An Abstract Model for Security Protocol Analysis

GENGE BELA
IOSIF IGNAT
Computer Science Department
Technical University of Cluj-Napoca
28, Gh. Baritiu St., 400027, Cluj-Napoca
ROMANIA
bgenge@upm.ro, Iosif.Ignat@cs.utcluj.ro, http://users.utcluj.ro/~ignat

Abstract – Security protocol analysis techniques are today mature enough to verify multiple properties of security protocols (correctness, secrecy, authentication). However, because security protocols are growing each day and more complex situations arise (like multi-protocol environments), we consider that the analysis process should be simplified by creating an abstract model for security protocols that captures the essence of the property that is checked. Therefore, in this paper we propose such an abstract model, based on message component types (session keys, nonces, participants), that captures the structure of security protocol messages, thus simplifying the analysis against attacks based on message similarities, that we call "structural attacks". The mentioned attacks are formalized using the proposed typed framework, which is validated by modeling and analyzing the Neuman-Stubblebine authentication protocol.

Key-Words: Security protocols, formal analysis, message typing, replay attacks.

1 Introduction

Security protocols, by definition, are protocols that use cryptographic primitives, which allow the involved parties to exchange some secret information (i.e. shared key or secret data). These protocols have been intensively analyzed in the last few decades, mainly because with the expansion of the internet, existing security holes can now be exploited by malicious users.

From the resulting protocol verification techniques, we identify three major approaches: model checking, theorem proving and static analysis.

Model checking [1] deals with the analysis of an input model viewed as a finite set of states. The analysis starts from an initial state and, by applying state transition rules, it generates the whole state space. In every step, a desired property is checked until a state which violates that property is found. Because it can be fully automated, it requires little or no human interaction. A major drawback of this method is the *state space explosion* problem, as the number of states can become very large even when dealing with simple protocols.

The second approach used in the process of protocol verification is *theorem proving* [3, 4]. Rather than dealing with a set of states, as in the previous method, this approach manipulates logical formulae. Thus, a protocol is viewed as a set of clauses predicating on participant capabilities and a set of inference rules for capturing the message flow between participants. Although *theorem proving* may deal with an unlimited number of protocol instances and with arbitrary complex data, the fact that it requires (expert) human intervention (to formulate inference rules, lemmas, theorems) makes it hard to use for the inexperienced user.

Static analysis has been traditionally used in software analysis to create program abstractions for a simplified analysis. In the area of security protocols, *static analysis* has only been recently applied in [5, 6]. Here, participants are represented as processes and security properties are embodied in participant specification by means of annotations, having the form of typed messages, which restrict the data flow through these points. A major drawback of this approach is that it may find false errors because of the abstraction model that eliminates much of the actual data flow.

The presented approaches have one major aspect in common: they all require an initial protocol specification. The above-presented methods are today mature enough to verify multiple properties of security protocols (correctness, secrecy, authentication).

However, because protocols are growing each day and more complex situations arise, like multi-protocol environments [7, 8], new abstract specifications must be created to simplify the analysis. Thus, if a specification captures the essence of a certain attack, existing tools (that allow the integration of the model, like *Maude* [16, 17]) may find these attacks much faster.

Therefore, we propose a *typed specification* model for *structural analysis* of security protocols that simplifies the analysis process undertaken by existing methods [1, 3, 5], by creating an abstract, simplified model of a security protocol.

A protocol specification is based on the specification of participants that communicate by exchanging messages. A *typed specification* defines a protocol message as the set of its component types (i. e. participant types, key types). This results in an abstract modeling of security protocols by transforming the actual values from the regular specification to their corresponding types.

Because the resulting model captures the structure of a security protocol, an analysis based on this model is called a *structural analysis*. Thus, by simply running a syntactical comparison of the component messages, we may find attacks resulting from message similarities that we call *structural attacks*. We identify two such attacks: *replay* attacks (re-use of generated messages in different contexts than the ones intended in the specification) and *type confusion* attacks (where a message type, for example a session key, may not be verified because of limited participant knowledge).

The paper is structured as follows. In section 2 we define the protocol attacks that we consider having a structural source. We continue with section 3 where we model security protocols and our typed security protocol concepts using first order predicate logic [18]. In section 4 we formalize *structural attacks* and validate our framework by modeling and analyzing the key exchange part from the Neuman-Stubblebine [12] authentication protocol. We compare our work to related ones in section 5 and we finish with a conclusion and future work in section 6.

2 Structural security protocol attacks

In this section we present the main security protocol attacks we consider to have a message structural source: *replay* and *type confusion* (also known as *type flaw*) attacks [10, 11].

Because security protocol specifications may not be so familiar to the reader, in this section we use only a minimal notation, a more extended one being given in section 3. Thus, encryption will be denoted by the "{}*Kab*" construct, where *Kab* is the key. Also, protocol participants will be referred to as *roles*, denoted by capital letters (A, B, S).

2.1 The Neuman-Stubblebine security protocol

To exemplify a new protocol analysis technique, it is always good practice to use an already known protocol. This is why we have chosen the Neuman-Stubblebine [12] (N-S) authentication protocol, which has benefited from an exhaustive analysis over the years [19, 20].

In the next sections, we only provide a limited analysis of the key distribution part of the N-S protocol, which should be enough to give the reader a general idea about the severity of replay and type-flaw attacks.

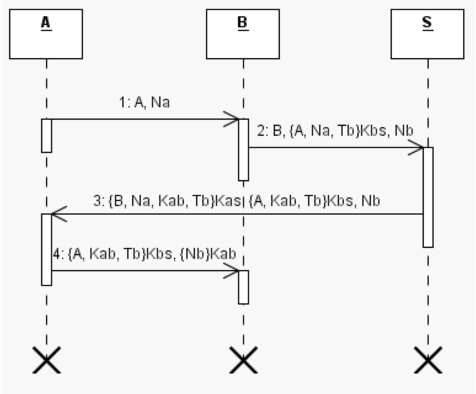

Figure 1. Key distribution part of the Neuman-Stubblebine authentication protocol

The N-S protocol (Figure 1) is initiated by participant A by sending to participant B a message containing his name (*A*) and a nonce (i.e. "number once used") *Na*. Receiving this message, B generates a new message that is sent to server S, containing his name (*B*), a message term encrypted with the shared key *Kbs* and a nonce *Nb*.

By receiving a valid message from B (i.e. message 2), the server knows that A wants to establish a session key with B, which he generates in the form of *Kab*. Then, he encrypts the session key in two different messages using *Kas* (the key that A shares with the server) and *Kbs* (the key that B shares with S) respectively. The messages are then sent to participant A along with the nonce generated by B.

On receiving a valid message from the server (i.e. message 3), participant A, knowing his shared key (*Kas*), decrypts the message part meant for him, thus determining the session key. Then, he uses this key to encrypt the *Nb* nonce, which is sent to B along with the message part encrypted using *Kbs*. This final message informs B about the session key and ensures him that the key was generated by the third-party server and that another party (possibly A) is in the possession of the same key.

2.2 Replay attacks

The generalized definition of a replay attack is: *the use of protocol generated messages in other contexts than the ones intended (provided in the specification), thereby fooling the honest participant(s) into accepting the messages as valid ones.*

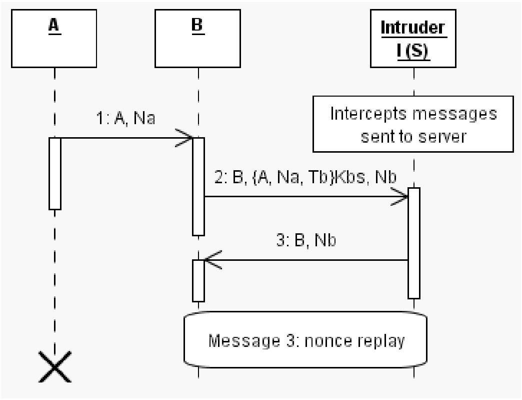

Figure 2. Replay attack on the Neuman-Stubblebine authentication protocol

The replay attack presented in Figure 2 involves two participants, A and B, and an intruder I(S) that intercepts and re-routes the messages sent to server S. An attack based on this configuration consists of resending the two message terms *B* and *Nb* to role B, thus fooling him that a session with himself must be generated. Although this looks like a harmless replay attack, if the effort on B to generate a new session (which may involve multiple server interaction, database communication) is high, then it may lead to a Denial of Service attack.

In the presented example, the intruder uses B's capabilities to generate nonces and valid messages (by simply *replaying* them), therefore B is said to act as an *oracle* [11] (because he always provides the correct answer).

2.3 Type flaw attacks

We say that a *type flaw* (i.e. *type confusion*) attack possibility arises when a component of a received message may be interpreted another way than it was intended by the source that created the message. This may happen because, for example, session keys generated by a third party server cannot be checked for validity by the receiving role. In this context, we identify two kinds of type flaws: *basic type flaws* and *all type flaws*. Thus, if a basic message component (which can not be further decomposed) is interpreted as another basic component, then we are dealing with a *basic type flaw*. If a message component (which has multiple basic components) is interpreted as a basic message component, then have an *all type flaw*.

The security protocol model presented in section 3 is only able to capture *basic type flaws* because *all type flaws* requires term reduction techniques that are considered to be part of a future work.

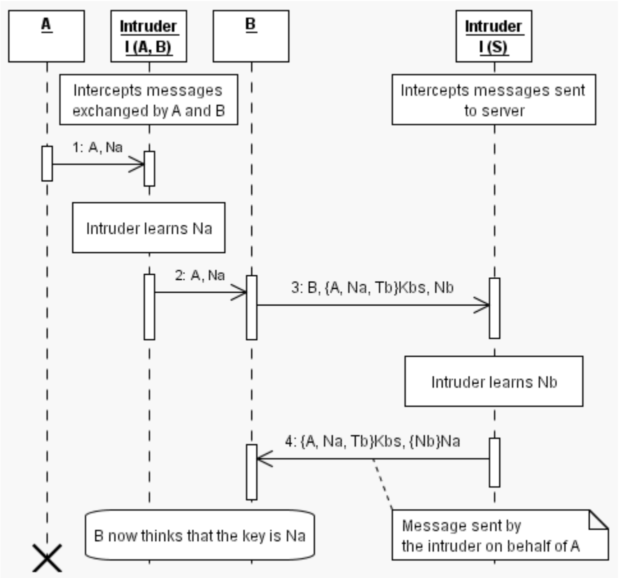

Figure 3. Type flaw attack on the Neuman-Stubblebine authentication protocol

An example of a *basic type flaw* attack is presented in figure 3, where I(A, B) denotes an intruder that is able to intercept messages sent to A and B and I(S) denotes the same intruder capable of intercepting messages sent to the server. In fact, the intruder has total control over the network, it can create new messages, it can compose and decompose messages, replay and delete messages. An intruder with these capabilities corresponds to the intruder model described in the Dolev-Yao security model [13].

Having unlimited access to the communication channel, the intruder intercepts message 1 sent by A to B, thus learning the nonce *Na*. After this, he forwards the received message to B. B then generates a new nonce (*Nb*) and a message encrypted with his shared key, which he both sends to the server, along side with his name. This message is also intercepted by the intruder, who thus learns *Nb*.

In the original message exchange (Figure 1), B reaches final state when he receives a valid final message from A (i.e. message 4). To make participant B reach this state, the intruder sends message 2 (after creating a new encrypted part), intercepted from the same B, to B. Because of structural similarity and because B cannot check the validity of *Kab*, this is actually possible, leading to type confusion and acceptance of the new message as a valid one. Thus, the key that B thinks it shares with A becomes *Na* instead of *Kab*, which was in fact the intended one.

3 Typed security protocol specification

In this section we construct our typed model of security protocols using first order predicate logic [18]. We start by defining a few basic concepts and continue with the formalization of a regular protocol specification. Then, we define our typed specification of security protocols and provide transformation functions from the regular specification to our typed one.

3.1 Basic concepts and considerations

A system consists of a number of communicating agents. A security protocol describes the behavior of these agents also known as *roles.* Thus, specifying a security protocol is reduced to specifying the behavior of the roles involved in the protocol.

Basic sets. The sets that form the foundation of our constructs are the following: $\mathcal{R}$ (denoting a set of roles, for example {*A*, *B*, *C*}, where *A*, *B*, *C* denote role names), $\mathcal{N}$ (denoting a set of nonces, for example {*Na*, *Nb*, *Nc*, *Nt*}, where *Na* is the nonce generated by role *A*, and so on, and *Nt* is the notation we use for timestamps considered to have the same behavior as a nonce), and $\mathcal{F}$ (denoting a set of function names, for example *sym*, representing *symmetric* encryption, and *asym*, representing *asymmetric* encryption).

Cryptographic primitives. The security protocol specifications that we consider use an idealized, *black-box* view on cryptographic primitives (i.e. mathematical constructs like encryption, decryption). This means that the primitives are considered to be implemented using flawless algorithms that guarantee "perfect encryption". The algorithm types we consider are symmetric (i.e. the same key is used for encrypting and decrypting data) and asymmetric (i.e. a public key is used for encryption and a private key for decryption).

Communication model. The communication model corresponds to the "Dolev-Yao" security model [13] where any role can read a message from the communication channel and any role can send a message to the channel. For a message to be considered readable it must correspond to a certain blueprint, taken from the role specification.

3.2 Security protocol specification

Because roles communicate by exchanging messages, to be able to define a role specification we have to define what a message is. To do this, first we define the encryption keys that appear in security protocol messages using the following grammar:

$$\begin{array}{lll} \textit{Keys}: \mathcal{K} ::= & k\ (\textit{session key}) & (1) \\ & \mid sh\ A\ B\ (\textit{shared}\ \text{permanent}\ \textit{key}) & \\ & \mid pk\ A\ (\textit{public key}) & \\ & \mid sk\ A\ (\textit{secret key}) & \end{array}$$

We use the symbol $\mathcal{K}$ to range over keys that appear in messages. In the above definition, the role names are not restricted to only *A* and *B*. If other roles appear in a specification, they may also be used.

The definition of a security protocol message (or simply a message) introduces constructors for encryption (denoted by curly brackets) and pairing. We expand on these after the definition. Thus, a *Message*, or more appropriately a *Message Term*, written as $\mathcal{M}$, is defined as:

$$\textit{Message Term}: \mathcal{M} ::= .\ \mid \mathcal{R} \mid \mathcal{N} \mid \mathcal{K} \mid \mathcal{F}(\mathcal{M}) \mid (\mathcal{M}, \mathcal{M}) \mid \{\mathcal{M}\}_{\mathcal{M}} \qquad (2)$$

To denote an empty message term, we use the "." symbol. The type of the encryption algorithm (symmetric or asymmetric) and the key that is used are denoted by a subscript placed after the "{ }" brackets. If the subscript is not a function of $\mathcal{F}$, then the encryption is considered to be symmetric. Also, if the context allows us (does not lead to confusions) we omit the specification of the function, leaving only the term representing the encryption key, thus denoting a symmetric encryption.

Example message constructions having the same meaning are $\{A\}_{sym(k)}$ and $\{A\}_k$: the name of the role A is encrypted with session key k using a symmetric algorithm. However, not the same thing may be said about these constructs: $\{A\}_{asym(k)}$ and $\{A\}_k$. In the first case, k is used in an asymmetric encryption, while in the second case it is used in a symmetric encryption.

Because we have to distinguish between sent and received messages, we define two predicates *send, recv* : $\mathcal{R} \times \mathcal{R} \times \mathcal{M}$ to denote the sending and receiving of a message having a source role and a destination role. The composition and decomposition of messages are defined inductively by the following rules:

$$send(r, r', t_1) \wedge send(r, r', t_2) \Leftrightarrow send(r, r', (t_1, t_2)), \tag{3}$$

$$send(r, r', t) \wedge send(r, r', f(t_1, \ldots, t_n)) \Leftrightarrow send(r, r', (t, f(t_1, \ldots, t_n))), \tag{4}$$

$$recv(r, r', t_1) \wedge recv(r, r', t_2) \Leftrightarrow recv(r, r', (t_1, t_2)) \tag{5}$$

$$recv(r, r', t) \wedge recv(r, r', f(t_1, \ldots, t_n)) \Leftrightarrow recv(r, r', (t, f(t_1, \ldots, t_n))), \tag{6}$$

where $r, r' \in \mathcal{R}$, $t, t_1, \ldots, t_n \in \mathcal{M}$ and $f \in \mathcal{F}$. r and r' are used to denote a source role and a destination role, respectively.

Thus, we can define a *role specification* as a set of *send* and *recv* predicates, using an index label $i \in \mathcal{I}$ to differentiate between similar occurrences:

$$RoleSpec = \{send_i(r, r', t), recv_i(r, r', t) \mid t \in \mathcal{M}, i \in \mathcal{I}, r, r' \in \mathcal{R}\} \tag{7}$$

Having defined the *role specification*, we can define a *protocol specification* describing the behavior of a number of roles, as the function $ProtSpec = \mathcal{R} \rightarrow RoleSpec$.

An example specification of a role in a *NonceExchange* protocol is:

$$NonceExchange(A) = \{send_1(A, B, (A, \{A, Na\}_{sh\,A\,B})), recv_2(B, A, (\{B, Na, Nb\}_{sh\,A\,B}))\},$$

where A and B are the participating roles, Na is the nonce generated by A, Nb is the nonce generated by B and *sh A B* is the shared key used by A and B to encrypt the exchanged messages.

3.3 Typed security protocol specification

Our typed specification is based on the *Basic Types* defined by the following grammar:

$$\begin{aligned} BasicTypes\colon \tau ::=\; & \mathrm{r}\ (\textit{role type}) \\ \mid\; & \mathrm{n}\ (\textit{nonce type}) \\ \mid\; & \mathrm{k}\ (\textit{session key type}) \\ \mid\; & \mathrm{sh\ A\ B}\ (\textit{shared key type}) \\ \mid\; & \mathrm{pk\ A}\ (\textit{public key type}) \\ \mid\; & \mathrm{sk\ A}\ (\textit{secret key type}) \end{aligned} \tag{8}$$

Comparing the definition of a *Message* from section 3.2 with the previous definition, it can be seen that the regular (non-italic) written letters stand for the types corresponding to the components of a *Message*. We use the symbol τ to denote all the possible basic types.

A *Typed Message*, or *Typed Message Term*, written as $t\mathcal{M}$, is constructed using the basic types τ and has the following definition (using the "." symbol to denote an empty typed message term):

$$Typed\ Message\ Term\colon t\mathcal{M} ::= .\ \mid \tau \mid \mathcal{F}(t\mathcal{M}) \mid (t\mathcal{M}, t\mathcal{M}) \mid \{t\mathcal{M}\}_{t\mathcal{M}} \tag{9}$$

The definition of a *typed role specification* uses the *tsend, trecv* : $\mathcal{R} \times \mathcal{R} \times t\mathcal{M}$ predicates to express sending and receiving of *Typed Messages* from a source role to a destination role and it is similar to the definition of a *role specification* from the previous section:

$$TRoleSpec = \{tsend_i(r, r', t), trecv_i(r, r', t) \mid t \in t\mathcal{M}, i \in \mathcal{I}, r, r' \in \mathcal{R}\} \tag{10}$$

The composition and decomposition rules for the typed messages are defined as:

$$tsend(r, r', t_1) \wedge tsend(r, r', t_2) \Leftrightarrow tsend(r, r', (t_1, t_2)), \tag{11}$$

$$tsend(r, r', t) \wedge tsend(r, r', f(t_1, \ldots, t_n)) \Leftrightarrow tsend(r, r', (t, f(t_1, \ldots, t_n))), \tag{12}$$

$$trecv(r, r', t_1) \wedge trecv(r, r', t_2) \Leftrightarrow trecv(r, r', (t_1, t_2)), \tag{13}$$

$$trecv(r, r', t) \wedge trecv(r, r', f(t_1, \ldots, t_n)) \Leftrightarrow trecv(r, r', (t, f(t_1, \ldots, t_n))), \tag{14}$$

where $r, r' \in \mathcal{R}$, $t, t_1, \ldots, t_n \in t\mathcal{M}$ and $f \in \mathcal{F}$.

Using the same symmetry of thought, as was the case of the *TRoleSpec*, we define a *typed security protocol specification* as the function $TProtSpec = \mathcal{R} \rightarrow TRoleSpec$.

An example modeling of a hypothetical *KeyGeneration* protocol for a role *A* is:

$$\begin{aligned} KeyGeneration(A) = \{ & tsend_1(A, B, (\mathrm{r}, \mathrm{r}, \{\mathrm{r}, \mathrm{n}\}_{\mathrm{sh\ A\ B}})), \\ & trecv_2(B, A, (\mathrm{r}, \{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\ A\ B}}))\}. \end{aligned}$$

3.4 Transformation

Although we can model security protocols using the definition of typed protocol specification, as was the case of the *KeyGeneration* protocol example, to model existing protocols in our typed framework, we need a function that transforms *untyped* messages (i.e. $\mathcal{M}$) into *typed* ones (i.e. $t\mathcal{M}$). For this, we consider a *message type transformation* function $MTr = \mathcal{M} \rightarrow t\mathcal{M}$.

Thus, the transformation of a message term $t \in \mathcal{M}$, with $t_1, \ldots, t_n \in \mathcal{M}$ and $f \in \mathcal{F}$, into a typed message term is defined as:

$$MTr(t) = \begin{cases} \mathrm{r}, & \text{if } t \equiv r \in \mathcal{R} \\ \mathrm{n}, & \text{if } t \equiv n \in \mathcal{N} \\ \mathrm{k}, & \text{if } t \equiv k \in \mathcal{M} \\ \mathrm{sh\ A\ B}, & \text{if } t \equiv sh\ A\ B \in \mathcal{M} \\ \mathrm{pk\ A}, & \text{if } t \equiv pk\ A \in \mathcal{M} \\ \mathrm{sk\ A}, & \text{if } t \equiv sk\ A \in \mathcal{M} \\ f(MTr(t_1), \ldots, MTr(t_n)), & \text{if } t \equiv f(t_1, \ldots, t_n) \\ (MTr(t_1), MTr(t_2)), & \text{if } t \equiv (t_1, t_2) \\ \{MTr(t_1)\}_{MTr(t_2)}, & \text{if } t \equiv \{t_1\}_{t_2} \end{cases} \quad (15)$$

For the transformation of a *role specification* into a *typed role specification* we use a *role transformation* function $RTr = RoleSpec \rightarrow TRoleSpec$.

As in the case of the message transformation, we define a role transformation function for $\rho, \rho_1, \rho_2 \in RoleSpec$, $t \in \mathcal{M}$, $i \in I$ and $r, r' \in \mathcal{R}$ as:

$$RTr(\rho) = \begin{cases} (RTr(\rho_1), RTr(\rho_2)), & \text{if } \rho \equiv (\rho_1, \rho_2) \\ tsend_i(r, r', MTr(t)), & \text{if } \rho \equiv send_i(r, r', t) \\ trecv_i(r, r', MTr(t)), & \text{if } \rho \equiv recv_i(r, r', t) \end{cases} \quad (16)$$

4 Analyzing typed security protocols

To be able to analyze security protocols against *structural attacks*, we must first formalize the considered attacks (*replay* and *basic type flaw*) using our typed framework. This is done in section 4.1. In section 4.2 we model and analyze the key distribution part from the Neuman-Stubblebine (N-S) [12] authentication protocol.

4.1 Attack formalization

In this section we formalize *structural attacks* using our typed specification of security protocols. For simplicity reasons, we use the term *message* to denote a *typed message* and the term *protocol specification* to denote a *typed protocol specification*.

To discover *replay* attacks in the protocol specification of a role $r \in \mathcal{R}$, we first construct the set of all "*send*" messages having any source role $r' \in \mathcal{R}$ and a destination role r'', such that $r'' \in \mathcal{R} - \{r\}$:

$$allsentMsgExcluding(r) = \bigcup_{r' \in R,\, r'' \in R - \{r\}} \{t \mid send_i(r', r'', t) \in TProtSpec(r')\} \quad (17)$$

Similarly, we define the set of all "*recv*" messages given in the protocol specification for the same $r \in \mathcal{R}$, having any source role $r' \in \mathcal{R}$, as:

$$recvdMsg(r) = \bigcup_{r' \in R} \{t \mid recv_i(r', r, t) \in TProtSpec(r)\} \quad (18)$$

Having the set of all messages in the protocol specification not destined for *r*, and the set of messages given in the specification of *r* that may be received by *r*, to determine if role *r* is opened to *replay* attacks, we must only find a message or sub-message in *recvdMsg* that is equal to a message or sub-message in *allsentMsgExcluding*. Formally, we use the *REPLAY* predicate to express the fact that a role $r \in \mathcal{R}$ is opened to *replay* attacks:

$$\begin{aligned} & REPLAY(r) \Leftrightarrow \\ & \quad \exists t_r \in recvdMsg(r), t_s \in allsentMsgExcluding(r), t \in \mathcal{P}(t_r), t' \in \mathcal{P}(t_s) \\ & \quad t = t' \end{aligned} \quad (19)$$

Because *basic type flaw* attacks result from accepted encrypted messages that contain keys (for example session keys generated by a third party server), we can model these attacks using the same line of thought as in the case of *replay* attacks.

To determine if a role $r \in \mathcal{R}$ is opened to *basic type flaw* attacks, we first construct the set of all "*send*" messages specified in the protocol for all the involved roles. We do not remove the messages destined for role r (as was the case of modeling replay attacks), because these messages may be reused in different contexts, thus generating themselves *basic type flaw* attacks.

$$allsentMsg = \bigcup_{r', r'' \in R} \{t \mid send_{\mathrm{i}}(r', r'', t) \in TProtSpec(r')\} \tag{20}$$

Hence, we use the *BASIC-TYPEFLAW* predicate to denote that a role $r \in \mathcal{R}$ is opened to *basic type flaw* attacks:

$$\begin{aligned} &BASIC\text{-}TYPEFLAW(r) \Leftrightarrow \\ &\quad \exists t_{\mathrm{r}} \in recvdMsg(r),\ t_{\mathrm{s}} \in allsentMsg,\ t \in \mathcal{P}(t_{\mathrm{r}}),\ t' \in \mathcal{P}(t_{\mathrm{s}}) \wedge \\ &\quad \exists t_1, t_2, t_3 \subset t,\ t'_1, t'_2 \subset t',\ f \in t,\ bt \in \tau\text{-}\{\mathrm{k}\} \\ &\qquad t = \{t_1, \mathrm{k}, t_2\}_{f(t_3)} \wedge t' = \{t'_1, bt, t'_2\}_{f(t_3)} \wedge \\ &\qquad |t_1| = |t'_1| \wedge |t_2| = |t'_2| \end{aligned} \tag{21}$$

where $recvdMsg(r)$ is the set defined in equation (18), the $\subset$ operator is used to denote that a message term is a sub-term of another message term and the $|t|$ operator returns the length of a message term t. The encryption function f and the encryption key t_3 are the same for the sent and received messages.

4.2 Analyzing the N-S security protocol

Using the structures described in section 3.2, the regular protocol specification of the N-S protocol becomes:

$$\begin{aligned} NS(A) = \{ & send_1(A, B, (A, Na)), \\ & recv_2(S, A, (\{B, Na, k, Nt\}_{sh\,A\,S}, \\ & \qquad \{A, k, Nt\}_{sh\,B\,S}, Nb)), \\ & send_3(A, B, (\{A, k, Nt\}_{sh\,B\,S}, \{Nb\}_k))\} \end{aligned}$$

$$\begin{aligned} NS(B) = \{ & recv_1(A, B, (A, Na)), \\ & send_2(B, S, (B, \{A, Na, Nt\}_{sh\,B\,S}, Nb)), \\ & recv_3(A, B, (\{A, k, Nt\}_{sh\,B\,S}, \{Nb\}_k))\} \end{aligned}$$

$$\begin{aligned} NS(S) = \{ & recv_1(B, S, (B, \{A, Na, Nt\}_{sh\,B\,S}, Nb)), \\ & send_2(S, A, (\{B, Na, k, Nt\}_{sh\,A\,S}, \\ & \qquad \{A, k, Nt\}_{sh\,B\,S}, Nb))\} \end{aligned}$$

By applying the role transformation function from section 3.4, we have the following typed specification:

$$\begin{aligned} RTr(NS(A)) = \{ & tsend_1(A, B, (\mathrm{r}, \mathrm{n})), \\ & trecv_2(S, A, (\{\mathrm{r}, \mathrm{n}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,A\,S}}, \\ & \qquad \{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \mathrm{n})), \\ & tsend_3(A, B, (\{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \{\mathrm{n}\}_{\mathrm{k}}))\} \end{aligned}$$

$$\begin{aligned} RTr(NS(B)) = \{ & trecv_1(A, B, (\mathrm{r}, \mathrm{n})), \\ & tsend_2(B, S, (\mathrm{r}, \{\mathrm{r}, \mathrm{n}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \mathrm{n})), \\ & trecv_3(A, B, (\{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \{\mathrm{n}\}_{\mathrm{k}}))\} \end{aligned}$$

$$\begin{aligned} RTr(NS(S)) = \{ & recv_1(B, S, (\mathrm{r}, \{\mathrm{r}, \mathrm{n}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \mathrm{n})), \\ & send_2(S, A, (\{\mathrm{r}, \mathrm{n}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,A\,S}}, \\ & \qquad \{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \mathrm{n}))\} \end{aligned}$$

Analyzing this typed specification for role B, the $REPLAY(B)$ predicate will hold, for example, for $trecv_1(A, B, (\mathrm{r}, \mathrm{n}))$ where the (r, n) message may be extracted from the $tsend_2(B, S, (\mathrm{r}, \{\mathrm{r}, \mathrm{n}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \mathrm{n}))$ message. This leads to a simple *replay* attack that is created by sending to B the message that himself has generated.

Because of message structure similarities, the $BASIC\text{-}TYPEFLAW(B)$ predicate will also hold in many cases. For example, in $trecv_3(A, B, (\{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \{\mathrm{n}\}_{\mathrm{k}}))$, the $\{\mathrm{r}, \mathrm{k}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}$ term may be generated in $tsend_2(B, S, (\mathrm{r}, \{\mathrm{r}, \mathrm{n}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}, \mathrm{n}))$, thus the first nonce that is sent in the $\{\mathrm{r}, \mathrm{n}, \mathrm{n}\}_{\mathrm{sh\,B\,S}}$ message term becomes the key.

5 Related work

Until now, typing has been used in the literature for checking secrecy violations in [2, 5, 6] (by verifying if a message component marked as having the type *private* is made *public*), for controlling network message flow in [14], for defending against type flaw attacks in [7] or for state space reduction in [9] (by using message typing).

Although the specification of security protocols in the "Typed MSR" [9] model allows the definition of types for message components, it is not syntactically typed as is the case of our model. Because we create a specification based only on message types, a simple syntactical search may be conducted to find replay or type flaw attacks.

In [15], a "zipper" (comparison) procedure is presented for detecting type flaws. The level of abstraction used is much lower than ours, because of the physical length of messages that is also included in the model. This is why, the procedure can only be used as described in [15] and cannot be used by existing protocol verification tools, as is the case of our model.

6 Conclusions and future work

This paper presented a typed specification for security protocols that may be used by existing protocol verification methods and tools as an input model.

The proposed abstract model captures the message structure of security protocols using types, hence not considering the actual message component values as regular specifications do. This is why we are able to conduct simple syntactical analysis on security protocols for detecting *replay* or *type flaw* attacks.

Because we construct the abstract model by simply replacing the message terms with their corresponding types, thus not destroying the data flow or message term ordering, the typed protocol model may be verified by existing tools, like the model checking tool *Maude* [16, 17].

Although our specification allows for a rapid syntactical analysis of security protocols against *replay* attacks and *basic type flaw* attacks, it is not yet equipped with enough formal power (message term reduction techniques, role knowledge specification) to allow a syntactical analysis for *all type flaw* attacks. This is why we consider this as a remaining future work by the end of which we will also be able to analyze multiple protocol interactions [7, 8] by using a simple syntactical message comparison.

Refrences:

[1] Clarke, E. M., Grumberg, O., and Peled, D, Model checking, *MIT Press*, 1999.

[2] M. Abadi, Secrecy by typing in security protocols, *Journal of the ACM*, 46(5), 1999, pp 749-786.

[3] Lawrence C. Paulson, The inductive approach to verifying cryptographic protocols, *Journal of Computer Science*, 1998, pp. 85-128.

[4] Bruno Blanchet, Automatic Verification of Cryptographic Protocols: A Logic Programming Approach, In *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Sweden, August 2003, pp. 1-3.

[5] M. Abadi and A. D. Gordon, A calculus for cryptographic protocols: The spi calculus, *Information and Computation*, 148(1), 1999, pp 1-70.

[6] M. Abadi, Bruno Blanchet, Secrecy types for Asymmetric Communication, *Theor. Comput. Sci.*, 3(298), 2003, pp 387-415.

[7] Cas J. F. Cremers, Feasability of Multi-Protocol Attacks, *ARES*, 2006, pp. 287-294.

[8] C.J.F. Cremers, Verification of multi-protocol attacks, *Computer Science Report CSR*, Eindhoven University of Technology, 2005, pp 5-10.

[9] I. Cervesato, Typed Multiset Rewriting Specifications of Securiy Protocols, *First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology*, 2000, pp. 1-43.

[10] G. Hollestelle, Systematic Analysis of Attacks on Security Protocols, Master's Thesis, Technical University of Eindhover, Department of Mathematics and Computer Science, November 2005.

[11] J. Clark and J. Jacob, Attacking Authentication Protocols, *High Integrity Systems*, 1(5), 1996, pp. 465-474.

[12] B. Clifford Neuman and Stuart G. Stubblebine, A note on the use of timestamps as nonces, *Operating Systems Review,* 27(2), 1993, pp 10-14.

[13] Dolev, D., Yao, A., On the security of public key protocols, *IEEE Transactions on Information Theory*, IT-29, 1983, pp. 198-208.

[14] Guoqiang Li, Bochao Liu, Li Xin, Mizuhito Ogawa, Type-directed Trace Analysis of Security Protocols in Process Calculus, *JSSST*, 2005, available at: nue.riec.tohoku.ac.jp/jssst2005/papers/05025.pdf

[15] Catherine Meadows, Identifying potential type confusion in authenticated messages, *16th IEEE Computer Security Foundations Workshop (CSFW'03)*, 2002, p. 62.

[16] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Mesequer, and C. Talcott, Maude Manual Version 2.1, 2004, http://maude.cs.uiuc.edu.

[17] Peter Csaba Olveczky, Formal Modeling and Analysis of Distributed Systems in Maude, Lecture Notes, University of Oslo, 2005.

[18] Simon Thompson, *Type Theory and Functional Programming*, Addison-Wesley, 1991.

[19] Security Protocol Open Repository, available at: www.lsv.ens-cachan.fr/spore.

[20] Yafen Li, Wuu Yang, Ching-Wei Huang, On Preventing Type Flaw Attacks on Security Protocols With a Simplified Tagging Scheme, *J. Inf. Sci. Eng.*, 21(1), 2005, pp. 59-84.