PAPER

On-the-Fly Trace Generation Approach to the Security Analysis of the TMN Protocol with Homomorphic Property: A Petri Nets-Based Method

Yongyuth PERMPOONTANALARP[†a)], ***Member and*** **Apichai CHANGKHANAK**[†], ***Nonmember***

SUMMARY Many Petri nets-based methods have been developed and applied to analyze cryptographic protocols. Most of them offer the analysis of one attack trace only. Only a few of them provide the analysis of multiple attack traces, but they are rather inefficient. Similarly, the limitation of the analysis of one attack trace occurs in most model checking methods for cryptographic protocols. Recently, we proposed a simple but practical Petri nets-based model checking methodology for the analysis of cryptographic protocols, which offers an efficient analysis of all attack traces. In our previous analysis, we assume that the underlying cryptographic algorithms are black boxes, and attackers cannot learn anything from cipher text if they do not have a correct key. In this paper, we relax this assumption by considering some algebraic properties of the underlying encryption algorithm. Then, we apply our new method to TMN authenticated key exchange protocol as a case study. Surprisingly, we obtain a very efficient analysis when the numbers of attack traces and states are large, and we discover two new attacks which exploit the algebraic properties of the encryption.

key words: *formal methods for cryptographic protocols, model checking, Petri nets*

1. Introduction

Cryptographic protocols are protocols which use cryptographic techniques to achieve certain tasks while preventing malicious parties from attacking the protocols. There are many applications of cryptographic protocols, for example, authenticated key exchange protocols, web security protocols, e-payment protocols, e-banking protocols, e-voting protocols.

The design and analysis of cryptographic protocols are difficult to achieve because of the increasingly sophisticated attacking capabilities and the complex requirement of the applications. Attacks in many cryptographic protocols have been found after they have been designed [1], [2] and even after implemented e.g. [3], [4]. Thus, it is desirable to have a method which is able to analyze *all possible attacks* to the protocols. Such a method would offer a comprehensive understanding of all vulnerabilities of protocols and certainly would help in developing a better and total protection for them. In this paper, we focus on message replay attacks [5] and the analysis of multiple sessions of protocol execution only.

Many Petri nets-based methods have been developed and applied to analyze cryptographic protocols [6]–[12]. Most of them offer the analysis of one attack trace only. Only a few of them provide the analysis of multiple attack traces, but they are rather inefficient. In fact, all of them employ an *off-the-fly* trace generation technique. It means that after a state space is generated either partially or fully and an attack state is found, then an attack trace is computed. This kind of trace generation is called *off-the-fly*, since the trace computation occurs after the state space is generated. The analysis of a single attack trace is rather limited, since one attack trace shows only one way amongst many possible ways to carry out an attack.

Similarly, the limitation of the analysis of one attack trace occurs in most model checking methods for cryptographic protocols [14]–[16], [18]–[22].

Recently, we proposed a very simple but practical Petri nets-based model checking methodology for the analysis of cryptographic protocols, which overcomes this limitation [39]. Our methodology offers an efficient analysis of all attack traces, and is essentially independent of model checking tools for the full and explicit state space analysis. We employ a novel method which is the *on-the-fly* trace generation for computing all attack traces. In the new method, while a state space is generated, attack traces for states are computed at the same time, and are stored at the states. In other words, the trace computation occurs at the same time as the state space computation. This technique provides a notable improvement in the computation time for all attack traces when the number of attack traces and the number of states are large. Then, we applied our new method to two case studies, which are Micali's contract signing protocol [30] and TMN authenticated key exchange protocol [31]. We found many new attacks on the two protocols

However, in our previous analysis, we assume that the underlying cryptographic algorithms are black boxes, and attackers cannot learn anything from a cipher text if they do not have a correct key. In this paper, we relax this assumption by considering some algebraic properties of the underlying encryption algorithm. In particular, we consider the homomorphic property of RSA encryption. Then we apply our new method to TMN authenticated key exchange protocol as a case study. Surprisingly, we found two new attacks which exploit the homomorphic property. In addition, we obtain a very efficient analysis when the numbers of attack traces and states are large. In fact, when the number of states

Manuscript received March 22, 2011.
Manuscript revised July 31, 2011.

[†]The authors are with King Mongkut's University of Technology Thonburi, Bangkok, Thailand.
a) E-mail: yongyuth.per@kmutt.ac.th

DOI: 10.1587/transinf.E95.D.215

and traces are 126,536 and 11,676, respectively, our method improves over the *off-the-fly* trace generation on the computation times by 5,043%. In particular, our *on-the-fly* method reduces the computation times from 25 hours in the *off-the-fly* method to 29 minutes. We implement our methodology in Coloured Petri Nets (CPN) [32] and use its model checker tool called CPNTools [33] to do the experimental analysis. Our preliminary results were reported previously [36]–[38].

Furthermore, the comprehensive analysis of all attack traces has not received adequate attention in the literature. In this paper we would like to point out the importance of this kind of analysis and show its use when analyzing algebraic properties of the underlying encryption algorithm in protocols. Indeed, the importance of the analysis is even clearer when dealing with algebraic properties of encryption, since the analysis shows clearly many different attacking ways to create the same damage, e.g. to expose a session key. Those attacks are significantly different. Thus, our analysis offers a thorough examination of all attacks.

In Sect. 2, we provide the background on TMN protocol and Coloured Petri nets. In Sect. 3, we compare our new method with existing related works, and discuss known attacks on TMN protocol. In Sect. 5, we present our new CPN methodology and apply it to TMN. Furthermore, in Sect. 5 new attacks are explained, the comparison on the performance between our *on-the-fly* method and the *off-the-fly* method is discussed, and an analysis of our method is given.

2. Background

We use the following notations throughout the paper. $S \rightarrow R : M$ means that user S sends message M to user R. While $\{M\}PK\text{-}I$ means public-key encryption by the RSA algorithm on message M by I's public key, $E_K(M)$ means the Vernam cipher or one-time pad on message M by key K. *In* is an attacker. Also, *In(A)* means that the attacker who impersonates user A. While 1) - 4) describe protocol steps in the 1st session, 1') - 4') indicate protocol steps in the 2nd session.

2.1 TMN Authenticated Key Exchange Protocol [31]

TMN is a cryptographic key exchange protocol for mobile communication system. TMN allows initiator A to exchange a session key with responder B by the help of server J. The detail of TMN is described as follows.

1) $A \rightarrow J : (B, \{K_{aj}\}PK\text{-}J), A$
2) $J \rightarrow B : A$
3) $B \rightarrow J : (A, \{K_{ab}\}PK\text{-}J), B$
4) $J \rightarrow A : B, E_{Kaj}(K_{ab})$

where K_{ab} is an exchanged session key and K_{aj} is A's secret which is used to transport the session key at the last step. Note that the session key is created by user B.

It is well-known that RSA public key encryption satisfies the following homomorphic property [40].

$\{M1\}PK \times \{M2\}PK = \{M1 \times M2\}PK$

where $\times$ means the multiplication modulo the public modulus. Note that throughout the paper, $\times$ means the multiplication modulo the public modulus. Therefore, a thorough analysis on TMN must deal with the homomorphic property of RSA public key encryption too.

2.2 Coloured Petri Nets [32]

CPN is a graph-based modeling language which is equipped with model checking algorithms and tools. CPN has been applied to many applications, for example, distributed systems and communication protocols. In CPN, a system is first modeled by a kind of graphs, called a net, and then a state space of all possible executions of the system is generated and analyzed to search for errors in the system. CPN provides a software tool called CPNTools [33] which facilitates the creation, the modification and the analysis of nets. Originally, CPN and CPNTools provide the *off-the-fly* trace generation method. Also, only a single error trace or attack trace can be detected by the built-in mechanism in CPNTools.

3. Related Works

3.1 Existing Petri Nets Methods for Cryptographic Protocols

Many Petri nets-based methods, which are equipped with model checking algorithms, have been developed and applied to analyze cryptographic protocols [6]–[12]. All of them [6], [8]–[12] with one exception [7] aim at analyzing only one attack trace. The exception [7] offers the analysis of multiple attack traces, but it is inefficient. In fact, all of them employ an *off-the-fly* trace generation technique which means that an attack trace or multiple attack traces are computed after a state space is generated either partially or fully. The *off-the-fly* trace generation for all attack traces involves the searching for all paths between two states in a state space, which is extremely time-consuming. Indeed, the searching for all paths can be seen as a core part of algorithms for solving the traveling salesman problem which is known to be NP-complete.

Nieh and Tavares's method [6] is the first work which applies PN to analyze security protocols. Their approach is based on CPN and provides a generic model of an attacker for message replay attack in a single session of protocol execution. Some attacks are discovered in some protocols. Later on, Lee et. al.'s works [7], [8] developed place/transition nets which are low-level nets to analyze security protocols. Their method [7] was applied to analyze multiple attack traces for the message replay attack. Also, TMN was chosen as a case study, but only known attacks on TMN were reconstructed. Another low-level PN was developed to analyze a key recovery protocol [8]. But the analysis is on the recoverability of a key rather than the message replay attack. Also, both works [7], [8] deal with the analysis of a single session only.

Al-Azzoni et. al. [9] have developed a CPN model to

analyze TMN protocol. Their method appears to analyze two sessions of protocol execution. However, it does not really analyze two concurrent sessions but two sequential sessions according to the detected attack reported in the work. Their CPN model is well-structured. But attacker capabilities in their model are quite limited. Indeed, our CPN model extends their work to analyze multiple concurrent sessions of protocol execution and to analyze more attacking capabilities, namely, a new session initiation by the attacker, receiver impersonation and message dropping.

Dresp [10] applied CPN to analyze a security protocol. The work strictly analyzes a single session, but can be used to analyze two sequential sessions by executing one session at a time.

Bouroulet and Devillers [11] developed a specification language and a verification for security protocols. A security protocol and attackers are specified by a language based on process algebra. For verification, the specification is translated into a high-level PN. Then, a general purpose high-level PN model checker called Helena is applied. Their method offers the analysis of multiple concurrent sessions. However, their method can analyze only one attack trace.

Liu et. al. [12] applied two PN methodologies to analyze an authenticated key exchange protocol in wireless network. In particular, they employ CPN to reproduce a known attack in the protocol by the simulation analysis and employ PEP [13] to detect the known attack by the state space analysis. In their state space analysis, only one known attack trace is reproduced and a single session is analyzed.

Indeed, PEP [13] indigenously computes one attack trace only. However, it can be used to compute multiple attack traces. The verification in PEP, called the unfolding, is considered as the *off-the-fly* trace generation method, since the search for attack traces is performed after a state space is computed partially. This amounts to the search of all paths in a graph.

3.2 Other Model Checking Methods for Cryptographic Protocols

Many model checking methods [14]–[25] have been developed and applied to analyze cryptographic protocols. All of them except for NRL [23], SATMC [17], Proverif [24], [25] and Scyther [27], [28] offer the analysis of one attack trace only. For the exceptions, while NRL is inefficient, SATMC and Proverif can compute a restricted kind of attack traces only, not all attack traces. Scyther can compute a greater number of attack traces than all others. We will discuss more about Scyther below.

In this paper, we focus on the discussion of methods which currently analyze multiple attack traces, but we will also discuss one well-known existing method which analyzes a single trace. A more detailed discussion on each method can be found in another work of our group [39].

SPIN [29] is a widely used explicit model checker which provides the analysis of a single trace. SPIN is employed to analyze a cryptographic protocol [19]. A known attack to a protocol is detected. SPIN is based on the *on-the-fly verification* which offers an advantage in that only a partial state space which is relevant to a verification property is constructed instead of the full state space. During the construction, a single attack trace is computed and is stored in a stack. SPIN can thus be considered as the *on-the-fly trace generation* but for *a single attack trace* only.

SATMC [17] which stands for SAT-based Model Checker employs the satisfiability approach for model checking. One of the interesting features of SATMC is that it can compute a single partial order attack. A partial order attack represents some specific kinds of multiple attack traces where some parts of the traces are partially ordered and other parts are totally ordered. The partially ordered parts of the traces are traces which can be interleaved without any effect, are grouped into a set and are interpreted by parallel execution. However, a single partial order attack represents only a specific structure of attack traces but not all possible attack traces, since all possible attack traces cannot always be organized into a single total order of sets of interleaving attack traces.

Both NRL [23] and Proverif [24], [25] analyze multiple attack traces and they are based on the logic programming approach. NRL can compute all attack traces, but its approach is inefficient, since it uses the path-searching in an entire state space. Thus, it is based on the *off-the-fly* trace generation. On the other hand, Proverif can compute only a restricted set of attack traces which contains only one trace in most cases, according to their experiment and analysis. Note that this restricted set of attack traces is far from the set of all attack traces.

Recently, Maude-NPA [26], a successor of NRL, has been developed. It appears that Maude-NPA records an attack trace into each state during the backward computation of a state space. However, the main purpose of Maude-NPA for recording an attack trace is to allow users to debug a protocol during the computation rather than to compute all attack traces as this is evident by the following sentence quoted from the paper [26]:

Note that two extra state components (the message sequence, or attack trace, and some auxiliary data) are associated to a Maude-NPA state ..., but they are irrelevant and useful only for user debugging of the protocol, ...

In addition, there is neither discussion nor analysis of the advantageous point of recording an attack trace to a state for computing all attack traces in Maude-NPA [26]. Our approach has been developed independently of their method, and we provide a comprehensive analysis on the computation of all attack traces by two different methods as discussed in Sect. 5.6.

Scyther [27] is a high performance model checking tool that is capable of analyzing the unbounded verification. Currently, Scyther does not offer the analysis of algebraic properties of encryption. Scyther provides the analysis of all attack traces. But the analyses of all attack traces in Scyther and in our method are different. There are attacks found by Scyther but not found by our method, and there are at-

tacks found by our method but not found by Scyther too. Scyther computes a state space for a specified total number of instances of user roles, but with many possibilities of the assignment of the number of instances to each user role within the same state space. For example, with the total of 4 instances of users, Scyther analyzes many possibilities, for example, 2 instances of initiator and 2 instances of server, 2 instances of responder and 2 instances of server, and 1 instance of initiator and 1 instance of responder and 2 instances of server. However, our method computes a state space for a fixed number of instances for each fixed user role. Thus, the attacks found by Scyther but not found by our method are those which occur in the other assignments than the fixed one in our method. But we argue that all the attacks found by Scyther can be found in our method by computing a state space with each possible assignment or configuration as which will be discussed in Sect. 5.2.

For TMN with black-box encryption and with 1 instance of initiator, 1 instance of responder and 2 instances of server, we found 10 attacks that allow the attacker to learn the session key, but Scyther found 3 attacks only, and these can also be found by our method. In addition, for the attacks that allow the attacker to learn the session key and fool the initiator to accept K_{aj} as a session key, we found 10 attacks, but Scyther found none. Note that the security analysis of TMN in Scyther reported here was done by running the tool on TMN supplied with the tool [28] as an example. The details of the attacks are given in the appendix.

3.3 Known Multiple-Session Attacks on TMN

In the literature, there are two kinds of known multi-session attacks on TMN. The first kind does not exploit the homomorphic property whereas the second kind does. In the first kind, there are four known attacks [20], [22], [35]. In the first attack found by Lowe and Roscoe [20], the attacker learns the session key K_{ab} and A's secret key K_{aj}. Thus, the attacker can then learn all the subsequent communications between the users. The second attack [22] is a variant of the first one. In the third and the fourth attacks found by us [37], user A is fooled to commit on a fake session key known by the attacker, and the attacker learns the valid session key K_{ab}. Thus the attacker can impersonate B to A by using the fake key and can impersonate A to B by using the correct session key.

In the second kind of known multiple-session attacks on TMN, there is one known attack found by Simmons [31]. In the attack, the attacker modifies message at step 1 by using the homomorphic property. We show the first attack in the following.

1) $A \rightarrow J : (B, \{K_{aj}\}PK\text{-}J), A$
2) $J \rightarrow B : A$
3) $B \rightarrow J : (A, \{K_{ab}\}PK\text{-}J), B$
4) $J \rightarrow A : B, E_{Kaj}(K_{ab})$
1') $In \rightarrow J : (B, \{K_{aj} \times K_c\}PK\text{-}J), In$
2') $J \rightarrow In(B) : In$
3') $In(B) \rightarrow J : (In, \{K_i\}PK\text{-}J), B$
4') $J \rightarrow In : B, E_{(Kaj \times Kc)}(K_i)$

where K_i and K_c are attacker's secret keys.

At step 1') of the second session, the attacker computes the cipher text $\{K_{aj} \times K_c\}PK\text{-}J$ from the cipher text $\{K_{aj}\}PK\text{-}J$ and $\{K_c\}PK\text{-}J$ by using the homomorphic property. At the completion of the two sessions, the attacker In learns $(K_{aj} \times K_c)$, and then obtains K_{aj} by computing the multiplicative inverse of K_c. Also, by the cipher text at step (4), the attacker obtains K_{ab}.

Lowe and Roscoe [20] argued that this attack is useful when J is able to detect the reuse of the cipher text, encrypted by J's public key, at steps 1 and 3 in the multiple concurrent sessions.

4. A Modified Version of TMN

To illustrate our new attacks by using the homomorphic property clearly, we will consider a modified version of TMN instead of the original TMN. The modified TMN is similar to the TMN, but J has the ability to detect the reuse of the cipher texts at steps 1 and 3 in multiple concurrent sessions. In addition, user A has the ability to check if the exchanged session key is identical to A's secret K_{aj}. So, if the session key is identical to K_{aj}, then A will abort the session. So, the modified TMN is more secure than the original TMN. Also, the analysis of the modified TMN would demonstrate the importance of attacks using the homomorphic property.

5. Our Model

5.1 Our General Methodology

In the following, we present our general methodology which is independent of model checking tools for the full and explicit state space analysis. Our methodology consists of five steps which are (1) protocol and attacker representation, (2) computations of a decomposed state space and multiple attack traces, (3) characterization of and search for attack states, (4) attack trace extraction and (5) attack trace classification. Our *on-the-fly* trace generation is employed in steps 2 and 4.

(1) Both a protocol and an attacker models are represented. In fact, such representation depends on a model checker tool.

(2) Then, a state space is generated from the representation. During the state space generation, when a state is generated, an attack trace to the state is computed at the same time and the computed trace is stored at the state. This computation is the core of the *on-the-fly* trace generation. Figure 1 illustrates the *on-the-fly* trace generation process in general. In the figure, a_1, a_2 and a_3 stand for message sending by users or attackers. In state s_3, the path or trace $<a_1,a_2>$ to the state is stored in the state. Conceptually, an attack trace for a state is constructed by simply extending an attack trace stored in the previous state.

For simplicity, we assume that each state stores only

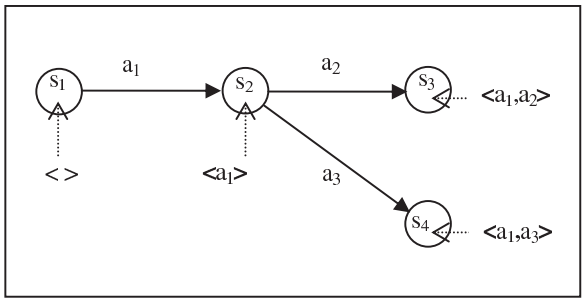

Fig. 1 The on-the-fly trace generation.

one attack trace. Thus, our state space in general may contain a greater number of states than the state space in the *off-the-fly* trace generation. A state which can be reached by two different attack traces in the *off-the-fly* method becomes two different states in our method. To reduce the size of a computed state space, we employ a *decomposition* technique in the state space computation. Our decomposition technique aims to generate a state space for one specific attack scenario at a time.

An attack scenario is defined by a configuration which consists of the information for the protocol execution in a multi-session setting, for example, the identities of initiator and responder, the role of attackers, secrets and nounces in each concurrent session, and a schedule of a specific interleaving execution of the multiple concurrent sessions. Thus, a decomposed state space contains one interleaving execution of multiple sessions, instead of all. However, we can explore each attack scenario one by one by computing a state space for each possible configuration. Note that it can be seen that our configuration contains a fixed number of instances for each fixed user role in each session.

(3) After the state space is obtained, attack states for each kind of attacks are specified and searched in the state space. An attack is characterized by a vulnerability event which is an event potentially leading to a compromise of protocols. Vulnerability events are protocol-dependent.

(4) When an attack state is found in the state space, an attack trace is extracted from the state immediately. By searching for all attack states of the same attack, all attack traces of the same attack can be obtained without any path searching. In other words, the computation for all attack traces is reduced to the searching for attack states which can be done very efficiently and linearly on the number of states.

(5) The number of attack traces obtained can be quite large. We propose *textual* trace analysis technique to classify such a large number of attack traces. Those attack traces are classified by using attack patterns which are minimal but necessary protocol traces for an attack. The development of an attack pattern is manual, because it is protocol-dependent. But the attack classification is automatic. Attack traces that contain the same attack pattern are classified into the same group of attack traces. As a result, a large number of attack traces is reduced to a reasonable number of attack patterns which are easier to analyze.

5.2 Our Method for the Modified TMN

In this section, we discuss the assumptions of our protocol analysis. We also describe vulnerability events of TMN, and a configuration of the protocol execution.

Definition 1 (Assumptions of the protocol execution): The following are the assumptions of the execution of the TMN protocol.

1. There is one attacker (*In*). The attacker abilities are based on Dolev and Yao's attacker assumption [41], but our attacker can modify any public-key cipher text by using the homomorphic property. Such a modified cipher text contains at most the multiplication of two keys as its plain text. Also, the attacker is allowed to create cipher texts by single encryption only, not by multiple and nested encryption. More generally, the attacker can create only messages with a bounded length.
2. For RSA public key encryption, we consider two properties: the homomorphic property and the commutative of the multiplication.
3. We consider the execution of two different concurrent sessions of the protocol where such execution can be performed in an interleaving and non-sequential manner between multiple sessions.
4. A well-behaved initiator and a well-behaved responder are involved in one session only, but they may or may not take part in the same session. While the server is involved in two sessions, the attacker may involve in up to two sessions.
5. The attacker has one secret (symmetric) key K_i.

In 3, we do not consider two identical sessions, since the message replay between them does not produce any new information but duplicates one and thus it is not useful for an attack.

In 4, a well-behaved initiator and a well-behaved responder are involved only in one session amongst the concurrent two sessions, because we are interested mainly in analyzing a *man-in-the-middle* attack where an attacker impersonates the initiator and the responder in the remaining session. Even though we assume that the server is involved in the two sessions, our attacker can impersonate the server in one session, two sessions or none.

Definition 2 (The finite analysis of all attack traces): Our analysis of all attack traces is *finite* in that it is performed in a *finite* setting where

- The length of messages that the attacker can create is bounded.
- The number of concurrent sessions of protocols that the attacker involves is bounded.

Therefore, our method produces a *finite* set of all attack traces, because we consider the finite setting of attack analysis. However, in general the set of all attack traces can be infinite due to the unbounded length of messages and the

unbounded number of sessions.

Attack states are characterized by vulnerability events. For the TMN protocol, there are two basic vulnerability events which are secret disclosure by an attacker and session key commitment by initiator and responder. Based on the two basic events, the following combined vulnerability events can be created.

Definition 3 (Vulnerability events): There are three combined vulnerability events for TMN.

1. The attacker learns K_{ab} and K_{aj}, and both A and B commit on K_{ab}. $[K_{ab},K_{aj}][K_{ab}][K_{ab}]$
2. The attacker learns K_{ab} and K_{aj}, and A is fooled to commit on K_i, but B commits on K_{ab}. $[K_{ab},K_{aj}][K_i][K_{ab}]$
3. The attacker learns K_{ab} and K_{aj}, and A is fooled to commit on MK, but B commits on K_{ab} where MK is a multiplicative key. $[K_{ab},K_{aj}][MK][K_{ab}]$

We use the notation $[KB_1][KB_2][KB_3]$ to describe each combined vulnerability event where KB_1 stands for keys that are known by the attacker, and KB_2 and KB_3 stands for keys that are committed by users A and B, respectively, at the completion of the protocol.

In event 1, the attacker learns all later communication between A and B, because the attacker knows the session key between A and B. In fact, events 2 and 3 can be seen as a kind of *man-in-the-middle* attack in that the attacker can impersonate B to A by using key K_i or MK, respectively, while the attacker can impersonate A to B by using key K_{ab}.

Definition 4 (Configuration): A configuration of a decomposed state space computation consists of $((S_1, S_2,\ldots,S_n), Sch)$ and $S_i = (s,I,R,T,K)$ for $1 \le i \le n$ where n is the number of concurrent sessions, and

1. S_i is a session information for the i-th session which consists of
   a. s is a session identity.
   b. I, R and T are identities for an initiator, a responder and a server, respectively.
   c. K is a list of keys for each party (including attacker) which consist of a pair of public and private keys, and a shared key with a specific party
2. Sch is a multi-session schedule which means a specific interleaving execution of multiple concurrent sessions of protocol runs

We consider the following four configurations according to our assumptions. Note that K and Sch are omitted for simplicity.

1) (1,A,B,J) & (2,In,In,J)
2) (1,A,In,J) & (2,In,B,J)
3) (1,In,B,J) & (2,A,In,J)
4) (1,In,In,J) & (2,A,B,J)

Configuration *(1,A,In,J) & (2,In,B,J)* means that A and B perform as initiator and responder in the first and second session, respectively. The attacker *In* impersonates initiator and responder in the second and first session, respectively. In all configurations, we consider the multi-session schedule for the *man-in-the-middle* attack [42].

5.3 Our CPN Model for the Modified TMN

Our CPN model extends Al-Azzoni et. al.'s CPN model [9] to provide the analysis of all attack traces and the analysis of multiple concurrent sessions. We refer to the work [9] for the background on the basic structure of protocol and attacker representation. In the following, we discuss some main parts of our CPN model only.

Our CPN model consists of three main levels: top, entity and control. The top level shows the interaction between all parties including the attacker, and the entity level shows the detailed behaviour of each party. The control level controls the model execution according to an input configuration. Figure 2 shows the top level. There are 4 entities in our model which are A, B, J and attacker In. Exchanged messages at all protocol steps pass through the attacker In. Since there are four steps in TMN, there are four corresponding paths between relevant parties in the model. For example, path <*A, P1, In, P2, J*> corresponds to protocol step 1.

We implement the *on-the-fly* trace generation in CPN by recording incrementally into a state each message sent by users and attackers. In CPN, a state is represented by tokens at all places. So, to record a trace into a state, we simply store all sent messages as tokens into a global place, i.e. place *ETrace*.

Figure 3 shows the entity level of user A for the first step of the protocol. Transition *T1* is to compose a message and transition *T3* is to send the message to a communication channel. The global place *ETrace*, which is adjacent to transition *T3*, stores a tuple (*lk1*, *lk2*, *lk3*, *tr*) where *lk1*, *lk2* and *lk3* stand for *[KB1]*, *[KB2]* and *[KB3]*, respectively. Also, *tr* stands for an attack trace which is represented by a list of tuples (*s, step, sid, rid, msg*) where *s* is a session identity, *step* is a protocol step number, *sid* is sender identity, *rid* is receiver identity and *msg* is the sent message. When a new message is sent, *tr* is appended with a new tuple for the new message. Similarly, in other users' model and the attacker model, a transition which sends a message to a communication channel would record an attack trace *tr* with appropriate

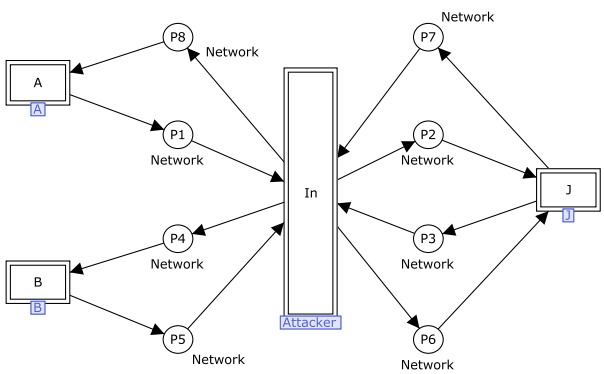

Fig. 2 Top level.

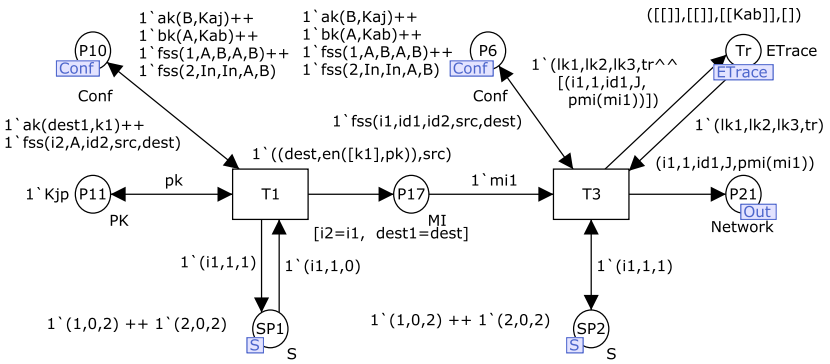

Fig. 3 User A.

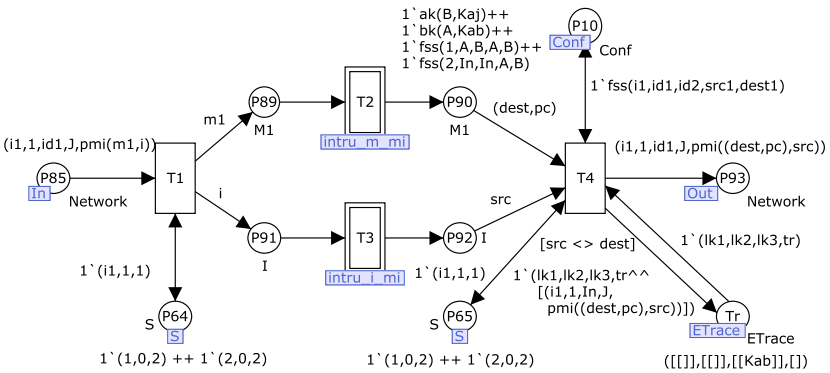

Fig. 4 Attacker for step 1.

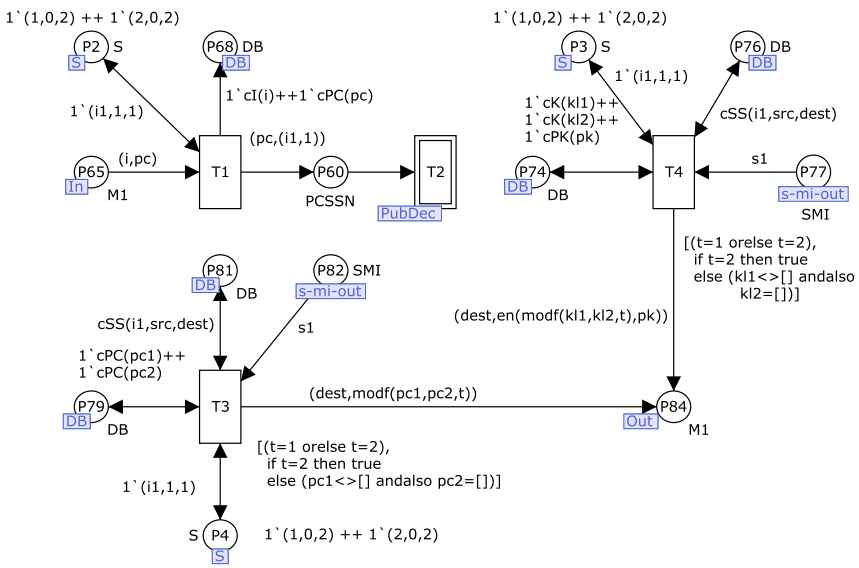

Fig. 5 Sub-attacker for step 1.

KB1, *KB2* and *KB3* into place *ETrace*.

For the attacker model, we create a CPN graph for each protocol step. The attacker graph aims to intercept a message at each step and to send a modified message instead. Figure 4 shows a CPN graph for step 1. Transition *T1* is to intercept the message at step 1 from *A*, and transition *T4* is to send a modified message to *J*. Since the message can be divided into two parts which are initiator identity and a pair of responder identity and a cipher text, the attacker graph for step 1 employs two attacker sub-graphs which are *intru_i_mi* and *intru_m_mi*, respectively. Figure 5 shows the graph for *intru_m_mi*. In the figure, transition *T1* is to decompose the pair of messages, and transition *T2* is to decrypt the cipher text. Note that the detail of transition *T2*

is omitted here. Transitions *T3* and *T4* shows two ways to compose a modified message at step 1 by using the homomorphic property. While *T3* takes two existing cipher texts from attacker's database and multiplies them, *T4* takes two plain texts and a key from attacker's database, multiplies the plain texts and encrypts the result by the key.

The homomorphic property of RSA is modeled in our CPN method by using the list data structure and some user-defined functions. Both plain text and cipher text are represented by lists *[M]* and *[(M,P)]*, respectively, where M is the original plain text in the protocol, P is a public key, and then *[(M,P)]* is a cipher text of message M by key P. The multiplication modulo public modulus is represented by the list concatenation (^^). The encryption function is defined as follows.

$E_P([M_1,\ldots,M_n]) = [(M_1,P),\ldots,(M_n,P)]$ where $n \geq 1$.

When n=1, we obtain a cipher text $[(M,P)]$. When n>1, we obtain the list $[(M_1,P),\ldots,(M_n,P)]$ which represents the multiplication of cipher texts.

The homomorphic property discussed in 2.1 is obtained in our method as follows.

$E_P([M1])\hat{}\hat{}E_P([M2]) = E_P([M1]\hat{}\hat{}[M2])$

It should be noted that the commutative of the multiplication holds due to the commutative of elements in the list.

It is assumed that plain text M cannot be extracted directly from cipher text $[(M,P)]$. But M can be obtained from the cipher text if the corresponding private key is given to a decryption function.

In our model shown in Fig. 5, we use two user-defined functions *modf* and *en* which represent the multiplication and encryption function E, respectively. Also, t stands for the number of terms taken for the multiplication.

To specify a configuration, we use four tokens: *fss*, *ak*, *bk* and a multi-session schedule token. Tokens *fss*, *ak* and *bk* are stored in place *Conf* in Figs. 3 and 4, but the multi-session schedule is stored in place *Sche* in Fig. 6. Token *fss(s,i1,i2,i3,i4)* contains information about actual initiator identity *i1*, actual responder identity *i2*, impersonated initiator identity *i3* and impersonated responder identity *i4* in session *s*. Tokens *ak(i1,k1)* and *bk(i2,k2)* contain initiator's secret key *k1* for responder *i1* and responder's session key *k2* for initiator *i2*, respectively. The multi-session schedule is represented by list of session identities $[s_1, s_2, \ldots, s_n]$ which stands for a specific interleaving execution of a protocol step between multiple sessions in the order specified by the list.

To compute a decomposed state space by a configuration, we employ the control level. The control level shown in Fig. 6 schedules the interleaving execution of multiple sessions according to the input schedule. When session *s* is scheduled for execution of a single protocol step, the control graph would inform all other graphs by creating in place *S* session state token *(s,sp,st)*, where *st* is a session status and *sp* is the current protocol step number to be executed, with ready status. The execution of a protocol step involves three graph models which represent the sender, the attacker and

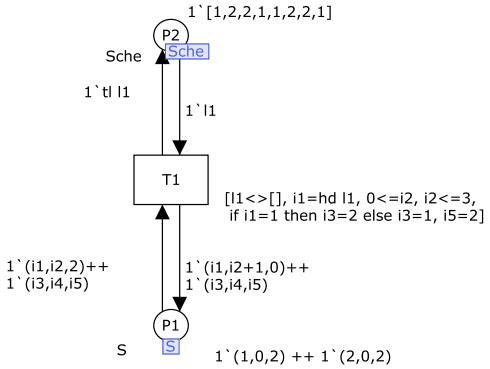

Fig. 6 Control level.

the receiver of the step. After the receiver has stored the received message from the sender into her database, the session state token is set to inactive status. Then, the control is passed back to the control graph to perform further scheduling.

5.4 Searching for Attack States in a State Space

The state space computation is done by using state space tools provided by CPNTools. After a decomposed state space is obtained, we search for attack states in the state space by running CPN-ML programs which extract specified nodes from the state space. In fact, our programs compute attack states for each vulnerability event.

The following program finds states where the attacker knows some key. In other words, it finds states for *[KB1]*.

```
val LeafNodes=ListDeadMarkings();
fun SecrecyViolation1(k:LK) :
Node list
= PredNodes(LeafNodes,
    fn n => (cf(cK(k), Mark.SymDec'P3 1 n) > 0),
    NoLimit);
```

SecrecyViolation1(K_{ab}) produces a set of all *terminal* states in the state space where the key K_{ab} is in the attacker's database. The terminal states are states which do not have any further computation, and they mean states at the end of protocol execution. The end of protocol execution means either the normal termination after the last protocol step or an abnormal termination where the latter is caused by the attacker who may drop or tamper with messages during transmission. It is sufficient to consider terminal states instead of all states, since the attacker's database on keys is never decreased and likewise for user A's database on an exchanged session key. Therefore, there is no loss of data at terminal states. Also, in many cases we obtain full details of attack traces at terminal states.

Function *(Mark.SymDec'P3 1 n)* returns a multi-set of tokens at place *P3* in page *SymDec* in state n, and they mean data in attacker's database. *cK* is a constructor used to indicate a key in attacker's database. Function *cf(c,ms)* produces the number of appearance of token colour c in multi-set *ms*. Also, function *PredNodes(a,p,l)* searches area a with search

limit l and produces a list of all nodes that satisfy boolean function p. Further details about functions used in this program can be found at CPNTools [33].

There is a similar program to the above program, but it computes states where user A accepts some key as a session key at the last protocol step. By taking the intersection between the results from the two programs, we obtain attack states for our vulnerability events. Note that there is no need to compute states where user B accepts a key as a session key, since B accepts only her own generated key which is K_{ab}.

Due to reasons of space, we omit the details on the extraction of all attack traces from states and the attack classification.

5.5 New Attacks in TMN

We argue that the analysis of all attack traces is important in that it helps us to discover all possible ways to attack protocols. In the modified TMN, the analysis shows all possible ways that the attacker can modify messages using the homomorphic property to evade the detection of the replay attack by the server and compromise the protocol. We found two new kinds of attacks. While Simmons's attack exploits the homomorphic property at step 1, our two new attacks exploit the homomorphic property at step 3 and at steps 1 and 3, respectively.

In the first kind of our new attacks, there are two categories. The first category is the attack at step 3 in one session while the second category is the attack at step 3 in two sessions. Below, attacks 1.1.1 - 1.1.3 and 1.2.1 - 1.2.3 are in the first and second categories, respectively, of the first kind of attacks. Also, in the following, we describe all attacks in two multiple sessions of protocols where, in the first session, A communicates with B, but in the second session, the attacker impersonates both A and B to server J.

There are three vulnerability events for attacks in the first category of the first kind. In the following, we show an attack for vulnerability event 2.

Attack 1.1.1
1) $A \rightarrow J : (B, \{K_{aj}\}PK\text{-}J), A$
1') $In(A) \rightarrow J : (B, \{K_i\}PK\text{-}J), A$
2') $J \rightarrow In(B) : A$
2) $J \rightarrow B : A$
3) $B \rightarrow In(J) : (A, \{K_{ab}\}PK\text{-}J), B$
$In(J) \rightarrow J : (A, \{K_i\}PK\text{-}J), B$
3') $In(B) \rightarrow J : (A, \{K_{ab}\times K_i\}PK\text{-}J), B$
4') $J \rightarrow In(A) : B, E_{Ki}(K_{ab}\times K_i)$
4) $J \rightarrow A : B, E_{Kaj}(K_i)$

In step 3), the cipher text $\{K_{ab}\}PK\text{-}J$ that is sent by B is intercepted by attacker, and it is replaced by cipher text $\{K_i\}PK\text{-}J$ which is delivered to J. Since $\{K_{aj}\}PK\text{-}J$ and $\{K_i\}PK\text{-}J$ are delivered to J at steps (1) and (3), respectively, then cipher text $E_{Kaj}(K_i)$ is created and sent to A. As a result, A is fooled to commit on a fake session key K_i. It can be seen that the attack exploits the homomorphic property at step 3') of the second session.

In the following, we show an attack for vulnerability event 1 for the first category of the first kind.

Attack 1.1.2
1) $A \rightarrow In(J) : (B, \{K_{aj}\}PK\text{-}J), A$
$In(J) \rightarrow J : (B, \{K_i\}PK\text{-}J), A$
1') $In(A) \rightarrow J : (B, \{K_{aj}\}PK\text{-}J), A$
2') $J \rightarrow In(B) : A$
2) $J \rightarrow B : A$
3) $B \rightarrow In(J) : (A, \{K_{ab}\}PK\text{-}J), B$
$In(J) \rightarrow J : (A, \{K_{aj}\}PK\text{-}J), B$
3') $In(B) \rightarrow J : (A, \{K_{ab}\times K_{aj}\}PK\text{-}J), B$
4') $J \rightarrow In(A) : B, E_{Kaj}(K_{ab}\times K_{aj})$
4) $J \rightarrow In(A) : B, E_{Ki}(K_{aj})$
$In(A) \rightarrow A : B, E_{Kaj}(K_{ab})$

This attack also demonstrates another way to modify message at step 3') by using the homomorphic property. After steps (4) and (4'), the attacker learns all secrets K_{aj} and K_{ab}. Then, the attacker can create cipher text $E_{Kaj}(K_{ab})$ which is sent to A at step (4). So, A commits on the correct session key.

In the following, we show an attack for vulnerability event 3 for the first category of the first kind of attacks. In this attack, the attacker fools A to commit on a multiplicative key (MK). There are many possible multiplicative keys and in the following, we show the third attack for multiplicative key $(K_{aj}\times K_{ab})$.

Attack 1.1.3
1) $A \rightarrow J : (B, \{K_{aj}\}PK\text{-}J), A$
1') $In(A) \rightarrow J : (B, \{K_i\}PK\text{-}J), A$
2') $J \rightarrow In(B) : A$
2) $J \rightarrow B : A$
3) $B \rightarrow J : (A, \{K_{ab}\}PK\text{-}J), B$
3') $In(B) \rightarrow J : (A, \{K_{aj}\times K_i\}PK\text{-}J), B$
4') $J \rightarrow In(A) : B, E_{Ki}(K_{aj}\times K_i)$
4) $J \rightarrow In(A) : B, E_{Kaj}(K_{ab})$
$In(A) \rightarrow A : B, E_{Kaj}(K_{aj}\times K_{ab})$

This attack demonstrates another way to modify message at step 3') by using the homomorphic property. After steps (4) and (4'), the attacker learns all secrets K_{aj} and K_{ab}. Then, the attacker can create cipher text $E_{Kaj}(K_{aj}\times K_{ab})$ which is sent to A at step (4). So, A commits on a fake session key. This attack is different from attack 1.1.1 in that the attacker does not have to disclose his own secret K_i to A.

For the second category of the first kind of attacks, there are also three vulnerability events. In the following, we discuss an attack for vulnerability event 2.

Attack 1.2.1
1) $A \rightarrow J : (B, \{K_{aj}\}PK\text{-}J), A$
1') $In(A) \rightarrow J : (B, \{K_i\}PK\text{-}J), A$
2') $J \rightarrow In(B) : A$
2) $J \rightarrow B : A$
3) $B \rightarrow In(J) : (A, \{K_{ab}\}PK\text{-}J), B$
$In(J) \rightarrow J : (A, \{K_{ab}\times K_i\}PK\text{-}J), B$
3') $In(B) \rightarrow J : (A, \{K_{aj}\times K_i\}PK\text{-}J), B$
4') $J \rightarrow In(A) : B, E_{Ki}(K_{aj}\times K_i)$
4) $J \rightarrow In(A) : B, E_{Kaj}(K_{ab}\times K_i)$
$In(A) \rightarrow A : B, E_{Kaj}(K_i)$

It is important to note that the attacker modifies messages at step 3 in two sessions by using the homomorphic property. We also find a variant of Simmons's attack where the homomorphic modification occurs at step 1) in two sessions.

We show an attack for vulnerability event 1 of the second category of the first kind in the following. All steps in the attack are identical to attack 1.2.1, except for the following step 4.

Attack 1.2.2

4) J → In(A) : B, $E_{Kaj}(K_{ab}\times K_i)$
In(A) → A : B, $E_{Kaj}(K_{ab})$

In the following, we show an attack for vulnerability event 3 in the second category. All steps in the attack are identical to attack 1.2.1, except for the following step 4.

Attack 1.2.3

4) J → In(A) : B, $E_{Kaj}(K_{ab}\times K_i)$
In(A) → A : B, $E_{Kaj}(K_{aj}\times K_{ab})$

There are many possible multiplicative keys that the attacker can fool *A* to commit. But in this attack, we show for multiplicative key $(K_{aj}\times K_{ab})$.

Next, we discuss attacks in the second kind which exploits the homomorphic property at steps 1 and 3. Also, there are two categories of the second kind of attacks. The first category is the attack at steps (1) and (3) in the same session while the second is the attack at steps (1) and (3) in two different sessions. However, both categories lead to the three vulnerability events. Below, attacks 2.1.1 - 2.1.3 and 2.2.1 - 2.2.4 are in the first and second categories, respectively, of the second kind of attacks.

The following shows an attack for vulnerability event 2 of the first category.

Attack 2.1.1

1) A → J : (B, $\{K_{aj}\}$PK-J), A
1') In(A) → J : (B, $\{K_{aj}\times K_i\}$PK-J), A
2') J → In(B) : A
2) J → B : A
3) B → In(J) : (A, $\{K_{ab}\}$PK-J), B
In(J) → J : (A, $\{K_i\}$PK-J), B
3') In(B) → J : (A, $\{K_{ab}\times K_{aj}\}$PK-J), B
4') J → In(A) : B, $E_{(Kaj\times Ki)}(K_{ab}\times K_{aj})$
4) J → A : B, $E_{Kaj}(K_i)$

It should be noted that the attacker modifies messages at steps (1') and (3') in the second session by using the homomorphic property. Since $\{K_{aj}\}PK\text{-}J$ and $\{K_i\}PK\text{-}J$ are delivered to *J* at steps (1) and (3), respectively, *A* is fooled to commit on fake session key K_i by cipher text $E_{Kaj}(K_i)$ at step (4).

The following attack leads to vulnerability event 1 of the first category.

Attack 2.1.2

1) A → In(J) : (B, $\{K_{aj}\}$PK-J), A
In(J) → J : (B, $\{K_i\}$PK-J), A
1') In(A) → J : (B, $\{K_{aj}\times K_i\}$PK-J), A
2') J → In(B) : A
2) J → B : A
3) B → In(J) : (A, $\{K_{ab}\}$PK-J), B
In(J) → J : (A, $\{K_{aj}\}$PK-J), B
3') In(B) → J : (A, $\{K_{ab}\times K_i\}$PK-J), B
4') J → In(A) : B, $E_{(Kaj\times Ki)}(K_{ab}\times K_i)$
4) J → In(A) : B, $E_{Ki}(K_{aj})$
In(A) → A : B, $E_{Kaj}(K_{ab})$

This attack also demonstrates another way to modify messages at steps (1') and (3') by using the homomorphic property. After steps (4) and (4'), the attacker learns all secrets K_{aj} and K_{ab}. Then, the attacker can create cipher text $E_{Kaj}(K_{ab})$ which is sent to *A* at step (4). Then *A* commits on the correct session key.

In the following, we show an attack for vulnerability event 3 in the second category. All steps in the attack are identical to attack 2.1.2, except for the following step 4.

Attack 2.1.3

4) J → In(A) : B, $E_{Ki}(K_{aj})$
In(A) → A : B, $E_{Kaj}(K_{aj}\times K_{ab})$

For the second category of the second kind of attacks, there are also three vulnerability events. In the following, we discuss an attack for vulnerability event 2.

Attack 2.2.1

1) A → In(J) : (B, $\{K_{aj}\}$PK-J), A
In(J) → J : (B, $\{K_i\}$PK-J), A
1') In(A) → J : (B, $\{K_{aj}\times K_i\}$PK-J), A
2') J → In(B) : A
2) J → B : A
3) B → In(J) : (A, $\{K_{ab}\}$PK-J), B
In(J) → J : (A, $\{K_{aj}\times K_i\}$PK-J), B
3') In(B) → J : (A, $\{K_{ab}\}$PK-J), B
4') J → In(A) : B, $E_{(Kaj\times Ki)}(K_{ab})$
4) J → In(A) : B, $E_{Ki}(K_{aj}\times K_i)$
In(A) → A : B, $E_{Kaj}(K_i)$

It should be noted that the attacker modifies the message at steps (1') in the second session and message at step (3) in the first session by using the homomorphic property.

In the following, we show an attack for vulnerability event 1.

Attack 2.2.2

1) A → In(J) : (B, $\{K_{aj}\}$PK-J), A
In(J) → J : (B, $\{K_{aj}\times K_i\}$PK-J), A
1') In(A) → J : (B, $\{K_i\}$PK-J), A
2') J → In(B) : A
2) J → B : A
3) B → J : (A, $\{K_{ab}\}$PK-J), B
3') In(B) → J : (A, $\{K_{ab}\times K_i\}$PK-J), B
4') J → In(A) : B, $E_{Ki}(K_{ab}\times K_i)$
4) J → In(A) : B, $E_{(Kaj\times Ki)}(K_{ab})$
In(A) → A : B, $E_{Kaj}(K_{ab})$

This attack also demonstrates another way to modify messages at steps (1) and (3) in two different sessions by using the homomorphic property. In this attack, the message at step (1) is modified in the first session, and the message at step (3') is modified in the second session.

In the following, we show an attack for vulnerability event 3.

Attack 2.2.3

1) A → J : (B, $\{K_{aj}\}$PK-J), A

1') In(A) → J : (B, {K_{aj}×K_i}PK-J), A
2') J → In(B) : A
2) J → B : A
3) B → In(J) : (A, {K_{ab}}PK-J), B
In(J) → J : (A, {K_{aj}×K_{ab}}PK-J), B
3') In(B) → J : (A, {K_i}PK-J), B
4') J → In(A) : B, $E_{(Kaj×Ki)}(K_i)$
4) J → In(A) : B, $E_{Kaj}(K_{aj} \times K_{ab})$

Since $\{K_{aj}\}PK\text{-}J$ and $\{K_{aj}{\times}K_{ab}\}PK\text{-}J$ are delivered to *J* at step 1 and 3, respectively, then *A* commit to multiplicative key $K_{aj}{\times}K_{ab}$ as a session key by cipher text $E_{Kaj}(K_{aj}{\times}K_{ab})$ at step 4.

There is another interesting variant of attack 2.2.1 where the attacker modifies message at step (1) in a session and also modifies messages at step (3) in two sessions. This variant belongs to the second category of the second attack since the message modification at steps (1) and (3) occurs in two different sessions.

Attack 2.2.4

1) A → In(J) : (B, {K_{aj}}PK-J), A
In(J) → J : (B, {K_{aj}×K_i}PK-J), A
1') In(A) → J : (B, {K_i}PK-J), A
2') J → In(B) : A
2) J → B : A
3) B → In(J) : (A, {K_{ab}}PK-J), B
In(J) → J : (A, {K_{aj}×K_{ab}}PK-J), B
3') In(B) → J : (A, {K_{aj}×K_i}PK-J), B
4') J → In(A) : B, $E_{Ki}(K_{aj}{\times}K_i)$
4) J → In(A) : B, $E_{(Kaj×Ki)}(K_{aj}{\times}K_{ab})$
In(A) → A : B, $E_{Kaj}(K_i)$

It can be seen that the attacker modifies the message at step (1) in the first session, but modifies messages at step (3) in two sessions.

In fact, there are many variants of attacks discussed above, but those variants are slightly different from the attacks shown. We omit them for reasons of space.

5.6 Performance

In this section, we compare the results between our *on-the-fly* and the *off-the-fly* trace generation methods both of which are implemented in CPNTools model checker. The experiment is done by using CPNTools version 3.0 and a PC with Intel Core2 Duo 2.33 Ghz and 2 GB of RAM.

In Table 1, we show the comparison of the sizes of the state spaces between the two methods for the four configurations. The four configurations are discussed in Sect. 5.2. In the worst case, the number of states and arcs in the *on-the-fly* method are increased for 39% and 33%, respectively.

Table 1 The comparison of the sizes of state spaces.

Config	On-the-fly		Off-the-fly		Inc %	
	nodes	arcs	nodes	arcs	n	a
1)	175,691	187,986	126,536	141,171	39	33
2)	95,513	102,990	70,319	78,807	36	31
3)	47,979	51,188	40,830	44,840	18	14
4)	24,755	26,905	21,197	23,707	17	13

However, in the best case the number of states and arcs are increased for only 17% and 13%, respectively. In the table, *n* and *a* mean the increment percentage (*Inc%*) of nodes and arcs, respectively, in the *on-the-fly* method when comparing with the *off-the-fly* method.

In Table 2, we compare the computation times for state spaces between the two methods for the two configurations which are for the largest and the smallest number of states. Also, we compare the computation times for attack traces (*tr*) and for both state spaces and traces (*total*) in Tables 3 and 4. In Tables 3 and 4, events (*ev*) 1 and 2 correspond to vulnerability events 1 and 2, respectively. But events (*ev*) 4, 5 and 6 are events other than vulnerability events 1 and 2 where the attacker learns K_{ab}, K_{aj} and both K_{ab} and K_{aj}, respectively. Note that vulnerability event 3 is included in event 6. Also, in Tables 3 and 4, *Imp%* stands for the improvement or reduced percentage of the total computation times in the *on-the-fly* method when comparing with the *off-the-fly* method.

It is clear that our *on-the-fly* method improves the overall computation times tremendously. When the numbers of states and traces are large, for example in event 5 of Table 3, it takes about 29 minutes (1,749 sec.) in our method, but about 25 hours (89,960 sec.) in the *off-the-fly* method. It should be noted that the *off-the-fly* trace generation method requires a large amount of time, because it searches all paths between an initial state and each attack state. As discussed in Sect. 3.1, the search of all paths can be considered as a main part of the solutions to the traveling salesman problem which is known to be NP-complete. When the numbers of states and traces are small, for example in event 1 of Table 3, it takes about 37 seconds in our method, but about 49

Table 2 The comparison of the computation times in seconds for state spaces for two configurations.

Config	On-the-fly	Off-the-fly
1) *(1,A,B,J)&(2,In,In,J)*	1,737	655
4) *(1,In,In,J)&(2,A,B,J)*	37	21

Table 3 The comparison of the total computation times in seconds for configuration *(1,A,B,J)&(2,In,In,J)*.

Ev	Attack Traces	On-the-fly		Off-the-fly		Imp %
		tr	total	tr	total	
1	1,068	1	1,738	8,541	9,196	429
2	1,068	1	1,738	8,860	9,515	447
4	8,460	7	1,744	65,499	66,154	3,693
5	11,676	12	1,749	89,305	89,960	5,043
6	7,668	6	1,743	59,670	60,325	3,360

Table 4 The comparison of the total computation times in seconds for configuration *(1,In,In,J)&(2,A,B,J)*.

Ev	Attack Traces	On-the-fly		Off-the-fly		Imp %
		tr	total	tr	total	
1	18	0	37	28	49	32
2	0	-	-	-	-	-
4	342	1	38	392	413	986
5	1,056	1	38	1,380	1,401	3,586
6	342	1	38	394	415	992

seconds in the *off-the-fly* method.

Indeed, our *on-the-fly* method requires more times for state space computation, but the *off-the-fly* method requires much more times for trace generation. It should be noticed that in Tables 3 and 4, when the number of traces is increased, the time for trace generation in the *off-the-fly* method grows greatly, but the time for trace generation in our method is almost constant. Even though our *on-the-fly* method requires a larger state space than the *off-the-fly* method, our method offers a much faster trace computation. In particular, the size of a state space in our method is increased at most 39%, but the total computation times in our method are reduced, at best, 5,043%.

5.7 Analysis

As discussed in Sect. 5.1, our *on-the-fly* trace generation method requires a greater number of states than the *off-the-fly* method, when there are states that can be reached by different attack traces in the *off-the-fly* method. Those states are called *merging* states. Recall that each state in our method stores only one attack trace. Thus, a merging state reachable by multiple attack traces is divided into multiple states in our method. In fact, the more merging states there are in the *off-the-fly* method, the larger number of states the *on-the-fly* method has. From here on, merging states refer to the *off-the-fly* method, but the division of the states refers to the *on-the-fly* method.

Merging states are caused by the removal of different information in previous states, which results in identical states. For example, if two states are different on some data and a transition which occurs at the states removes the data, then two states are merged into the same next state. On the other hand, if the information in states is added with various possibilities after some transitions, then those states are split into multiple next states according to the possibilities. In general, a state contains two main kinds of information: messages exchanged between all entities and data stored at each entity's database where an entity means a user or an attacker. Since sent messages are usually stored at receivers' databases, the information in a state can be characterized mainly by data at all entities' databases. Therefore, the removal of such data that cause the difference between states is responsible for merging states. We assume below that any removal of the data will result in merging states.

Data at attacker's database are never removed, since the attacker always keeps those data for attacking protocols subsequently. However, data at users' databases may be removed, since those users want to minimize their resources as soon as they finish their tasks in the protocol. For example, in a general authenticated key exchange protocol, while a server does not keep any data in a session after the completion of its tasks in the session, initiators and responders keep exchanged session keys and related session information within their databases for further communications between them.

If the information in users' databases is removed at an earlier protocol step, then the *on-the-fly* method requires a larger number of states. This is because the removal at an earlier step results in merging states nearer to the top of the state space. Then, the division of the merging states requires a large number of additional states, because states below the merging states need to be copied in order to store different attack traces. Thus, the higher position the merging state is, the larger number of additional states the *on-the-fly* method requires. However, the division of merging states near to the bottom of the state space does not require many additional states, since there are not many states at the bottom.

In general, the protocol step in which data in users' databases are removed can be determined by the last step that a server is involved in a protocol. If the last step is near to the end of a protocol, then the number of additional states required in the *on-the-fly* method is small due to the occurrence of merging states near to the bottom of a state space. Otherwise, the number of additional states required can be very large.

In TMN, initiator A and responder B keep their data after the end of session execution, but server J removes session data after J sends the message at step 4. During the session execution, server maintains session information *(src, dest, k1, k2, s)* where *src* is an initiator identity, *dest* is a responder identity, *k1* is initiator's secret sent at step 1 and *k2* is a session key sent by the responder at step 3 and *s* is a session identity. Before the session information is removed from J's database, the message at step 4, which contains *dest* and a cipher text generated by *k1* and *k2*, is sent to and stored at A. Thus, only some session information such as *src* and *s* is removed from users' databases. Note that the removed session information causes the difference between states. Since the removal occurs at the last step, the number of additional states required is not too much.

We have examined protocols in the Clark/Jacob library [1] to discuss the generality of our method. We found that there is no removal of the data in 23 protocols, but there is some removal in 18 protocols. In the latter, the removal takes place at either the last step or the next-to-the last step for 8 protocols, and the removal occurs between the first step and step $n/2$, where n is the total number of steps in each protocol, for 4 protocols. Thus, in 31 out of 41 protocols the number of additional states required in the *on-the-fly* method should be acceptable. Also, it is in only 4 protocols that the number of additional states in our method would be very high.

6. Discussion

We do not claim that our tool is the fastest. But we would like to point out the advantage of the *on-the-fly* trace generation over the *off-the-fly* trace generation in terms of the computation times. Note that the *off-the-fly* trace generation can be considered as a conventional method to compute all attack traces. In addition, we believe that many existing model checking methods can be extended to deal with the analysis of all attack traces. But, currently, most of them

provide only the analysis of a single attack trace. So it is our objective in this paper to point out the importance of the analysis of all attack traces and to show the practical use of this method as discussed above.

As a future work, we aim to optimize the memory requirement for the state space in the *on-the-fly* trace generation. Also, we aim to explore other algebraic properties of encryption algorithms in other protocols.

7. Conclusion

The comprehensive analysis of all attack traces for cryptographic protocols has not received adequate attention in the literature. In this paper, therefore, we would like to emphasize the importance of the analysis and show its use when analyzing algebraic properties of the underlying encryption algorithm in protocols. In particular, in this paper, we extend our previous method to analyze some algebraic properties of the underlying encryption algorithm in cryptographic protocols. Then we apply it to TMN protocol. As a result, we found two new attacks. The result also shows that our method improves over the conventional method with respect to the total computation times, by 5,043% in the best case.

Acknowledgements

The first author would like to thank Kurt Jensen and his group for the suggestion about the names of the *on-the-fly* trace generation and the *textual* trace analysis after his presentation at CPN'09 workshop. Also, we would like to thank anonymous reviewers for their helpful comments. Finally, the first author would like to thank Somsak Vanit-Anunchai for motivating discussion, and he would like to acknowledge financial support from the Thailand Research Fund and the National Research Council of Thailand.

References

[1] J. Clark and J. Jacob, "A survey on authentication protocol literature: version 1.0," http://www-users.cs.york.ac.uk/~jac/PublishedPapers/reviewV1_1997.pdf, accessed July 17 2011.

[2] C. Meadows, "Formal methods for cryptographic protocol analysis: Emerging issues and trends," IEEE J. Sel. Areas Commun., vol.21, no.1, pp.44–54, Jan. 2003.

[3] U. Meyer and S. Wetzel, "A man-in-the-middle attack on UMTS," Proc. 3rd ACM workshop on Wireless security, pp.90–97, Philadelphia, USA, Oct. 2004.

[4] I. Cervesato, A.D. Jaggard, A. Scedrov, J. Tsay, and C. Walstad, "Breaking and fixing public-key kerberos," Inf. Comput., vol.206, no.2-4, pp.402–424, Feb. 2008.

[5] P. Syverson, "A taxonomy of replay attacks," Proc. 7th IEEE Computer Security Foundations Workshop, pp.187–191, New Hampshire, USA, June 1994.

[6] B. Nieh and S. Tavares, "Modelling and analyzing cryptographic protocols using petri nets," Proc. Workshop on the Theory and Application of Crypt. Tech., pp.275–295, Queensland, Australia, LNCS Springer Verlag, Dec. 1992.

[7] G. Lee and J. Lee, "Petri net based models for specification and analysis of cryptographic protocols," J. Syst. Softw., vol.37, no.2, pp.141–159, May 1997.

[8] S. Lim, J. Ko, E. Jun, and G. Lee, "Specification and analysis of n-way key recovery system by extended cryptographic timed petri net," J. Syst. Softw., vol.58, no.2, pp.93–106, Sept. 2001.

[9] I. Al-Azzoni, D.G. Down, and R. Khedri, "Modeling and verification of cryptographic protocols using coloured Petri nets and design/CPN," Nordic J. Comput., vol.12, no.3, pp.200–228, June 2005.

[10] W. Dresp, "Security analysis of the secure authentication protocol by means of coloured petri nets," Proc. 9th IFIP Commun. and Multimedia Security, pp.230–239, Salzburg, Austria, LNCS Springer Verlag, Sept. 2005.

[11] R. Bouroulet, R. Devillers, H. Klaudel, E. Pelz, and F. Pommereau, "Modeling and analysis of security protocols using role based specifications and petri nets," Proc. 29th Int. Conf. on App. and Theory of Petri Nets, pp.72–91, Xi'an, China, LNCS Springer Verlag, June 2008.

[12] J. Liu, X. Ye, J. Zhang, and J. Li, "Security verification of 802.11i 4-way handshake protocol," Proc. 2008 IEEE Inter. Conf. on Commun., pp.1642–1647, Beijing, China, May 2008.

[13] B. Grahlmann and E. Best, "PEP - More than a petri net tool," Proc. 2nd Inter. Workshop on Tools and Algorithms for Construction and Analysis of Syst., pp.397–401, Passau, Germany, LNCS Springer verlag, March 1996.

[14] The AVISPA project, http://avispa-project.org, accessed July 17 2011.

[15] M. Turuani, "The CL-Atse protocol analyser," Proc. 17th Inter. Conf. on Term Rewriting and App., pp.277–286, Seattle, USA, LNCS Springer Verlag, Aug. 2006.

[16] D. Basin, S. Mödersheim, and L. Viganò, "OFMC: A symbolic model checker for security protocols," Int. J. Information Security, vol.4, no.3, pp.181–208, June 2005.

[17] A. Armando and L. Compagna, "SAT-based model checking for security protocols analysis," Int. J. Information Security, vol.7, no.1, pp.3–32, Jan. 2008.

[18] Y. Boichut, P.-C. Héam, and O. Kouchnarenko, "Automatic verification of security protocols using approximations," INRIA Research Report, 2005.

[19] P. Maggi and R. Sisto, "Using SPIN to verify security properties of cryptographic protocols," Proc. 9th Inter. SPIN Workshop, pp.85–87, Grenoble, France, LNCS Springer Verlag, April 2002.

[20] G. Lowe and B. Roscoe, "Using CSP to detect errors in the TMN protocol," IEEE Trans. Softw. Eng., vol.23, no.10, pp.659–669, Oct. 1997.

[21] G. Lowe, "Breaking and fixing the needham-schroeder public-key protocol using FDR," Software - Concepts and Tools, vol.17, no.3, pp.93–102, 1996.

[22] J. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murϕ," Proc. 1997 IEEE Symp. on Security and Privacy, pp.141–151, California, USA, May 1997.

[23] C. Meadows, "The NRL protocol analyzer: an overview," J. Logic Programming, vol.26, no.2, pp.113–131, Feb. 1996.

[24] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," Proc. 14th IEEE Comp. Security Foundations Workshop, pp.82–96, Nova Scotia, Canada, June 2001.

[25] X. Allamigeon and B. Blanchet, "Reconstruction of attacks against cryptographic protocols," Proc. 18th IEEE Comp. Security Foundations Workshop, pp.140–154, Aix-en-Provence, France, June 2005.

[26] S. Escobar, C. Meadows, and J. Meseguer, "Maude-NPA: Cryptographic protocol analysis modulo equational properties," Foundations of Security Analysis and Design V FOSAD 2007/2008/2009 Tutorial Lectures, pp.1–50, Springer-Verlag, 2009.

[27] C. Cremers, "Unbounded verification, falsification, and characterization of security protocols by pattern refinement," Proc. 15th ACM Conf. on Comp. and Commun. Security, pp.119–128, Virginia, USA, Oct. 2008.

[28] Scyther tool, http://people.inf.ethz.ch/cremersc/scyther, accessed July 17 2011.

[29] G. Holzmann, The SPIN Model Checker: Primer and Reference

Manual, Addison-Wesley, 2003.
[30] S. Micali, "Simple and fast optimistic protocols for fair electronics exchange," Proc. 21st Symp. on Prin. of Distributed Computing, pp.12–19, Massachusetts, USA, July 2003.
[31] M. Tatebayashi, N. Matsuzaki, and D. Newman, "Key distribution protocol for digital mobile communication systems," Proc. Advances in Cryptology - CRYPTO' 89, pp.324–334, California, USA, LNCS Springer Verlag, Aug. 1990.
[32] K. Jensen, Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts, Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, 2nd ed., 1997.
[33] CPNTools, http://cpntools.org, accessed July 17 2011.
[34] R. Kemmerer, C. Meadows, and J. Millen, "Three systems for cryptographic protocol analysis," J. Cryptology, vol.7, no.2, pp.79–130, June 1994.
[35] Y. Zhang and X. Liu, "An approach to the formal analysis of TMN protocol," Progress on Cryptography: 25 years of Cryptography in China, ed. K. Chen, pp.235–243, The Kluwer Inter. Series in Eng. and Comp. Science, 2004.
[36] Y. Permpoontanalarp and P. Sornkhom, "A new coloured petri net methodology for the security analysis of cryptographic protocols," Proc. 10th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, pp.81–100, Arhus, Denmark, Oct. 2009.
[37] Y. Permpoontanalarp, "Security analysis of the TMN protocol by using coloured petri nets: multi-session case," Proc. 10th Inter. Conf. on Intelligent Tech., pp.401–410, Guilin, China, Dec. 2009.
[38] Y. Permpoontanalarp, "On-the-fly trace generation and textual trace analysis and their applications to the analysis of cryptographic protocols," Proc. 30th Formal Tech. for Networked and Distributed Syst., pp.201–215, Amsterdam, The Netherlands, LNCS Springer Verlag, June 2010.
[39] Y. Permpoontanalarp and P. Sornkom, "On-the-fly trace generation and textual trace analysis and their applications to the analysis of cryptographic protocols: A petri nets-based method," submitted for a journal, Dec. 2010.
[40] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," in Foundations of Secure Computation, ed. R. DeMillo and et. al., pp.169–180, Academic Press, New York, 1978.
[41] D. Dolev and A. Yao, "On the security of public key protocols," IEEE Trans. on Inf. Theory, vol.29, no.2, pp.198–207, March 1983.
[42] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," Inf. Process. Lett., vol.56, no.3, pp.131–133, Nov. 1995.

Appendix: Detailed Comparison between Our Method and Scyther

In the following, we discuss the comparison between our method and Scyther for analyzing TMN with black box encryption. Also, we deal with the situation of the man-in-the-middle attack [42] where there is 1 instance of initiator, 1 instance of responder and 2 instances of server. In Scyther, we chose 4 runs and computed for all attacks in the search pruning option. We found 10 attacks for vulnerability event 1, but Scyther found 3 attacks only. The 3 attacks can be found by our method by using two configurations: one with the schedule for the man-in-the-middle attack [37] and the other with the schedule for two sequential sessions. The following is one of the attacks that is not found by Scyther, but found by our method.

1) A → In(J) : (B, {K_{aj}}PK-J), A
   In(J) → J : (B, {K_i}PK-J), A
1') In(A) → J : (B, {K_{aj}}PK-J), A
2') J → In(B) : A
2) J → In(B) : A
   In(B) → B : A
3) B → J : (A, {K_{ab}}PK-J), B
3') In(B) → J : (A, {K_i}PK-J), B
4') J → In(A) : B, $E_{Kaj}(K_i)$
4) J → In(A) : B, $E_{Ki}(K_{ab})$
   In(A) → A : B, $E_{Kaj}(K_{ab})$

where K_i is attacker's secret key.

In the first session, K_i and K_{ab} are encrypted at steps 1 and 3, respectively, and in the second session, K_{aj} and K_i are encrypted at steps 1 and 3, respectively. This can be represented by *(<K_i,K_{ab}>,<K_{aj},K_i>)* where the notation *(<k1,k3>,<k1',k3'>)* means that *k1* and *k3* are keys encrypted by *J*'s public key at steps 1 and 3, respectively, in the 1st session, and *k1'* and *k3'* are those keys in the 2nd session. Indeed, the 10 attacks found by us are all possible attacks in the setting of the man-in-the-middle attack and the bounded execution. The 3 attacks found by Scyther can be represented in general by *(<K_{aj},K_{ab}>,<K_i,K_{aj}>)*, *(<K_{aj},K_{ab}>,<K_i,K_{ab}>)* and *(<K_i,K_{aj}>,<K_{aj},K_{ab}>)* where the first two attacks occur in the schedule for the man-in-the-middle attack, and the last attack occurs in the schedule for two sequential sessions.

Also, we found 10 attacks for vulnerability event 2, but Scyther found only 1 attack. In addition, we found 10 attacks for the vulnerability event described by *[K_{aj},K_{ab}][K_{aj}][K_{ab}]* where the attacker fools *A* to commit on fake key K_{aj}. But Scyther found none. The following shows one of the attacks.

1) A → In(J) : (B, {K_{aj}}PK-J), A
   In(J) → J : (B, {K_i}PK-J), A
1') In(A) → J : (B, {K_i}PK-J), A
2') J → In(B) : A
2) J → In(B) : A
   In(B) → B : A
3) B → J : (A, {K_{ab}}PK-J), B
3') In(B) → J : (A, {K_{aj}}PK-J), B
4') J → In(A) : B, $E_{Ki}(K_{aj})$
4) J → In(A) : B, $E_{Ki}(K_{ab})$
   In(A) → A : B, $E_{Kaj}(K_{aj})$

The detail of our attacks for TMN with black box encryption can be found in [37].

Yongyuth Permpoontanalarp received Ph.D. in Computer Science from Imperial College London, UK., in 1998. Currently, he is an associate professor and head of Logic and Security laboratory in Department of Computer Engineering at King Mongkut's University of Technology Thonburi, Bangkok, Thailand.

Apichai Changkhanak received B.Eng. in Electrical Engineering from Chiang Mai University, Thailand in 2002. Currently, he is a student for Master degree at the department of Computer Engineering at King Mongkut's University of Technology Thonburi, Bangkok, Thailand.