K_1 \cup \{e\}$; $\overline{e} \leftarrow pair(\overline{l}, \overline{r})$
    end if
    if $e \in K_1$, $l \in K_0$, $r^{-1} \in K_1$, with $e = \{l\}_r$ **then**
      $K_0 \leftarrow K_0 \backslash \{l\}$ $K_1 \leftarrow K_1 \cup \{l\}$; $\overline{l} \leftarrow decrypt(\overline{e}, inv(\overline{r}))$
    end if
    if $e \in K_1$, with $e = pair(l,r)$ **then**
      $K_1 \leftarrow K_1 \cup \{l,r\}$; $\overline{l} \leftarrow P_1(\overline{e})$ and $\overline{r} \leftarrow P_2(\overline{e})$
    end if
  end for
until K_1 unchanged or $t \in K_1$
if $t \in K_1$ **then**
  $\overline{t}$
else
  $False$
end if

Algorithm 1: Message construction

formal model are specified as first-order terms. Therefore, it is necessary to map these terms to concrete messages and operations. This is the main role of the *Scenario execution engine* (highlighted in Figure 22), which ensures the association between abstract messages and concrete ones, stored in the *Data store* module. Operation execution is held with the functionality provided by the *Primitive holder*.

The attack scenario instructions can be classified into three categories: (i) message construction; (ii) message sending and (iii) message receiving. To do this, cryptographic primitives (*crypt, pair* and *unpair*) and network primitives (*send* and *receive*) are used.

Primitive holder. The needed cryptographic operations are defined in the *primitive holder* module. In relation with the specification of the protocol, this component provides a library of operations such as encryption, decryption, nonce generation, signing and concatenation. It is necessary to make sure that the whole scenario can be executed without any errors. To do that, the primitive holder provides all the possible operations needed by the protocol implementation.

Data store. Message creation depends on the knowledge acquired in previous step of the scenario, because the tested protocols are stateful. Hence, all the messages handled by the platform are saved in the *data store* in their real format and in an indexed way, which facilitates data processing. The Data store also contains all objects required for intermediate computation like encryption keys, data nonces, agent identities and sub-messages.

Scenario execution handler. This is the platform core algorithm that handles the instantiation of abstract operations by concrete executable ones. It takes as input the elementary steps of an attack scenario and processes each instruction in order to identify the next operation to perform and its arguments. It interacts with the primitive holder module to execute cryptographic operations and

DOI: 10.1002/stvr

with the data store module to save or retrieve arguments depending on the attacker behaviour described in the attack scenario. Algorithm 2 describes all the interactions between the different modules.

Input: Instruction I
Output: Request to another component
Case $\{I$ is send$(X_i)\}$ **then**
    Get data from the Data Store at position i ;
    Call A−Simulator to send message
Case $\{I$ is X_i = receive()$\}$ **then**
    Call A-Simulator to get the received message ;
    Store the message on the Data Store at position i
Case$\{I$ is X_i = operation$(X_y,X_z)\}$ **then**
    Get data from Data Store at positions y and z;
    Call the Primitive Holder to execute the primitive;
    Store the message on the Data Store at position i
Case$\{I$ is finish()$\}$ **then**
    Exit with success

Algorithm 2: Scenario execution handler

The first (respectively second) case corresponds to a message sending (respectively receiving) operation over the network. The third case of Algorithm 2 corresponds to the message construction or decomposition. In both cases, the handler invokes the data store and the primitive holder modules. Consider, for instance, instruction $X_1 = pair(X_2, X_3)$. First, the scenario execution handler collects the arguments by requesting them from the data store. Then, it calls the **concat** method in the primitive holder to construct the message. Finally, the latter is stored at the result position 1 in the data store.

7.4. Attack simulator

After mapping a formal message to the real format, the *scenario execution handler* processes emission and reception operations. In these cases, it sends a request to the *attack simulator* module, which is the interface of the platform with the external environment. At the formal level, the protocol model abstracts some fields existing in a real implementation, which needs to be restored at the concrete level. The *attack simulator* is responsible for the conformance of the exchanged data with the protocol model, meaning that it has to identify the relevant fields and retrieve data from the SUT response (case of receiving operation) and to instantiate the relevant fields in the request message (case of sending operation). In general, the *attack simulator* tasks include (i) creating the real communication channels; (ii) sending messages and (iii) receiving messages.

This module is also responsible for validating the execution of the whole attack scenario. By monitoring the SUT states, it can easily detect the state that corresponds to the success of an attack.

This platform has been used for testing a real-world protocol, namely, the PayPal standard protocol. This case study is now reported.

8. CASE STUDY: THE PAYPAL STANDARD PAYMENT PROTOCOL

In order to assess the effectiveness of the proposed approach, the whole process was applied on a payment protocol used in e-commerce Web applications. More specifically, the test process has targeted the *PayPal standard* protocol implementation, in a recent version of the *Magento* platform [31].

DOI: 10.1002/stvr

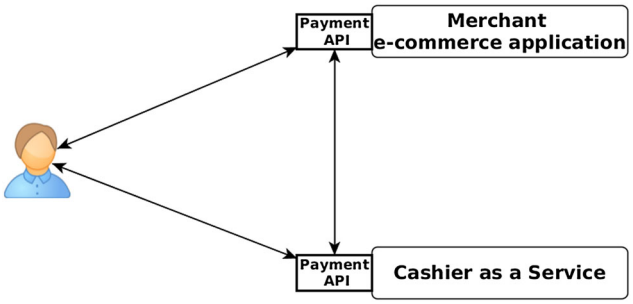

Figure 23. Entity interactions during a checkout.

Using this approach, it was possible to exhibit a serious logical flaw that was recently discovered and can be exploited by a malicious user in order to purchase a product with a chosen-by-him amount. This section describes the whole process that leads to this vulnerability detection, starting from a presentation of the protocol case study, followed by the test case concretization and completed by the experimental details.

8.1. Payment protocol and e-commerce web application

Nowadays, e-commerce Web applications increasingly integrates a trusted third-party component presented as a *cashier-as-a-service* (CaaS) in the payment process. The main purpose is to better guarantee secured payment transactions, as the CaaS can collect the payment of a purchase from the shopper and inform the merchant of the completion of the payment without revealing the shopper's sensitive data such as a credit card number. In the considered case study, the well-known Paypal server is used as an example of CaaS server. Unfortunately, this third-party integration introduces a complexity in the payment protocol implementation within the e-commerce application that brings new security issues [32]. Indeed, an improper distribution of the protocol functionality between the involved entities leads to logic flaws that can be exploited by a malicious shopper. Actually, an online purchase transaction is always initiated by the client (Web browser) and managed by some public API methods implemented in the two sides: the merchant and the CaaS (Figure 23). A dishonest shopper can make Web-API calls of methods existing on the e-commerce application with well-chosen arguments and in an arbitrary order so that he can shop products for free or alter the way the payment is verified. This shows that having communications over HTTPS does not prevent severe attacks against e-commerce applications.

8.2. High-level protocol specification language formal modelling

This work concentrates on logic flaws in payment protocols used in e-commerce and especially in inconsistencies between the states of the CaaS and the merchant that lead to logic flaws. In order to conduct the security analysis of the PayPal payment protocol, the approach starts by specifying the protocol relying on Alice–Bob notation. The checkout process begins when the 'Pay Now' button on the merchant website is clicked. This operation directs the shopper's browser to the PayPal website where he is invited to give his PayPal buyer account credentials to continue the purchase process. If the information entered by the user is correct, the shopper is again redirected to a payment success merchant Web page. Behind the scene, there are HTTP interactions between the three parties who communicate by calling Web-APIs exposed by the merchant and the CaaS. Such APIs are essentially dynamic Web pages and are invoked through HTTP requests. A client sends an HTTP request through a URL with a list of arguments and receives an HTTP response, often a Web page dynamically constructed by the server, as the outcome of the call. The formal model of the protocol was designed using the PayPal documentation [33] and some traffic analysis. To examine the traffic, Magento was deployed on a local xampp server [34] and HTTP traffic capturing tools were used, such as Fiddler [35], retrieving all the HTTP exchanged messages between the involved

DOI: 10.1002/stvr

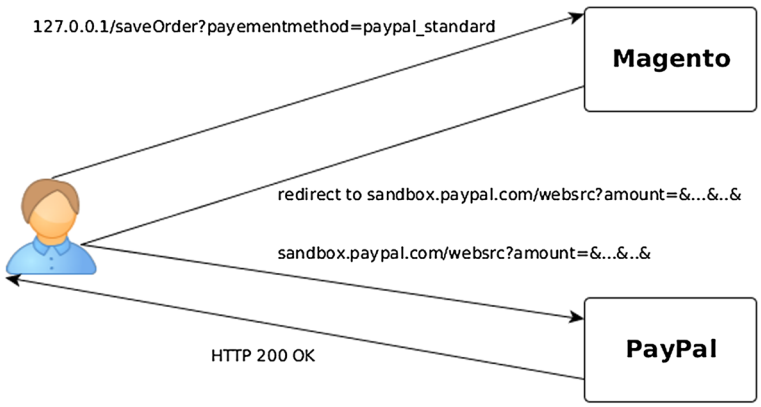

Figure 24. PayPal standard workflow.

```
C -> M : C, OrderDescr
M -> C : M, redirectLink, Sign(PurchAmt,M), OrderDescr
C -> P : C, PWD, Session, Sign(PurchAmt,M), OrderDescr, redirectLink
P -> C : Success
```

Figure 25. Alice-Bob notation of PayPal standard protocol.

entities during a checkout process. Figure 24 depicts all the exchanges of HTTP messages. Figure 25 provide the Alice–Bob notation of the PayPal standard protocol, which is described in an associated HLPSL specification (not shown here). For the purpose of efficiency and conciseness, only the most significant steps and fields were modelled. In this figure, C denotes the client (Web browser), M is the merchant (Magento) and P is the CaaS (Paypal). Also, the `Sign`(m, id) function represents a message m signed by the agent having identity id. First, the client starts with sending his identity and the order description information (`OrderDescr`): product details, shipping address, billing address and so on. The merchant redirects the client to the `redirectLink` which corresponds to the authentication page on Paypal server. The merchant response also contains the product purchase amount (`PurchAmt`) signed by his private key. Then, the client sends his account credentials (`C` and `PWD`) to the Paypal server with the received purchase amount (still signed). Note that a nonce `Session` is used to identify the session initiated by the client. Finally, the Paypal server validates the received information and returns a `Success` payment page.

8.3. Mutation and abstract test case generation

The use of the mutation operators makes it possible to eliminate the signature of the product amount, which is normally performed by the merchant in order to guarantee the integrity of the data sent to PayPal. Figure 26 shows a mutated excerpt from the HLSPL specification.

Notice that the applied mutation, that is, deleting a signature, is not presented in Section 4. However, it is very close to the *hash function* mutation operator described in Section 4.5.

After applying the mutation process, the model checking tools are used to verify the protocol and possibly generate abstract attack traces if unsafe. Therefore, the purpose of the model checking is specified using goals in HLPSL (Figure 27). The test purpose is defined using HLPSL `witness` and `request` features. These latter inform the Cl-AtSe model checker to ensure that the purchase amount (PurchAmt) value is generated by the merchant, which represents a data authentication goal. Cl-AtSe finds the attack trace given in Figure 28 on this model.

8.4. Experimental details: testing environment

As mentioned before, the study was conducted on the PayPal payment protocol used within the Magento Platform. PayPal standard is the simplest payment method provided by Paypal, which can be integrated by merchant website.

DOI: 10.1002/stvr

```
role client(M,C,P: agent,SND,RCV: channel(dy))
played_by C
def=
local  OrderDescr,RedirectLink, Session,PWD: text,
    PurchAmt,State: nat,

const paypal_merchant_purchamt : protocol_id
init State :=0

transition

[...]

2.State=1 /\ RCV(M.RedirectLink'.PurchAmt'.OrderDescr)
=|> State':=2 /\ Session':=new() /\ PWD':= new()
/\ SND(C.PWD'.Session'.PurchAmt'.OrderDescr.RedirectLink')
/\ SND(OrderDescr) /\ witness(C,P,paypal_client_pwd,PWD')
end role
```

Figure 26. High-level protocol specification language of PayPal standard mutated protocol.

```
role session(M,C,P: agent)
def=
local  SC,SM,SP,
       RC,RM,RP: channel(dy)
composition
client(M,C,P,SC,RC)
/\merchant(M,C,P,SM,RM)
/\paypal(M,C,P,SP,RP)
end role

role environment()
def=
const paypal_client_pwd, purchAmt: protocol_id,
      p,c,m: agent

intruder_knowledge={p,c,m}

composition

session(m,i,p)

end role

goal
authentication_on paypal_merchant_purchamt
end goal

environment()
```

Figure 27. Test environment and goal specifications in high-level protocol specification language.

Actually, the testing environment consists of three agents: (i) the SUT; (ii) the CaaS server and (iii) the dishonest client. The **SUT** is a recent version of the Magento community edition, which was deployed on a local xampp server and which implements a Paypal standard payment method. To avoid any legal problem that might occur while performing tests, the sandbox environment provided by PayPal was used to simulate the **CaaS**. The PayPal sandbox is a self-contained environment wherein one can test PayPal features and use its APIs. PayPal sandbox is an almost identical copy of the live PayPal website. Hence, two test accounts were created: a merchant account used by the Magento platform and a buyer account used by the intruder. The **intruder** behaviour is simulated by the test platform presented in Section 7. The main difficulty for testing real protocol implementations using the model-based approach is the adaptation process, which consist of adding data to the modelled fields, in order to ensure that the constructed message would be accepted by the protocol implementation. Also, while receiving responses from the SUT, the platform, and more specifically the *attack simulator* module, is responsible of retrieving relevant fields from the received response.

DOI: 10.1002/stvr

```
SUMMARY
  UNSAFE

DETAILS
  ATTACK_FOUND
  TYPED_MODEL
  BOUNDED_SEARCH_DEPTH

PROTOCOL
  /home/ghazi/Bureau/dast/avispa-1.1/testsuite/results/paypalStandard.if

GOAL
  Authentication attack on (p,m,paypal_merchant_purchamt,PurchAmt(3))

BACKEND
  CL-AtSe

STATISTICS

  Analysed   : 2 states
  Reachable  : 1 states
  Translation: 0.00 seconds
  Computation: 0.00 seconds


ATTACK TRACE
 i -> (m,4):  i.OrderDescr(1)
 (m,4) -> i:  m.redirectLink.n1(PurchAmt).OrderDescr(1)
              & Witness(m,p,paypal_merchant_purchamt,n1(PurchAmt));
              & Add m to set_72;  Add p to set_72;

 i -> (p,5):  i.PWD(3).Session(3).PurchAmt(3).OrderDescr(3).redirectLink
 (p,5) -> i:  n3(Sucess)
              & Request(p,m,paypal_merchant_purchamt,PurchAmt(3));
```

Figure 28. CL-AtSe attack trace output on the mutated Paypal standard protocol.

Therefore, HTMLUNIT was integrated in the attacker simulator module. HTMLUNIT is a Java unit testing framework for testing Web-based applications. This headless browser allows Java test code to examine returned pages either as text, as XML DOM or as collections of forms, tables and links. It can also deal with HTTPS security, basic HTTP authentication, automatic page redirection and other HTTP headers. Furthermore, this testing framework was used to automate clicks on links and navigation between pages of the online store.

Attack validation is the final and most important step of the testing process. The simulator needs to identify the final state of the SUT when the attack succeeds. In the case study, it is achieved by reaching a successful payment page. Upon completion of the attack scenario execution, the simulator asserts whether the final state of the system is the state corresponding to an attack success or not. This was performed using the JUNIT testing framework, which helps in deploying the attack validation process.

8.5. Results

The platform for implementing test automation aims to perform vulnerability tests on payment protocols used within e-commerce application. Therefore, given an online store at the initial state, a preliminary step is required before starting an execution of the test scenario. More specifically, the attack simulator has to put the SUT in an intermediate state before simulating the intruder behaviour. In the presented case, it consists in creating a client account on the Magento website, selecting a product to purchase and proceeding with a payment using Paypal. Also tests generated from a model are not immediately executable because modelling and programming use different languages. An automated execution of these tests requires to implement specific test adapters. Mainly, modelled objects (e.g. agents, fields and events) must be mapped to implementation-level constructs (e.g. classes and methods in object-oriented programme). This is ensured by the test engineer.

DOI: 10.1002/stvr

The attack that was concretized has been replayed entirely in the environment on which the experimentation was conducted, meaning that this implementation of Magento contains a logical security flaw that consists in the absence of signature of the purchase amount. To exploit this flaw, a malicious attacker can modify the amount to pay and purchase expensive items by paying a ridiculous price. This logical flaw was discovered recently by the NBS Systems company [††].

9. RELATED WORK

A preliminary version of this article has been presented at the International Conference on Software Testing, Verification and Validation (ICST) conference in 2011 [36].

Mutation testing is a large area of research, as shown by Jia and Harman's survey [37]. The work presented in this article focuses on mutation of models to build test cases, as originally introduced by Gopal and Budd [12]. In this context, various approaches consist in mutating pieces of the model so as to introduce behavioural faults that aim at being observed [38–40] using various model analysing tools, such as model checkers [41].

Mutation testing of protocols starts with the work of Probert and Guo [42] proposing a set of mutation operators for security protocols. More recently, mutation-based testing of security protocols has been studied in various approaches. The largest majority of these approaches consider passive testing (i.e. monitoring) so as to observe the violation of security properties at runtime [43–45].

Only a few approaches do consider active testing of security protocols. For example, Lee *et al*. use random walks to ensure the conformance of security protocols specified with FSM [46]. In their paper [47], Shu and Lee use machine learning algorithms to infer EFSM representing the behaviour of implementations of security protocols, before checking security properties on the inferred automaton. The same authors also investigated the use of mutations [48], introducing a simple fault model, based on predicate or guard absence in the protocols.

Wimmel and Jurjens [49] use mutations of system structure diagrams in AutoFocus [50] to build test cases based on attack scenarios. Considered mutation operators are more basic than those introduced here and mainly correspond to omissions in agents identification verification, wrong use of identities and corruption of messages, inspired from a taxonomy of security faults [51]. The fault model presented in this article addresses a wider range of mutations, related to the wider expressiveness of HLPSL.

A closest work, succeeding the work of Dadeau *et al*. [36], has been carried out recently by Pretschner *et al.* [52]. The authors consider a similar technique for generating mutants from HLPSL specifications (closed to the *substitution* mutation operator defined in Section 4.4) and draw a relationship between the mutation operators and actual errors in the code, as presented in this article in Section 6. The question of test cases concretization is addressed subsequently [53], based on a semi-automatic technique proposed by Armando *et al.* [27]. Note that these latter both focus on protocols specified in the ASLan specification language [54]. This language differs from HLPSL in the sense that it is more concrete. Thus, mutations in this language are very similar to classical code mutation operators, while HLPSL mutations more specifically target the security functions of a protocol.

Finally, notice that testing security protocols are a complementary approach to the verification of protocols. Dozens of articles address the verification problem. The interested reader may refer to the survey from Cortier and *et al*. [16], in which many pointers on various aspects of protocols verification are given.

10. CONCLUSION AND FUTURE WORK

This article has presented a security protocol testing approach based on formal models written in HLPSL. In a first step, a relevant set of mutation operators, representing possible security faults in the implementation lifted at the model level, are applied on a safe HLPSL model to produce mutant

[††]http://www.nbs-system.com/wp-content/uploads/Advisory_Magento_Paypal.pdf.

DOI: 10.1002/stvr

protocols. The mutants are then analysed using the AVISPA tool (namely, the OFMC, SATMC and CT-AtSe back-ends). When declared unsafe, the tool produces an attack trace that can be used as a test case: if this attack trace can be replayed as is on the implementation of the protocol, then this implementation is unsafe. In the end, the attack trace, composed of message exchanges, is concretized into a executable test script using a dedicated framework that recreates the behaviour of the intruder, restores message data that were abstracted by the HLPSL model and decides the success or the failure of the test. This approach can be seen as a way to complement the verification of security protocols, by checking if a protocol is sensitive, or not, to a given error/implementation choice.

Both test generation and test execution parts were tool-supported and experimented at a large scale. Experimental results have discussed the relevance of the mutation operators that was initially proposed, evaluating their capability to produce unsafe mutant protocols. In addition, HOM was experimented but without success in the production of new security flaws despite the exhaustive combination of the proposed mutation operators. The approach has been illustrated on a case study of a real payment protocol, and an existing security flaw has been rediscovered by this process.

Current investigations consist in applying these techniques for testing vulnerabilities in e-commerce Web applications, in the context of the DAST national project[‡‡]. To achieve that, it is possible to reuse a large part of the development made for the Paypal case study, with slight modifications. In order to simulate attack scenarios on more complex protocol implementations with other merchants, there is an adaptation process that must been carried out before executing tests. Mainly, new modelled objects (e.g. agents field and events) must be mapped to implementation-level constructs (e.g. classes and methods in the programme). Regarding the protocol specification, one has to ensure that the attacker simulator module is able to execute all the steps stated by the protocol specification. Also, Web navigation between the pages of the Web application must be adapted to the online store implementation (clicks on buttons, form submissions etc.).

ACKNOWLEDGEMENTS

This article is an extended version of a conference paper published at the ICST 2011. A first extension concerns (besides some improvements of the presentation of the mutation operators) the experimental part of the approach for which more details are provided here, along with additional experiments and a clear relationship between the proposed mutation operators and real faults/implementation choices that can be made. A second extension concerns the addition of the test framework that is used to concretize the abstract test cases into executable ones. The authors would like to thank the reviewers and participants of the ICST in Berlin for their precious and encouraging comments on the preliminary work. Especially, we thank Alexander Pretschner and his team for the interesting discussions concerning the preliminary experimental results. This study is funded by the French Fonds National pour la Société Numérique DAST Project.

REFERENCES

1. Armando A, Basin DA, Boichut Y, Chevalier Y, Compagna L, Cuéllar J, Drielsma PH, Héam PC, Kouchnarenko O, Mantovani J, Mödersheim S, von Oheimb D, Rusinowitch M, Santiago J, Turuani M, Viganò L, Vigneron L. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *CAV*, vol. 3576, Etessami K, Rajamani SK (eds), Lecture Notes in Computer Science. Springer: Berlin, 2005; 281–285.
2. The high level protocol specification language: AVISPA project, Delivrable 2.1, 2003. Available from: http://www.avispa-project.org/delivs/2.1/d2-1.pdf [accessed on 18 April 2014].
3. Turuani M. The CL-Atse protocol analyser. In *Term Rewriting and Applications - Proceedings of RTA*, vol. 4098, Lecture Notes in Computer Science, Seattle, WA, USA, Springer-Verlag, 2006; 277–286.
4. Boichut Y, Héam PC, Kouchnarenko O. Approximation-based tree regular model-checking. *Nordic Journal of Computing* 2008; **14**:194–219.
5. Armando A, Carbone R, Compagna L, Cuellar J, Abad LT. Formal analysis of SAML 2.0 Web browser single sign-on: breaking the SAML-based single sign-on for Google apps. In *The 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*. ACM Press: Hilton Alexandria Mark Center, Virginia, USA, 2008; 1–10.
6. *Web-based reference implementation of SAML-based SSO for Google apps*: Google. Available from: https://developers.google.com/google-apps/sso/saml_reference_implementation_web [accessed on 18 April 2014].

[‡‡]http://dast.univ-fcomte.fr.

DOI: 10.1002/stvr

7. Falcone A, Focardi R. Formal analysis of key integrity in PKCS#11. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'10)*. Springer-Verlag, LNCS, March 2010; 77–94.
8. Bortolozzo M, Centenaro M, Focardi R, Steel G. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM Press: Chicago, Illinois, USA, October 2010; 260–269.
9. Pkcs #11 v.2.20: cryptographic token interface standard: RSA Security Inc, 2004.
10. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. *Computer* 1978; **11**(4):34–41.
11. DeMillo RA. Test adequacy and program mutation. *International Conference on Software Engineering,* Pittsburg, PA, USA, 1989; 355–356.
12. Budd TA, Gopal AS. Program testing by specification mutation. *Computer Languages* January 1985; **10**:63–73.
13. Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons: New York, USA, 1995.
14. Emerson EA, Clarke EM. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP'80)*, vol. 85, de Bakker JW, van Leeuwen J (eds), Lecture Notes in Computer Science. Springer-Verlag: Berlin, July 1980; 169–181.
15. Lowe G. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tacas*, vol. 1055, Margaria T, Steffen B (eds), Lecture Notes in Computer Science. Springer: Berlin, 1996; 147–166.
16. Cortier V, Delaune S, Lafourcade P. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security* 2006; **14**(1):1–43.
17. A Beginner's guide to modelling and analysing Internet security protocols: AVISPA project. Available from: http://www.avispa-project.org/package/tutorial.pdf [accessed on 18 April 2014].
18. Lamport L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 1994; **16**(3):872–923.
19. Dolev D, Yao AC. On the security of public key protocols. In *SFCS '81: Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society: Washington, DC, USA, 1981; 350–357.
20. Bozga M, Graf S, Ober I, Ober I, Sifakis J. The IF toolset. In *Formal Methods for the Design of Real-Time Systems*, vol. 3185, Bernardo M, Corradini F (eds), Lecture Notes in Computer Science. Springer: Berlin / Heidelberg, 2004; 131–132.
21. Basin DA, Mödersheim S, Viganò L. OFMC: a symbolic model checker for security protocols. *International Journal of Information Security* 2005; **4**(3):181–208.
22. Armando A, Compagna L. SAT-based model-checking for security protocols analysis. *International Journal of Information Security* January 2008; **7**(1):3–32.
23. Clark J, Jacob J. A survey of authentification protocol literature, University of York, UK, 1997.
24. IEEE 802.1 local and metropolitan area networks: wireless LAN medium acess control (MAC) and physical (PHY) specifications, 1999.
25. Zhou J, Gollman D. A fair non-repudiation protocol. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*. IEEE Computer Society: Washington, DC, USA, 1996; 55.
26. Millen JK, Shmatikov V. Constraint solving for bounded-process cryptographic protocol analysis. *ACM Conference on Computer and Communications Security,* Philadelphia, PA, USA, 2001; 166–175.
27. Armando A, Pellegrino G, Carbone R, Merlo A, Balzarotti D. From model-checking to automated testing of security protocols: bridging the gap. In *Tests and Proofs*, vol. 7305, Brucker AD, Julliand J (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2012; 3–18.
28. Jia Y, Harman M. Higher order mutation testing. *Information and Software Technology* October 2009; **51**(10): 1379–1393.
29. Lenstra AK, Hughes JP, Augier, Bos JW, Kleinjung T, Wachter C. Public keys. In *Advances in Cryptology—CRYPTO 2012*, vol. 7417, Safavi-Naini R, Canetti R (eds), Lecture Notes in Computer Science. Springer: Berlin, 2012; 626–642.
30. Avanesov T, Chevalier Y, Mekki MA, Rusinowitch M. Web services verification and prudent implementation. In *4th SETOP International Workshop on Autonomous and Spontaneous Security*, Lecture Notes in Computer Science. Springer: Leuven, Belgique, 2012.
31. Magento community edition. Available from: http://www.magentocommerce.com/ [accessed on 18 April 2014].
32. Wang R, Chen S, Wang X, Qadeer S. How to shop for free online—security analysis of cashier-as-a-service based Web stores. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11. IEEE Computer Society: Washington, DC, USA, 2011; 465–480.
33. Paypal development & integration guides. Available from: https://developer.paypal.com/webapps/developer/docs/classic/products/ [accessed on 18 April 2014].
34. XAMPP an easy to install Apache distribution. Available from: http://www.apachefriends.org/en/xampp.html [accessed on 18 April 2014].
35. Fiddler: the free Web debugging proxy for any browser, system or platform. Available from: http://www.telerik.com/fiddler [accessed on 18 April 2014].
36. Dadeau F, Héam PC, Kheddam R. Mutation-based test generation from security protocols in HLPSL. In *ICST'11*. IEEE Computer Society: Berlin, 2011; 240–248.

DOI: 10.1002/stvr

37. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 2010; **99**(PrePrints):649–678.
38. Okun V, Black P, Yesha Y. Testing with model checkers: insuring fault visibility. *WSEAS Transaction System* January 2003; **2**(1):77–82.
39. Hierons RM, Merayo MG. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software* 2009; **82**(11):1804–1818.
40. Fabbri SCPF, Delamaro ME, Maldonado JC, Masiero P. Mutation analysis testing for finite state machines. *Proceedings of the 5th International Symposium on Software Reliability Engineering*: Monterey, California, 1994; 220–229.
41. Fraser G, Wotawa F. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*. IEEE Computer Society: Los Alamitos, CA, USA, 2006: 16–26.
42. Probert RL, Guo F. Mutation testing of protocols: principles and preliminary experimental results. *Proceedings of the Workshop on Protocol Test Systems*: Leidschendam, Netherland, 15-17 1991 October; 57–76.
43. Bayse E, Cavalli A, Núñez M, Zaïdi F. A passive testing approach based on invariants: application to the WAP. *Computer Networks* 2005; **48**(2):247–266.
44. Ural H, Xu Z, Zhang F. An improved approach to passive testing of FSM-based systems. In *Ast '07: Proceedings of the Second International Workshop on Automation of Software Test*. IEEE Computer Society: Washington, DC, USA, 2007; 6.
45. Arnedo JA, Cavalli A, Núñez M. Fast testing of critical properties through passive testing. In *TestCom'03: Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems*. Springer-Verlag: Berlin, Heidelberg, 2003; 295–310.
46. Lee D, Sabnani K, Kristol D, Paul S. Conformance testing of protocols specified as communicating finite state machines—a guided random walk based approach. *IEEE Transactions on Communications* 1996; **44**(5):631–640.
47. Shu G, Lee D. Testing security properties of protocol implementations—a machine learning based approach. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*. IEEE Computer Society: Washington, DC, USA, 2007; 25.
48. Shu G, Lee D. Message confidentiality testing of security protocols—passive monitoring and active checking. In *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference (TESTCOM 2006)*, vol. 3964, Uyar MU, Duale AY, Fecko MA (eds), LNCS. Springer: Berlin, 2006; 357–372.
49. Wimmel G, Jürjens J. Specification-based test generation for security-critical systems using mutations. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*. Springer-Verlag: London, UK, 2002; 471–482.
50. Huber F, Schätz B, Schmidt A, Spies K. AutoFocus: a tool for distributed systems specification. In *Ftrtft '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag: London, UK, 1996; 467–470.
51. Aslam T, Krsul I, Spafford EH. Use of a taxonomy of security faults. *19th NIST-NCSC National Information Systems Security Conference,* Baltimore, MD, USA, 1996; 551–560.
52. Büchler M, Oudinet J, Pretschner A. Security mutants for property-based testing. In *Proceedings of the 5th International Conference on Tests and Proofs*, TAP'11. Springer-Verlag: Berlin, Heidelberg, 2011; 69–77.
53. Büchler M, Oudinet J, Pretschner A. Semi-automatic security testing of Web applications from a secure model. In *Sixth International Conference on Software Security and Reliability (SERE 2012)*. IEEE: Washington, DC, USA, 2012; 253–262.
54. AVANTSSAR. *Delivrable 2.1: requirements for modelling and aslan v1*, 2008. Available from: http://www.avantssar.eu [accessed on 18 April 2014].

DOI: 10.1002/stvr