Message Confidentiality Testing of Security Protocols – Passive Monitoring and Active Checking*

Guoqiang Shu and David Lee

Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA
{shug, lee}@cse.ohio-state.edu

Abstract. Security protocols provide critical services for distributed communication infrastructures. However, it is a challenge to ensure the correct functioning of their implementations, particularly, in the presence of malicious parties. We study testing of message confidentiality – an essential security property. We formally model protocol systems with an intruder using Dolev-Yao model. We discuss both passive monitoring and active testing of message confidentiality. For adaptive testing, we apply a guided random walk that selects next input online based on transition coverage and intruder's knowledge acquisition. For mutation testing, we investigate a class of monotonic security flaws, for which only a small number of mutants need to be tested for a complete checking. The well-known Needham-Schroeder-Lowe protocol is used to illustrate our approaches.

1 Introduction

Security protocols have been playing an important role in the critical distributed systems such as E-Commerce and military infrastructure. Most security protocols use cryptography to achieve data transmission, authentication and key distribution [16, 17] in a hostile environment [1]. The existence of diverse intruders renders the resilience of those protocol systems more significant, and more challenging. Various formal modeling and analysis techniques, such as BAN logic, model-checking and strand spaces [14, 18, 19] have been developed in the recent years to ensure the correctness of security protocol system. These works are focused on validating the protocol specification. However, errors can also be introduced to the system in implementation phases, even if the specification is proven to be flawless. Furthermore, interconnected communication system interfaces may result in security problems, such as message content exposure. Systematic testing approaches for security protocols have been largely neglected by the research community, even though numerous reports show programming errors in security-critical systems are very common [22,23].

Testing for system security, often known as penetration testing [22] or red-team testing, refers to the activity of executing a predefined test script with the goal of finding a security exploit. Thomson in [23] classified four general penetration testing

* This work was supported in part by the U.S. National Science Foundation (NSF) under grant awards CNS-0403342, CNS-0548403, and by the U.S. Department of Defense under grant award N41756-06-C-5541.

M.Ü. Uyar, A.Y. Duale, and M.A. Fecko (Eds.): TestCom 2006, LNCS 3964, pp. 357 – 372, 2006.
© IFIP International Federation for Information Processing 2006

methods: (1) Testing dependency; (2) Testing unanticipated user input; (3) Expose design vulnerabilities; and (4) Expose implementation vulnerabilities. Under these guidelines practical testing has been conducted in industry and proved to be very helpful. Nonetheless, most of the current penetration testing activities are ad-hoc and rely on expert knowledge of target systems or existing exploits [7]; the cost of a comprehensive testing is high and the response time is too long. On the other hand, current testing methods are largely at system level on system misconfiguration [20] or unexpected side effect of operations [2]. Protocol level penetration testing has not drawn adequate attention yet is crucial for discovering security protocol implementation errors. Particularly, automated test selection and execution techniques are desirable for complex protocols and for real-time response to security flaws.

In this paper we focus on automated testing of the key property of security protocols: message confidentiality. Several unique characteristics of security protocols make the traditional conformance testing approaches insufficient and pose new challenges for both modeling and test generation tasks. First, security protocols have a huge and special data portion. The I/O messages are from a language defined by cryptographic primitives such as public/private encryption and decryption. The formidable size of the alphabet makes generating a complete checking sequence infeasible. Therefore, tradeoff is usually made to focus only on a special type of nonconformance – security flaws. We use Extended Finite State Machine (EFSM) based formal model [5, 12] to specify the security protocol and augment the model to include security protocol message types as the parameter of I/O symbols. On the other hand, security properties can be tested only with a precise intruder model. We use EFSM to formally specify the intruder's behaviors based on the well-known Dolev-Yao model [4], which models most powerful and yet realistic intruder. Consequently, the whole protocol system is modeled as the communication system composed of the intruder and a set of legitimate principals. Furthermore, we define the notion of intruder's knowledge and message confidentiality requirement, and use it as the goal of testing.

Based on this formal model several testing approaches are proposed. We first give a simple passive monitoring procedure and then describe an active guided random walk algorithm. The algorithm is inspired by the earlier work [11] where heuristics are used to achieve high coverage of transitions in a CFSM model. Here our algorithm uses a new heuristic transition selection criterion that favors both new transition and new knowledge acquisition by the intruder. Both testing algorithms are unstructured in terms of the global system model, and the composite EFSM does not need to be computed. We also study mutation testing, since it is known to be efficient for a range of particular types of errors in software testing [3, 15]. Wimmel's et al's work based on their elegant validation tool AutoFocus [9, 24] is among the first attempts to apply the idea of mutation testing to security system. The greatest challenge (unaddressed by [9, 24]) of mutation testing is to control the number of mutants. This paper defines mutation functions with special property such that only mutants with single fault need to be considered for test generation. As a case study, we model the predicate (guard) absence fault type F_{PA} with this property, then present and analyze the test generation algorithm. We use the well-known Needham-Schroeder-Lowe (NSL) mutual authentication protocol [18] to illustrate our formal model and testing algorithms.

2 Modeling and Methodologies

After describing a formal model of security protocol systems we present our testing methods for both passive monitoring and active testing for message confidentiality.

2.1 Security Protocol Model

We define the security protocol message type as follows. First, there are three atom types: *Int*, *Key* and *Nonce*. A value of type *Int* is a non-negative bounded integer. A value of type *Key* ranges over a finite set *K* of keys. A value of type *Nonce* ranges over a finite set *N* of nonces. A protocol message is recursively defined as: (1) An atom; (2) Encryption of a message with a key; or (3) Concatenation of atoms and encryptions. A message can be represented by a string. For example $E(k_b,(k_a.n_a))$ is a messages that is formed by encrypting the concatenation of k_a and n_a with another key k_b. Given A=<K, N>, a set of keys and nonces, denote $L(A)$ as the message type and the set of messages formed using atoms in A. $L(A)$ is obviously infinite, and even if we restrict the number of atoms that a message contains, its size is exponential.

Among the atom types, *Key* and *Nonce* are treated as symbols in the sense that they can not be composed or calculated using other atom values. Also, *Key* type contains both symmetric keys and asymmetric key pairs. For the latter we use *ku* to represent a public key and *kr* for a private key. There are some basic operations defined on message type. Let *msg* be a message, function *Elem*(*msg*,*i*) calculates the *i*th component in *msg*, and *D*(*k*,*msg*) returns *m'* when *msg* = *E*(*k*,*m'*). Both functions are partial and they are undefined for messages with incompatible formats.

We define an extended finite state machine model that uses protocol message set $L(A)$ as input and output alphabet.

Definition 1. An Extended Finite State Machine (EFSM) with symbolic message type is a 7-tuple M=<S, s_{init}, A, I, O, X, T> where

1. S is a finite set of states;
2. s_{init} is the initial state;
3. A is the set of atoms, and $L(A)$ is the set of messages formed using atoms in A;
4. $I = \{i_0, i_1,\ldots, i_{P-1}\}$ is the input alphabet of size P; each input symbol i_k $(0 \le k < P)$ contains a parameter $\pi(i_k)$ of type $L(A)$;
5. $O = \{o_0, o_1,\ldots, o_{Q-1}\}$ is the output alphabet of size Q ;each output symbol o_k $(0 \le k < Q)$ contains a parameter $\pi(o_k)$ of type $L(A)$;
6. X is a vector denoting a finite set of variables of type $L(A)$;
7. T is a finite set of transitions; for $t \in T$, t = <s, s', i, o, $p(x, \pi(i))$,$a(x, \pi(i), \pi(o))$> is a transition where s and s' are the start and end state, respectively; $\pi(i)$ and $\pi(o)$ are the input/output symbol parameters; $p(x, \pi(i))$ is a predicate, and $a(x, \pi(i), \pi(o))$ is an action on the current variable values and parameters.

For practical protocol systems, the machine is often partially specified because a transition can only be triggered by a message with expected format. We use predicates to model the basic type checking capability. Upon receiving a message, the machine can reconstruct each element of it if the message format is correct. The special case is that some of the encrypted messages might be opaque to a machine

because it does not possess the required key. For each combination of state and input symbol, there is one transition with a special predicate "else", meaning that it is enabled when all other transitions are not. We further assume that upon an input if none of the transitions are triggered, and then an implicit "else" transition make the machine stay at the current state and output nothing. For simplicity, we assume the machine contains a reliable reset symbol that takes the machine back to the initial state s_{init} and resets all state variables. In order to model the global uniqueness of nonce, a fresh new set of nonce $N' \subseteq N$ will be used whenever the machine is reset, so that different runs (sessions) of the protocol will use different nonces. Since N is finite we can only model finite number of sessions and each session only uses finite nonces. Finally, we only consider deterministic EFSM model.

A security protocol is specified by a set of communicating EFSM $\{M_1,M_2,\cdots,M_C\}$ that share the same message type L. Each component machine M_k represents a principal in the protocol system. It is possible that two machines are the same, meaning there are symmetric peers in the protocol. Moreover, for each transition in a component machine M_k, the input (output) symbol carries an extra parameter of the sender (receiver) identifier. We denote an input message received from M_a as $M_a?i$ and an output to M_a as $M_a!o$. The semantics of message sending/receiving follow the typical communicating FSM model [11]: the input/output is synchronized as a rendezvous and executed simultaneously.

2.2 Intruder Model

We model an intruder as an additional EFSM M_I in the protocol system that runs a special protocol. To model general behaviors of the intruder, we adapt Dolev-Yao's assumptions for two party message exchange protocols [4] that define a widely accepted powerful intruder model. It has been proved that one intruder poses the same security threat as multiple intruders and we model only one in our study.

An intruder is first a legitimate principal of the communication system; it can not only initiate a session with any other component machine M_a but also be the (passive) peer of any session. Furthermore, the intruder is capable of intercepting messages between any two legitimate principals. The important effect of this behavior is that the semantics of message sending and receiving in the original communicating EFSM model are altered. A transition in M_a with output message $M_b!msg$ now will be jointly executed with a transition in M_I that takes input $M_a \rightarrow M_b?msg$, instead of the transition in the intended receiver M_b. This should be clearly distinguished with the first case where the intruder M_I is the intended receiver (e.g. M_a outputs $M_I!msg$). Similarly, the intruder can inject any message, impersonating any other machines. That is, M_I can send output $M_a \rightarrow M_b!msg$ to M_b and this output matches the transition of M_b with input $M_a?msg$.

Besides the capability of catching and injecting normal protocol traffic, the intruder is also assumed to be able to generate any new message based on all and only the messages it possesses. Formally, we define the knowledge of the intruder as a set of messages $\Omega = Encl(\Omega_0+MSG)$ where Ω_0 is the initial knowledge known to the intruder containing the public and intruder's own information, and MSG represents the set of messages the intruder has received. Function $Encl(L)$ is defined as the enclosure of L under the functions $Elem()$, $D()$ and $E()$. Therefore, Ω can be regarded as all the mes-

sages that the intruder is able to construct, using only the messages it obtains. Once the intruder gains a message it will not forget it and Ω never shrinks. As far as a realistic testing scenario is concerned, we have to assume the intruder has the capability of recognizing the message format, either by guessing the data field, or by reading the meta-info such as an XML schema.

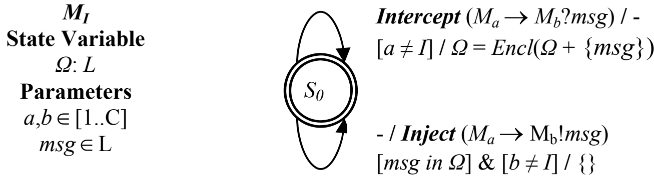


Fig. 1. EFSM model for the intruder

Fig. 1 shows the EFSM model of the intruder. M_I contains only one state and two transitions for message interception and injection respectively. *Intercept* transition takes any message *msg* sent from M_a to M_b as input. The guard ensures *msg* is not from M_I itself and the action updates the knowledge set. *Inject* transition outputs a message in the current knowledge set to another machine M_b under the disguise of M_a. The model of M_I is obviously independent of the other component machines. Note that messages meant to be delivered to M_I will also be caught by the *Intercept* transition, and in case the intruder does not want to intercept a message, the *Inject* transition is fired right after *Intercept* transition with the same message.

2.3 Testing Security Requirement: Message Confidentiality

Given the specification of protocol roles $\{M_1, M_2,\ldots, M_C\}$ and the intruder M_I, the global behavior of the whole protocol system under investigation is described by the Cartesian product of all the machines: $M_1 \times M_2 \times \cdots \times M_C \times M_I$ with the input/output matching rule we define in the previous subsection. Since all the transitions involve one of the two intruder transitions, an I/O trace produced by the system can be described by an interleaving sequence of *Intercept* and *Inject* transitions each with a message in the parameters.

In this paper we focus on black-box penetration testing. The implementations of all protocol principals are treated as pure black boxes, i.e. $B = \{B_1, B_2,\ldots, B_C\}$, each B_i can be a correct or faulty implementation of M_i. The tester plays the role of intruder and simulates the machine M_I. This is an active testing process because the tester can choose arbitrarily the parameters of *Inject* transition, namely the sender, receiver and message. A test sequence *seq* is defined as an I/O trace produced by the communicating system of M_I and B. Starting from the initial states, denote the value of Ω in M_I after a test sequence *seq* is applied as $\Omega(B, seq)$, which is the knowledge that an intruder gains by performing penetration test *seq*.

For a given security protocol system, there are many security requirements depending on the specific application needs. Typically, they include message confidentiality, message integrity, authentication, and non-repudiation [20]. In this paper we focus on

the message confidentiality requirement that is the key property of a security protocol system. Other requirements can be handled, for instance, by appropriate hash functions, and we shall not digress here.

Definition 2. A protocol implementation $B = \{B_1, B_2, \ldots, B_C\}$ is insecure with regard to the confidentiality of messages $M^* \subset L$ if and only if there exists a test sequence *seq* and a message $m \in M^*$ such that $m \in \Omega(B, seq)$.

An implementation is flawed if and only if message content can be uncovered by the intruder after a test sequence is applied. We model the intruder following Dolev-Yao's approach, our definition 2 is consistent with their notion of security of two party cascade and name stamp protocols [4]. One natural question is that whether the protocol specification itself is secure. When the implementation of each component machine is equivalent to its specification, i.e. $B_i = M_i$, the intruder might still be able to obtain the secret if the protocol design itself is flawed [14]. Since our goal is testing rather than validation, we assume the protocol design and specification are secure.

As an example of modeling security protocols, we consider the well-known Needham-Schroeder-Lowe (NSL) mutual authentication protocol [18]. Among many of its variants, we use a simplest one with three message exchanges [14]. Two principles, the initiator and the responder, are involved and they are specified as M_A and M_B, respectively. The message sequence of a successful run is shown below.

$A \rightarrow B$ (Ask): $A.B.E(KU_B,(N_A.A))$
$B \rightarrow A$ (Rpl): $B.A.E(KU_A, (N_A.N_B.B))$
$A \rightarrow B$ (Cfm): $A.B.E(KU_B, (N_B))$

The protocol functions as follows. The initiator A encrypts a nonce with the responder B's public key and sends it to B. B then decrypts it and encrypts it together with another nonce using A's public key. Finally A gets the second nonce and sends it back. The purpose of NSL protocol is to allow both principles authenticate each other and exchange some secrets (two nonces), which later on can be used to construct shared keys. Fig. 2 shows the two complete EFSM specifications. We assign index 0, 1 and 2 to M_A, M_B and the intruder M_I. The intruder can participate legally as both the initiator and responder. The atom messages in this protocol include the public keys (KU_A, KU_B and KU_I) and the nonces. In order to express the security requirement conveniently, we distinguish the nonces used for different peers. For instance, the nonce M_B uses to challenge M_I is $N_B[I]$. Initially the intruder only knows its own key and nonces, i.e., $\Omega_0 = \{KU_I, N_I[A], N_I[B]\}$. The secret message set is $M^* = \{N_A[B], N_B[A]\}$; the intruder should not obtain the nonces that are only supposed to be shared only between A and B. Note that in Fig. 2 the parameter (message) of each I/O symbol is expanded by its structure, which is a short notation for format checking of the message. A special symbol Rst is used to reset the session when invalid message is processed.

To summarize this section, we essentially reduce the security testing problem to searching for special I/O sequences produced by a mixed communicating system, which contains M_I and one or more black boxes as principals. The characteristic of those sequences is that they lead the reachability graph of M_I to a state where the value of variable Ω contains message content/secret. The tester has full control over M_I but can only observe the I/O behaviors of the other protocol principals.

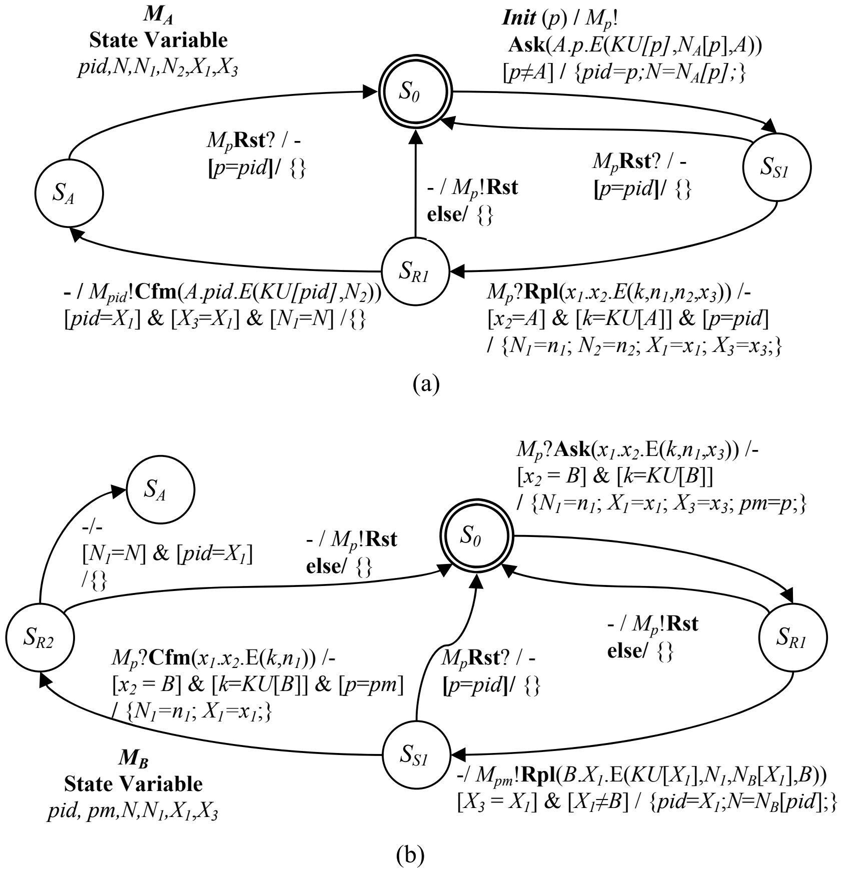


Fig. 2. Needham-Schroeder-Lowe protocol (a) Initiator M_A (b) Responder M_B

3 Message Confidentiality Testing

After presenting a simple passive monitoring algorithm, we describe an active testing procedure that is based on a guided random walk.

3.1 A Simple Passive Monitoring Algorithm

A passive tester or monitor of security protocol implementation is easy to devise. The intruder (tester) intercepts all messages among the component machines, updates its knowledge, replies if the message is directed to itself, and otherwise forwards it without any modification. The testing terminates when the intruder derives any secrets. The procedure is shown in Algorithm 1. As inherent to all passive testing approaches, this algorithm only utilizes part of the intruder's capability and it is suitable when the intruder could only conduct eavesdropping [1].

Algorithm 1 (Passive Monitoring)
Input: $\{B_1, B_2,\ldots, B_C\}$, message secrets M^*, Intruder initial knowledge Ω_0.
Output: security flaw if observed.
begin
1. $\Omega=\Omega_0$;
2. while (true)
3. try to execute *Intercept* ($B_i \rightarrow B_j?msg$) transition with any B_i;
4. if succeed
5. if ($M^* \cap \Omega \neq \phi$) return *flaw*;
6. if ($j=I$)
7. generate reply *msg'* ;
8. execute *Inject* ($M_I \rightarrow B_i?msg'$) transition;
9. else
10. execute *Inject* ($B_i \rightarrow B_j?msg$) transition;

End

3.2 Active Testing – Guided Random Walk

Now we study active testing approaches that utilize the full power of the intruder. One simple-minded method of active testing is random walk. Starting with an initial knowledge set, the intruder (tester) randomly chooses either to intercept a message from a pair of principals or to construct a message using its current knowledge and send it to a principal. Pure random walk has several limitations; the coverage of the model is not high and, more importantly, it does not use the intruder's knowledge acquired. We present a guided random walk approach with a high coverage and fully utilizing the intruder's knowledge acquired.

The approach is adaptive and unstructured in terms of the composite (global) state machine. We keep track of the current state S_i and variable values X_i for each black box B_i in order to guide the selection of next transitions. Note that in general tracking current state is not always possible even under the assumption that B_i contains no transition errors; it is due to the fact that part of the message is encrypted and intruder can not utilize the information to infer the current transition and state if he does not have the key. In this case, the algorithm makes a random guess.

At each step, the intruder always tries to intercept the messages coming from every machine B_i. Once a message is intercepted, the state of the sender as well as the intruder's knowledge is updated. Then the intruder constructs a message and injects it to a machine to fire a carefully selected transition. Our algorithm selects transition and message based on the following criteria. First the transitions of all component EFSM should be covered fairly. The algorithm keeps track of a counter *cnt*[*t*] for each transition *t,* and at each step the one that has been executed least is favored. Moreover, we only select the transitions that could possibly be enabled by some input message and ignore those transitions that will definitely not be triggered (the current state variables themselves disable the predicate). We calculate T_{true} as the set of all possible transitions:

$$T_{true} = \{\ t\langle S_i, S'_i, I, p, a\rangle \mid t \in M_i \text{ and } \exists\, msg \in \Omega: p(X_i,msg) = true\}$$

Once a transition t is determined, we construct an enabling input message for t using a greedy algorithm. Ideally we want an input message that will lead the machine to a state that can generate more new knowledge. That is, for all candidate messages we calculate the destination state S' of t, and select one that enables at least one output transition t' with parameter msg' not in Ω. We use subroutine $lookahead(\Omega, S, X, t)$ to calculate such messages. If such messages do not exist or there are ties, an enabling message is randomly picked:

$lookahead(\Omega, S, X, t\langle S_i, S'_i, I, p, a\rangle) = \{msg \mid p(X_i, msg) = true$ and $(\exists\, t'\langle S_i', S_i'', O(msg'), p', a'\rangle : p'(X_i') = true$ and $msg' \notin \Omega)\}$

Algorithm 2 (Active Testing - Guided Random Walk)
Input: $\{B_1, B_2, \ldots, B_C\}$, secrets M^*, Intruder initial knowledge Ω_0.
Output: Adaptive test sequence.
begin
1. initialize each M_i , for all transition t, $cnt[t] = 0$;
2. $X=\langle X_1,\ldots,X_C\rangle$, $S=\langle S_1,\ldots,S_C\rangle$;
3. $\Omega=\Omega_0$, $seq=\phi$;
4. while (seq.len < L)
5. foreach component B_i
6. try to execute $Intercept(B_i \rightarrow B_j?msg)$ with B_i;
7. if succeed
8. deduce or guess the transition t;
9. update X_i, S_i, $cnt[t] = cnt[t] + 1$, $seq = seq + \{t\}$;
10. calculate T_{true} ,select $t \in T_{true}$ with smallest $cnt[t]$;
11. select msg from $lookahead(\Omega, S, X, t)$;
12. try to execute *Inject* transition with t using msg;
13. if succeed
14. update X_i, S_i, $cnt[t] = cnt[t] + 1$, $seq = seq + \{t\}$;
15. if ($M^* \cap \Omega \neq \Phi$) return seq;
16. return seq;

end

To avoid infinite tests, the algorithm terminates when either the secret message content is obtained or the length of test sequence reaches a preset limit. This algorithm is more effective than random walk because the greedy heuristics take into account both coverage and intruder knowledge acquisition. However, it still has many inherent limitations. For example, calculation of T_{true} and *lookahead*() is rather expensive. Also, the effectiveness of the heuristic relies on the estimation of current state and variable values, and if it fails the algorithm behaves the same as random walk. Advanced passive testing techniques [8, 10] that estimate data portion more accurately could be applied here to improve the performance.

3.3 Experiment

We conduct an experiment of Algorithm 2 on NSL protocol specified as Fig. 2. Two implementations are created with a common programming error in each. Then we treat them as black-boxes and run the algorithm to test for confidentiality violations.

Implementation X: The responder does not verify the encrypted identifier of the initiator after it receives Ask message, and proceeds as if it were correct.

Implementation Y: The initiator does not verify the encrypted identifier of the responder after it receives Rpl message, and proceeds as if it were correct. This error was first uncovered by Lowe [14] as a design flaw in the original Needham-Schroeder protocol.

For both Implementation *X* and *Y* errors have been detected. Table 1 (a) and (b) show the successful test sequences for them. In the first test sequence, at the beginning the intruder intercepts an Ask message from M_0 to M_1, and updates the state to $<S_{s1},S_0>$. Now three transitions are feasible and as the result M_1?Ask is selected. *Lookahead*() returns a random message that enables M_1?Ask because no message will further trigger an output transition. In the second round we intercept an Rpl message, and the intruder will obtain a secret ($N_0[1]$) and terminate the test. The sequence for *Y* is more complex. After injecting an Ask message to M_1 and intercepting the response, we have two transitions in T_{true}. $M_1!Cfm$ is chosen and executed with a random message. At next step M_0 happens to initiate a session with M_1. This is a rare event yet critical for detecting errors in this implementation. The only transition that could be enabled is M_0?Rpl, and now the intruder happens to have a message to enable it. The last step is the interception of Cfm message from M_0 that exposes the nonce – secret $N_1[0]$.

Table 1. Detection of Errors in Implementation *X* (a) and *Y* (b)

States	**Action**	**Note**
$<S_0, S_0>$	*Intercept* $M_0 \rightarrow M_1$? Ask $(0.1.E(KU[1], N_0[1], 0))$	$\Omega^+ = \{E(KU[1],N_0[1],0)\}$
$<S_{S1}, S_0>$	*Inject* $M_2 \rightarrow M_1!$ Ask $(2.1.E(KU[1], N_0[1], 0))$	$T_{true} = \{M_0?\text{Rpl}, M_0?\text{Rst}, M_1?\text{Ask}\}$ $t = M_1?\text{Ask}$
$\underline{<S_{S1}, S_{R1}>}$	*Intercept* $M_1 \rightarrow M_2$? Rpl $(1.2.E(KU[2], N_0[1], N_1[2],1))$	$\Omega^+ = \{N_0[1], N_1[2]\}$ $N_0[1] \in M^*$

(a)

States	**Action**	**Note**
$<S_0, S_0>$	*Inject* $M_0 \rightarrow M_1!$ Ask $(0.1.E(KU[1], N_2[1], 0))$	$T_{true} = \{M_1?\text{Ask}\}$ $t = M_1?\text{Ask}$
$<S_0, S_{R1}>$	*Intercept* $M_1 \rightarrow M_0$? Rpl $(1.0.E(KU[0], N_2[1], N_1[0], 1))$	$\Omega^+ = \{ E(KU[0], N_2[1], N_1[0], 1)\}$
$<S_0, S_{S1}>$	*Inject* $M_0 \rightarrow M_1!$ Cfm $(0.1.E(KU[1], N_2[1], 0))$	$T_{true} = \{M_1?\text{Rst}, M_1?\text{Cfm}\}$ $t = M_1?\text{Cfm}$
$<S_0,S_{R2}>$	*Intercept* $M_0 \rightarrow M_2$? Ask $(0.2.E(KU[2],N_0[2],0))$	$\Omega^+ = \{N_0[2]\}$
$<S_{S1},S_{R2}>$	*Inject* $M_2 \rightarrow M_0!$ Rpl $(2.0.\ E(KU[0], N_2[1], N_1[0], 1))$	$T_{true} = \{M_0?\text{Rpl}\}$ $t = M_0?\text{Rpl}$
$\underline{<S_{R1},S_{R2}>}$	*Intercept* $M_0 \rightarrow M_2$? Cfm $(0.2.E(KU[2],N_1[0],0))$	$\Omega^+ = \{N_1[0]\}$ $N_1[0] \in M^*$

(b)

4 Mutation Testing

In this section we investigate mutation testing of security protocol, and design structured and preset test sequences. As introduced earlier mutation testing is a powerful technique for detecting specific types of security errors. Given the specification $M_{spec} = \{M_1, M_2,\ldots, M_C\}$, we introduce some faults, resulting in a mutant $\{M_1', M_2',\ldots, M_C'\}$. Given a set of mutants P, a test suite is generated such that for each mutant p, there is at least one test sequence that distinguishes (detects) it with the specification (correct implementation). A main challenge of mutation testing, when applied to software in general, is that the number of mutants (therefore the number of tests required) is huge. The situation is not mitigated in our EFSM model given its equivalent computing power of Turing machine. We model a security flaw as a mutation function δ on a specification EFSM, and a type of fault F as a set of similar mutation functions. A mutant under F is the application of one or more such functions. If the type F contains k functions, then the number of mutants is $O(2^k)$.

One can take two hypotheses to reduce the number of mutants generated [3]. First, competent programmer hypothesis assumes that an implementation only contains a small number (C) of faults. This reduces the number of mutants to $O(k^C)$, which is still quite large. Second, coupling effect hypothesis states that the test sequences used to distinguish mutants with simple fault are sensitive enough to also uncover complex fault. Clearly this is not always true. Given an arbitrary mutation function, a test sequence that obtains the secret on $\delta_1(M_{spec})$ may not be effective for $\delta_2\delta_1(M_{spec})$. In fact, mutant $\delta_2\delta_1(M_{spec})$ could even be secure. On the other hand, if we could select test sequence that satisfies this property, then the number of mutants could be further reduced to k. For message confidentiality testing, we can reduce the number of mutants based on this observation.

4.1 A Fault Model: Predicate or Guard Absence

There are generally two categories of security sensitive fault in the protocol model. The first is message format fault. For example, one might use the private key to encrypt part of the message instead of the public key, or attach an unnecessary part, both giving the intruder more information. This type is easier to observe since it changes the alphabet of some component machines. The second category of fault is related to the predicate or action of the transitions, but has no effect on the message types. Based on the observation of security protocols, a commonly encountered implementation error is neglecting critical condition checking. Usually an action is taken place only if some condition – predicate - is satisfied by the current state and/or the input message. For example in the NSL protocol, the responder only replies to the message Ask($x_1.x_2$.E(k,n_1,x_3)) when the x_2 is equal to its own index, and similarly the initiator only generates to the Cfm message when it verifies the responder's reply with the same nonce as the one it sends out. If the programmer neglects to check such condition such as in Implementation X and Y in section 3, it is likely that the resulting implementation is insecure. This type of fault is reflected in the EFSM model as the absence of part of the predicate in a transition - or often called a guard. Assuming the predicate is specified as a conjunctive normal form of Boolean expressions (i.e. $b_1\&b_2\&b_3$) , we formally define this fault type.

Definition 3. For all the transitions t_j, $j=0,1,\ldots$, from a state s with a same input/output symbol y, a predicate absence (PA) mutation function $\delta_{PA}(s,y,t,b)$ with regard to a Boolean expression b in the predicate p_j of $t=t_i$, is obtained by removing b from p_j and adding $(!p_i)$ to p_j for all $i \neq j$.

Basically the mutation function removes one Boolean expression from a transition. In order to keep the resulting machine deterministic, we add its negation to all other transitions with the same start state and input/output symbol. Fig. 3 shows an example of a mutant of the function $\delta_{PA}(S_1, Y, t, [a{=}1])$.

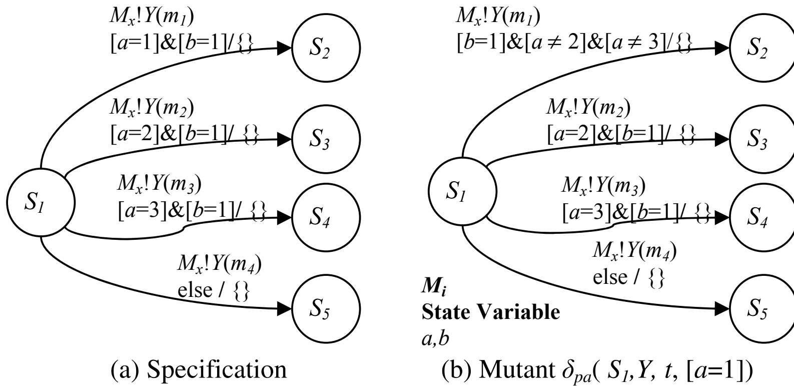


Fig. 3. Example of mutant δ_{PA}

Definition 4. For a protocol specification M_{spec}, a predicate absence (PA) fault type F_{PA} is obtained by applying one or more PA mutation functions $\delta_{PA}(s, y, t, b)$ on M_{spec}. A mutant under F_{PA} is defined as $\delta_S(M_{spec}) = \delta_1\delta_2\ldots\ \delta_n(M)$, where $S = \{\delta_1, \delta_2, \ldots, \delta_n\} \subseteq F_{PA}$, and for any $\delta_a(s, y, t_a, b_a), \delta_b(s, y, t_b, b_b) \in S$, $t_a = t_b$.

A mutant under the PA fault type is the result of application of a set of PA mutation functions, each removing a Boolean expression from a predicate. Note that although this definition does not limit the number of faults in one mutant, it relies on the competent programmer hypothesis to assume that for each combination of component machine, state and I/O symbol, only a predicate from one transition could be removed. Consequently, if each transition contains a constant number of Boolean expressions, there are totally $O(T)$ mutation functions and $O(2^{(C \times N \times P)})$ mutants where T is the number of transitions, C is the number component machines, N is the maximum number of states and P is the number of I/O symbols.

Intuitively a mutant with more predicate missing should allow more transitions to be executed and therefore the security flaws are "monotonically" increasing with inclusion of more faults in F_{pa}. This is formulated in the following proposition.

Definition 5. A progressive I/O sequence of a communicating system is an I/O sequence that does not trigger any "else" transition of any component machine.

Proposition 1 (Monotonicity). For any two mutants $\delta_{S1}(M)$ and $\delta_{S2}(M)$ under F_{pa} with $S_1 \subseteq S_2$, if a progressive I/O sequence *seq* could be generated by M_I and $\delta_{S1}(M)$, then *seq* could also be generated by M_I and $\delta_{S2}(M)$.

Sketch of proof: The proof of this proposition is quite straightforward using an induction on the length of the sequence. Suppose a prefix of *seq* has already been executed by $\delta_{S2}(M)$ and the next message in *seq* will trigger transition *t* in M_i. if $\delta_{S2}(M)$ has the same *t* as $\delta_{S1}(M)$ then *t* will be executed. If $\delta_{S2}(M)$ further removes some expressions from t, then the current states and input message will satisfy the guard of the new transition, since *t* is not the "else" transition, and, therefore, *t* is executed.

An important implication of Proposition 1 is that if a progressive test sequence discovers a message secret for *M*, and we apply some other mutation functions to introduce more errors, the same test sequence can still expose the message content on the new mutant. In other words, faults do not cancel the evidence of each other with regard to a progressive test sequence. We remark that singularity about "else" transition does not decrease the applicability of this model because this special type of transition is usually used to model the behavior in abnormal conditions, and will not be included in an I/O sequence that achieves the functionalities of the protocol.

4.2 Mutation Test Generation Algorithm

Now we describe the procedure of generating test sequences for monotonic flaw type of F_{PA}. The goal is to generate a set of test sequence that distinguishes all mutants under F_{PA}. One valid concern would be that not all mutants are necessarily insecure according to the confidentiality requirement and it is reasonable to only focus on mutants, which lead to message confidentiality violations. This is a well-studied validation problem and we shall not digress here. For simplicity, we treat all mutants as potentially insecure and generate tests to detect each of them:

Algorithm 3 (Test Generation for Fault Type F_{PA})
Input: $M_{spec} = \{M_1, M_2,\ldots, M_C\}$, secrets M^*.
Output: test suite *S*, fault type F'_{PA}
begin
1. $S = \{\}$; $F'_{PA} = \{\}$;
2. remove all "else" transitions from M_{spec}
3. calculate and minimize $M_I \times M_{spec}$;
4. foreach mutation function δ_i
5. calculate $\delta_i(M_{spec})$;
6. calculate and minimize $M_I \times \delta_i(M_{spec})$;
7. if $(M_I \times M_{spec} \mathrel{!}= M_I \times \delta_i(M_{spec}))$
8. t = separating sequence of $M_I \times M_{spec}$ and $M_I \times \delta_i(M_{spec})$
9. $S = S + \{t\}$;
10. $F'_{PA} = F'_{PA} + \{\delta_i\}$;
11. return *S*
end

Algorithm 3 applies each mutation function alone to the specification and calculates a progressive separating sequence. This is done by removing all "else" transitions, minimizing the Cartesian product of the mutant and intruder machine, and calculate a separating sequence. The comparison in Line 7 refers to an equivalence test of two machines. The algorithm produces a new fault type F'_{PA} which only contains the mutation functions if the corresponding mutants are distinguishable. The number of test sequences generated by Algorithm 3 is no more than the number of mutants in F'_{PA}. The time needed for minimization is $O(N\log N)$ with online minimization algorithm [13], and the calculation of separating sequence requires $O(N^2)$ where N is the number of states in the reduced machine. We propose an optimization technique for generating separating sequence online in [21], which will reduce the cost of this algorithm for average case but the worst case complexity is the same.

As far as the fault detection capability is concerned, the test suite generated includes a test case to distinguish every mutant that is derived by applying one mutation function in F'_{PA}. Since all test sequences are progressive sequence, from Proposition 1, we have:

Proposition 2. Tests generated from Algorithm 3 detect all mutants under F'_{PA} in time $O(N^2)$ where N is the number of states in the reduced machine.

Algorithm 3 also applies to all other fault models that satisfy proposition 1. Note that the test suite does not discover all faulty mutants in F_{PA}; if a mutation function itself is not distinguishable, then Algorithm 3 simply discards it.

4.3 Experiment

We again conduct the experiment using NSL protocol. In the specification (Fig. 2) a total of 19 Boolean expressions are identified, as shown in Fig. 4. These expressions are used to construct the fault type F_{PA} and the mutants. Among them δ_{b12} and δ_{b7} correspond to the three implementations X and Y in Section 4, respectively. Algorithm 3 produces $F'_{PA} = F_{PA} - \{\delta_{b18}, \delta_{b19}\}$ and the set of 17 test sequences. The last two Boolean expressions are not associated with any I/O behaviors and are not observable. The lengths of those sequences are shown in Table 2 and the details are omitted. All the sequences are short (less than 4). This set of test sequences detect all implementations with one or more Boolean expressions missing.

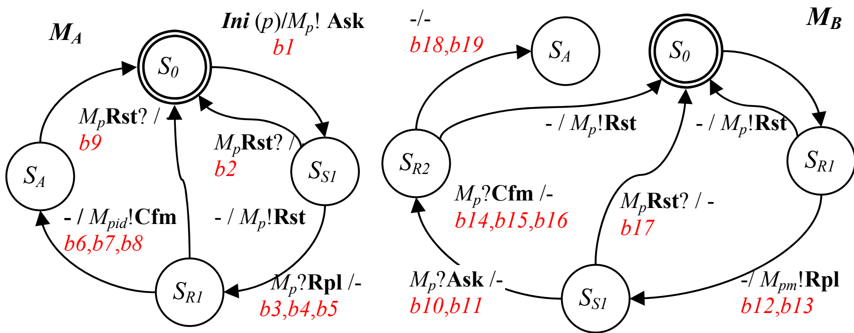
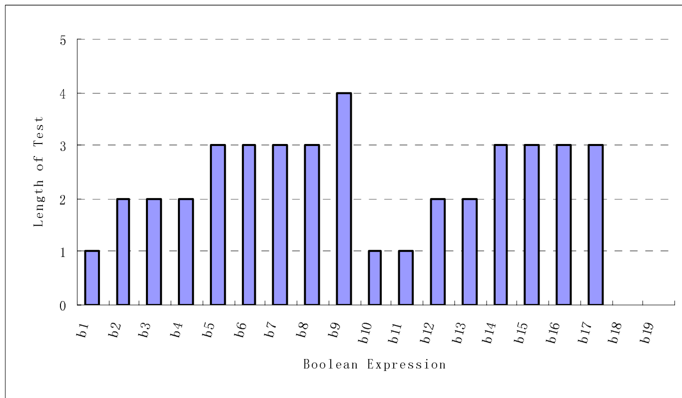


Fig. 4. Boolean Expressions in NSL Specification

Table 2. Test Sequence Lengths Generated by Algorithm 3



5 Conclusion

This paper studies the problem of testing message confidentiality of security protocols. EFSM with symbolic message type is used to model security protocol system with an omnipotent intruder. A formal definition of message confidentiality property and the black box testing model are provided. Passive monitoring, guided random walk and mutation testing approaches are presented with case studies.

A lot of issues remain to be explored, such as efficient modeling for intruder knowledge acquisition for more powerful testing results, thorough and structured active testing procedures, and more general mutation testing with more focus on message confidentiality violation yet with less computation costs. On the other hand, systematic experiments are to be conducted on the de-facto security protocols, such as Kerberos, electronic payment, and IPSec.

References

1. Achilles Proxy. http://www.mavensecurity.com/achilles
2. S. Chen, Z. Kalbarczyk, J. Xu and Ravishankar K. Iyer. A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities. International Conference on Dependable Systems and Networks (DSN'03), page 605, 2003.
3. R. DeMillo, R. Lipton, and F. Sayward. Hints on Test. Data Selection : Help For The Practicing Programmer. IEEE Computer, vol. 1 l(4), pages 34-41, 1978.
4. D. Dolev and A. Yao. On the security of public-key protocols. IEEE Transaction on Information Theory 29, pages 198-208, 1983.
5. A. Duale and M. Ümit Uyar. A Method Enabling Feasible Conformance Test Sequence Generation for EFSM Models. IEEE Trans. Computers 53(5): pages 614-627, 2004.
6. S. Fabbri, J. Maldonado, T. Sugeta, and P. Masiero. Mutation testing applied to validate specifications based on statecharts. In International Symposium on Software Reliability Systems (ISSRE), pages 210-219, 1999.

7. D Geer and J. Harthorne. Penetration Testing: A Duet. In Proceedings of. the 18th Annual Computer Security Applications Conference (ACSAC), pages 185–198, 2002.
8. S. Jaiswal, G. Iannaccone, J. Kurose and D. Towlsey. Formal Analysis of Passive Measurement Inference Techniques. To appear in Proceedings of IEEE Infocom 2006.
9. J. Jurjens and G. Wimmel. Formally Testing Fail-Safety of Electronic Purse Protocols. IEEE International Conference on Automated Software Engineering, page 408, 2001.
10. D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In Proceedings of ICNP, pages 122–131, 2002.
11. D. Lee, K. K. Sabnani, D. M. Kristol and S. Paul. Conformance Testing of Protocols Specified as Communicating Finite State Machines - a Guided Random Walk Based Approach. IEEE Trans. on Communications, Vol. 44, No. 5, pages 631-640, 1996.
12. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In Proceedings of the IEEE, pages 1090–1123, August 1996.
13. D. Lee and M. Yannakakis. Online minimization of transition systems. In Proceedings of STOC, pages 264–274, 1992.
14. G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In Proceedings of TACAS'96, LNCS 1055, 1996.
15. B. Marick. The Weak Mutation Hypothesis. Proceedings of The ACM SIGSOFT Symposium on. Testing, Analysis, and Verification, October, 1991.
16. C. Meadows. Applying formal methods to the analysis of a key management protocol, J. Comput. Security 1, pages 5-53, 1992.
17. C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. IEEE Journal on Selected Areas in Communications, 21(1), pages 44-54, 2003.
18. R. Needham, M. Schroeder. Using encryption for authentication in large networks of computers, Communications of the ACM, 21(12), pages 993-999, 1978.
19. S. Schneider. Security Properties and CSP, Proceedings of the 1996 IEEE Symposium on Security and Privacy, page 174, 1996.
20. O. Sheyner, J. Haines, S. Jha, R. Lippmann and J. Wing. Automated Generation and Analysis of Attack Graphs. IEEE Symposium on Security and Privacy, 2002.
21. G. Shu and D. Lee. Network Protocol System Fingerprinting – A Formal Approach. To appear in Proceedings of IEEE Infocom 2006.
22. H. Thompson. Application Penetration Testing. IEEE Security & Privacy. 3(1), pages 66–69, 2005.
23. H. Thompson. Why Security Testing Is Hard. IEEE Security and Privacy. 1(4), pages 83-86, July-August, 2003.
24. G. Wimmel, J. Jürjens, Specification-Based Test Generation for Security-Critical Systems Using Mutations. Proceedings of ICFEM pages 471-482, 2002.