Formal verification of IFF & NSLPK authentication protocols with CiMPG

Thet Wai Mon, Shuho Fujii, Duong Dinh Tran, and Kazuhiro Ogata

School of Information Science, Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {thetwaimon,s1910186.duongtd.ogata}@jaist.ac.jp

Abstract—Proof scores are programs written in an algebraic specification language, such as CafeOBJ, to conduct formal verification. Thus, the proof score approach to formal verification (PSA2FV) can be regarded as a kind of proving by programming and then flexible. PSA2FV, however, is subject to human errors. To address the issue, a proof assistant called CiMPG was developed for CafeInMaude, the world’s second implementation of CafeOBJ. Furthermore, a proof generator called CiMPA was developed to benefit from the strong points of both PSA2FV and CiMPA. Although some case studies have been conducted with CiMPA, it is necessary to do some more. The present paper reports on case studies in which it is formally verified that two authentication protocols enjoy desired properties with CiMPG.

Keywords—algebraic specification language; proof assistant; proof generator; authentication protocol

I. INTRODUCTION

Theorem proving that systems enjoy some desired properties by writing proof scores have been intensively used. This approach uses observational transition systems (OTSs) [1] as state machines to formalize systems. Then, the OTSs are specified in CafeOBJ [2], which is a formal specification language. Formal verification is conducted by writing what is called “proof scores” [1] in CafeOBJ and executing them with CafeOBJ. Although writing proof scores is flexible to conduct formal verification, the proof may contain some flaws since proof scores are subject to human errors (e.g., users may overlook some cases during the proof).

CafeInMaude is the second implementation of Maude of CafeOBJ in addition to the original implementation in Common Lisp, where Maude [3] is a sibling language of CafeOBJ. CafeInMaude introduces CafeOBJ specifications into the Maude system. It comes with two extension tools CafeInMaude Proof Assistant (CiMPA) and CafeInMaude Proof Generator (CiMPG) [4]. CiMPA is a proof assistant that allows users to write proof scripts in order to prove invariant properties on their CafeOBJ specifications. CiMPG provides a minimal set of annotations for identifying proof scores and generating CiMPA scripts for these proof scores. Using CiMPA to develop the formal verification by writing proof scripts can help us to avoid the flaw made by human users as in the proof score approach. However, it is often the case that CiMPA is not flexible enough to conduct formal verification. CiMPG allows users to combine the flexibility of the proof score approach and the reliability of CiMPA. Given proof scores that should be slightly annotated, CiMPG generates proof scripts for CiMPA. Feeding the generated proof scripts into CiMPA, if CiMPA successfully discharges all goals, the proof scores are correct for the goals.

This paper presents the formal verification with CiMPA and CiMPG of two authentication protocols: Identity-Friend-or-Foe authentication protocol (IFF) [5] and Needham-Schroeder-Lowe Public Key authentication protocol (NSLPK) [6]. The former is a simple protocol used to check if a principal (or an agent) is a member of a group. The latter is an advanced authentication protocol, which is a modification of NSPK protocol [7] made by Lowe. We use CiMPA and CiMPG to formally verify that: (1) IFF enjoys the identifiable property, and (2) NSLPK enjoys the nonce secrecy property and one-to-many correspondence property.

Although it has been formally verified that NSLPK enjoys the nonce secrecy property with CiMPG [4], we are the first to formally verify that NSLPK enjoys the one-to-many correspondence property with CiMPG as well as CiMPA. IFF is a tiny protocol but nobody has formally verified that it enjoys a desired property with either CiMPA or CiMPG. All specifications and proofs presented in this paper are available at https://github.com/twmon14/fvap.

II. FORMAL VERIFICATION OF IFF

IFF [5] is used to check if a principal is a member of a group. The IFF protocol can be described as the following two message exchanges:

- Check $p \rightarrow q : r$
- Reply $q \rightarrow p : \varepsilon_k(r, q)$

Each principal (or agent) such as $p$ and $q$ belongs to only one group. A symmetric key is given to each group, whose members share the key, and keys are different from group to group. If a principal $p$ wants to check if a principal $q$ is a member of the $p$’s group, $p$ generates a fresh random number $r$ and sends it to $q$ as a Check message. On receipt of the message, $q$ sends back to $p$ a Reply message that consists of $r$ and ID $q$ encrypted by the symmetric key $k$ of the $q$’s group. When $p$ receives the Reply message, $p$ tries to decrypt the ciphertext received with the symmetric key of the $p$’s group. If the decryption succeeds and the plaintext...
which the enemy can use as his/her storage. Any message
where Msg
where Key
(i.e., the empty multiset) means that no messages have
been sent. The enemy gleans as much information as
possible from messages flowing in the network and createsake messages based on the gleaned information, provided
that the enemy cannot break the perfect cryptosystem.

A. Formal Specification of the Protocol

We first declare the operator enc to specify the ciphertexts
used in the protocol as follows:

\[
\begin{align*}
\text{op enc} &: \text{Key Rand Prin} \to \text{Cipher} \\
\text{op k} &: \text{Cipher} \to \text{Key} \\
\text{op r} &: \text{Cipher} \to \text{Rand} \\
\text{op p} &: \text{Cipher} \to \text{Prin}
\end{align*}
\]

where Key is the sort (or type) representing symmetric keys,
Rand is the sort denoting random numbers, Prin is the sort
representing principals, and Cipher is the sort denoting
ciphertexts. Given a key \( k \), a random number \( r \) and a principal
\( p \), \( \text{enc}(k,r,p) \) denotes the ciphertext obtained by encrypting
\( r \) and \( p \) with \( k \). Operators \( k \), \( r \) and \( p \) return the first, second
and third arguments of \( \text{enc}(k,r,p) \), respectively.

We specify two messages Check and Reply by two operators
cm and rm as follows:

\[
\begin{align*}
\text{op cm} &: \text{Prin Prin Prin Rand} \to \text{Msg} \\
\text{op rm} &: \text{Prin Prin Prin Cipher} \to \text{Msg}
\end{align*}
\]

where \( \text{Msg} \) is the sort denoting messages. The first, second
and third arguments of each of \( \text{cm} \) and \( \text{rm} \) are the actual
creator, the seeming sender and the receiver of the corre-
"sponding message. The first argument is meta-information
that is only available to the outside observer and the principal
that has sent the corresponding message, and that can not be
forged by the enemy; while the remaining arguments may
be forged by the enemy.

The network is modeled as a multiset of messages, in
which the enemy can use as his/her storage. Any message
that has been sent or put once into the network is supposed
to be never deleted from the network because the enemy
can reply the message repeatedly, although the enemy can not
forge the first argument. Consequently, the empty network
(i.e., the empty multiset) means that no messages have been
sent.

The enemy tries to glean two kinds of values from the
network, which are random numbers and ciphertexts. The
collections of these values gleaned by the enemy are denoted
by operators \( \text{rands} \) and \( \text{ciphers} \), which are declared as follows:

\[
\begin{align*}
\text{op rands} &: \text{Network} \to \text{ColRands} \\
\text{op ciphers} &: \text{Network} \to \text{ColCiphers}
\end{align*}
\]

where \( \text{Network} \) is the sort denoting networks, \( \text{ColRands} \)
is the sort denoting collections of random numbers, and
\( \text{ColCiphers} \) is the sort denoting collections of ciphertexts.

\( \text{ciphers} \) is defined by the following equations:

\[
\begin{align*}
\text{eq C} \in \text{ciphers}(\text{void}) &= \text{false} \\
\text{ceq C} \in \text{ciphers}(M, NW) &= \text{true if rm?(M)} \\
&\quad \text{and } C = c(M) \\
\text{ceq C} \in \text{ciphers}(M, NW) &= C \setminus \text{ciphers}(NW) \\
&\quad \text{if not} (\text{rm?}(M) \text{ and } C = c(M)).
\end{align*}
\]

where \( \text{void} \) denotes the empty multiset (or empty network),
operator \( \text{rm?} \) checks if a given message is a Reply message,
operator \( \text{c} \) takes a Reply message as a parameter and returns
its ciphertext (i.e., the fourth argument of \( \text{rm} \) operator), \( \text{\in} \)
is an infix operator checking the existence of an element in
a collection, and operator \( \text{r} \) returns the data constructor of nonempty multisets. The equations say that
a ciphertext \( C \) is available to the enemy iff there exists a
Reply message whose content is \( C \). \( \text{rands} \) can be defined
likewise.

Now, we are ready to specify the protocol. We use two obsevational functions \( \text{nw} \) and \( \text{ur} \) to observe the network
and the set of used random numbers, respectively as follows:

\[
\begin{align*}
\text{op nw} &: \text{Sys} \to \text{Network}. \text{op ur} &: \text{Sys} \to \text{URands}
\end{align*}
\]

where \( \text{Sys} \) is the sort denoting the state space of IFF,
\( \text{URands} \) is the sort denoting the sets of random numbers.

We use five transitions together with one constant of \( \text{Sys} \)
to represent an arbitrary initial state as follows:

\[
\begin{align*}
\text{op init} &: \to \text{Sys (const)} \\
\text{op sdrm} &: \text{Sys Prin Msg} \to \text{Sys (const)} \\
\text{op sdcm} &: \text{Sys Prin Prin Rand} \to \text{Sys (const)} \\
\text{op fkcm1} &: \text{Sys Prin Prin Cipher} \to \text{Sys (const)} \\
\text{op fkrm1} &: \text{Sys Prin Prin Rand} \to \text{Sys (const)} \\
\text{op fkrm2} &: \text{Sys Prin Prin Rand} \to \text{Sys (const)}
\end{align*}
\]

\( \text{sdcm} \) and \( \text{sdcm} \) formalize sending Check and Reply
messages exactly following the protocol, respectively. The
remaining actions \( \text{fkcm1}, \text{fkrm1}, \text{fkrm2} \) are the
enemy’s faking messages, which can be understood as follows:

- \( \text{fkcm1} \): a random number \( R \) is available to the enemy,
  the enemy fakes and sends a Check message using \( R \).
- \( \text{fkrm1} \): a ciphertext \( C \) is available to the enemy,
  the enemy fakes and sends a Reply message using \( C \).
- \( \text{fkrm2} \): a random number \( R \) is available to the enemy,
  the enemy fakes and sends a Reply message using \( R \).

\( \text{sdcm} \) is defined as follows:

\[
\begin{align*}
\text{ceq nw(sdcms(S,P1,P2,R))} &= \text{cm(P1,P1,P2,R)} \to \text{nw(S)} \text{ if not c-sdms(S,P1,P2,R)} \\
\text{ceq ur(sdcms(S,P1,P2,R))} &= \text{R ur(S)} \text{ if not c-sdms(S,P1,P2,R)}
\end{align*}
\]
where c-sdcm(S, P1, P2, R) is not (R \in ur(S)). The equations say that if c-sdcm(S, P1, P2, R) is true (i.e., R has not been used), then the Check message cm(P1, P2, R) is put into the network nw(S). R is put into ur(S) in the state denoted by sdc(S, P1, P2, R); if c-sdcm(S, P1, P2, R) is false, nothing changes. The remaining transitions can be defined likewise.

B. Formal Verification with CiMPA

One property of IFF we would like to confirm is whenever p receives a valid Reply message from q, q is always a member of the p’s group. Such property is called identifiable property in this paper. The property is specified as follows:

\[ \text{iff} \text{ :sys Key Rand} \rightarrow \text{Bool} . \]
\[ \text{eq \iff(S:Sys, P:Prin, K:Key, R:Rand) = (\text{not}(K = k(\text{enemy})) \text{ and rm}(P1, P2, P3, \text{enc}(K, R, P2)) \text{ \in nw(S)) implies not}(P2 = \text{enemy}) .} \]

We describe how to prove that IFF enjoys the property by writing proof scripts and running with CiMPA. In the proof of \iff, we need to use a lemma \iff2 that is as follows:

\[ \text{op invl : Sys Prin Prin Key Rand} \rightarrow \text{Bool} . \]
\[ \text{eq invl(S, P1, P2, P3, K, R) = ((\text{not}(K = k(\text{enemy})) \text{ and rm}(P1, P2, P3, \text{enc}(K, R, P2)) \text{ \in nw(S)) implies not}(P2 = \text{enemy}) .} \]

where k(\text{enemy}) denotes the symmetric key of the group to which the enemy belongs to.

The proof starts with the goals we need to prove:

\[ \text{open IFF} . \]
\[ \text{eq [iff1 :nonexec] : invl(S:Sys, P:Prin, P1:Prin, P0:Prin, K:Key, R:Rand) = true .} \]
\[ \text{eq [iff :nonexec] : inv2(S:Sys, K:Key, R:Rand) = true .} \]

where IFF is the module in which the specification of IFF together with invl and inv2 are available. :nonexec instructs CafeInMaude not to use the equations as rewrite rules.

Then, we select S with the command \ind on as the variable on which we start proving the goals by simultaneous induction:

\[ \text{ind on (S:Sys) : apply(si)} \]

The command :apply(si) starts the proof by simultaneous induction on S, generating six goals for fkcm1, fkrm1, fkrm2, init, sdc, and sdrm, where si stands for simultaneous induction. Each goal consists of two equations to prove, corresponding to invl and inv2. With the first goal for fkcm1, we first apply theorem of constants by using the command: :apply(tc). The command generates two sub-goals as follows:

1-1. TC \text{eq [iff1 :nonexec] : invl(fkcm1(S:Sys, P#Prin, P0#Prin, K@Key, R@Rand), P#Prin, P0#Prin, K@Key, R@Rand) = true .}
1-2. TC \text{eq [iff :nonexec] : inv2(fkcm1(S:Sys, P#Prin, P0#Prin, R@Rand), K@Key, R@Rand) = true .}

The command :apply(tc) replaces CafeOBJ variables with fresh constants in goals. S#Sys, P#Prin, P0#Prin, and R@Rand are fresh constants introduced by :apply(si), while P#Prin, P0#Prin, K@Key, and R@Rand are fresh constants introduced by :apply(tc). To discharge goal 1-1, the following commands are first introduced:

\[ \text{:def cl = :ctf [R@Rand \in rands(nw(S#Sys))] .} \]
\[ \text{:apply(cl)} \]

Goal 1-1 is split into two sub-goals 1-1-1 and 1-1-2 correspond to when R@Rand \in rands(nw(S#Sys)) holds and does not hold, respectively. Then, two sub-goals are discharged by the following commands:

\[ \text{:imp [iff1] by \{(K:Key <- K@Key ; P0:Prin <- P0@Prin ; P1:Prin <- P1@Prin ; P:Prin <- P0@Prin ; R:Rand <- R@Rand ;\}} \]
\[ \text{:apply(rd) \}} \]
\[ \text{:imp [iff1] by \{(K:Key <- K@Key ; P0:Prin <- P0@Prin ; P1:Prin <- P1@Prin ; P:Prin <- P0@Prin ; R:Rand <- R@Rand ;\}} \]
\[ \text{:apply(rd) \}} \]

The induction hypothesis is instantiated by replacing the variables with the fresh constants and the instance is used as the premise of the implication. For example, P1:Prin is replaced with P1@Prin. Then, :apply(rd) is used to check if the current goal can be discharged. Two goals 1-1-1 and 1-1-2 are discharged in this case. The current goal is changed to 1-2.

Goal 1-2 is split into two sub-goals and they are discharged by the following commands:

\[ \text{:def c2 = :ctf [R@Rand \in rands(nw(S#Sys))] .} \]
\[ \text{:apply(c2) \}} \]
\[ \text{:imp [iff] by \{K:Key <- K@Key ; R:Rand <- R@Rand ;\}} \]
\[ \text{:apply(rd) \}} \]
\[ \text{:imp [iff] by \{K:Key <- K@Key ; R:Rand <- R@Rand ;\}} \]
\[ \text{:apply(rd) \}} \]

We have all done with goal 1, the current goal moves to 2. With goal 2, we first introduce the following commands to conduct case splitting:

\[ \text{:def c3 = :ctf [C#Cipher \in ciphers(nw(S#Sys))] .} \]
\[ \text{:def c4 = :ctf \{eq P#Prin = enemy .\}} \]
\[ \text{:def c5 = :ctf \{eq P#Prin = P1@Prin .\}} \]
\[ \text{:def c6 = :ctf \{eq P0#Prin = P0@Prin .\}} \]
\[ \text{:def c7 = :ctf \{eq k(C#Cipher) = K@Key .\}} \]
\[ \text{:def c8 = :ctf \{eq r(C#Cipher) = R@Rand .\}} \]
\[ \text{:def c9 = :ctf \{eq p(C#Cipher) = P1@Prin .\}} \]
\[ \text{:def c10 = :ctf \{eq K@Key = k(enemy) .\}} \]
\[ \text{:apply(c3) \}} \]
\[ \text{:apply(c4) \}} \]
\[ \text{:apply(c5) \}} \]
\[ \text{:apply(c6) \}} \]
\[ \text{:apply(c7) \}} \]
\[ \text{:apply(c8) \}} \]
\[ \text{:apply(c9) \}} \]
\[ \text{:apply(c10) \}} \]

Case splittings are carried out based on one Boolean term and seven equations. The first sub-goal in which the Boolean term is true and seven equations hold can be discharged:
open IFF.
:proof(iff)
op s : -> Sys. ops a b c d e : -> Prin.
op k : -> Key. ops r1 r2 : -> Rand.
eq (r2 \in rands(nw(s))) = true.
red invl(s,a,b,c,k,r1)
implies invl(fkcm1(s,d,e,r2),a,b,c,k,r1).
close

Moreover, we need to add one more open-close fragment to the proof scores, which is as follows:
open IFF.
:proof(iff)
close

where iff is just an identifier, can be replaced by another one that is more preferred.

Feeding the annotated proof scores into CiMPG, CiMPG generates the proof script for CiMPA. The generated proof script is quite similar to the one written manually. Feeding the generated proof script into CiMPA, CiMPA discharges all goals, confirming that the proof scores are correct.

III. FORMAL VERIFICATION OF NSLPK

NSLPK [6] is a modification of NSPK authentication protocol [7] made by Lowe. The NSLPK protocol can be described as the following three message exchanges:
Init \( p \rightarrow q : \varepsilon_q(n_p, p) \)
Resp \( q \rightarrow p : \varepsilon_p(n_p, n_q, q) \)
Ack \( p \rightarrow q : \varepsilon_q(n_q) \)

Each principal such as \( p \) and \( q \) has a pair of keys: public and private keys. \( \varepsilon_p(m) \) denotes the ciphertext obtained by encrypting the message \( m \) with the principal \( p \)'s public key. \( n_p \) is a nonce (a random number) generated by principal \( p \). A nonce is a unique and non-guessable number that is used only once time. Again, we also suppose that the cryptosystem used is perfect.

A. Formal Specification of the Protocol

We introduce the following three operators to represent the ciphertexts used in the protocol:

\[
:ctf \quad \text{enc}(K@Key, R@Rand, \text{enemy})
\]
where Nonce is the sort denoting the nonce numbers; Cipher1, Cipher2, and Cipher3 are the sorts denoting three kinds of ciphertexts contained in Init, Resp, and Ack messages, respectively. Given principals p, q and a nonce \( n_p \) term \( \text{enc1}(q, n_p, p) \) denotes the ciphertext \( \varepsilon_q(n_p, p) \) obtained by encrypting \( n_p \) and \( p \) with principal \( q \)'s public key. \text{enc2} and \text{enc3} can be understood likewise.

We specify three messages used in NSLPK as follows:

\[
\begin{align*}
\text{op m1 : Prin Prin Cipher1} & \rightarrow \text{Msg} \\
\text{op m2 : Prin Prin Cipher2} & \rightarrow \text{Msg} \\
\text{op m3 : Prin Prin Cipher3} & \rightarrow \text{Msg}
\end{align*}
\]

\( \text{Msg} \) as well as the first three arguments of each operator can be understood as in the specification of IFF explained in the last section.

The intruder tries to glean four kinds of values from the network, which are nonces and three kinds of ciphertexts. Then, we use following four operators to denote those values:

\[
\begin{align*}
\text{op cnonce : Network} & \rightarrow \text{ColNonce} \\
\text{op cenc1 : Network} & \rightarrow \text{ColCipher1} \\
\text{op cenc2 : Network} & \rightarrow \text{ColCipher2} \\
\text{op cenc3 : Network} & \rightarrow \text{ColCipher3}
\end{align*}
\]

where Network is the sort denoting networks (i.e., multisets of messages) and Col\( X \) is a sort denoting collections of values corresponding to the sort \( X \). The equations defining \text{cenc1} are as follows:

\[
\begin{align*}
\text{eq E1 \in cenc1(\text{void})} & = \text{false} . \\
\text{ceq E1 \in cenc1(M, NW)} & = \text{true if m1(M) and} \\
& \text{not(key(cipher1(M)) = intruder) and} \\
& \text{E1 = cipher1(M)} . \\
\text{ceq E1 \in cenc1(NW)} & = \text{E1 \in cenc1(NW) if} \\
& \text{not(m1(M)) and E1 = cipher1(M)} \\
& \text{and not(key(cipher1(M)) = intruder)} .
\end{align*}
\]

where \( E1 \) is a CafeOBJ variable of Cipher1. \text{m1?} checks if a given message is an Init message. Operator cipher1 takes an Init message as an argument and returns its ciphertext (i.e., the fourth argument of \text{m1} operator). Operator key takes a ciphertext as an argument and returns the principal in which the ciphertext is encrypted with its public key. \( \text{void}, M, NW, \) as well as \( (M, NW) \) can be understood as explained in the last section. The equations say that a ciphertext \( E1 \) is available to the intruder iff there exists an Init message whose content is \( E1 \) and \( E1 \) is not encrypted by the intruder’s public key. Let us note that, if \( E1 \) is encrypted by the intruder’s public key, \( E1 \) can be rebuilt by the intruder. \text{cnonce}, \text{cenc2}, and \text{cenc3} can be defined likewise.

We use two observers, nine transitions, together with one constant that represents an arbitrary initial state to specify NSLPK as follows:

\[
\begin{align*}
\text{op ur : Sys} & \rightarrow \text{URand} . \text{op nw : Sys} \rightarrow \text{Network} \\
\text{op init :} & \rightarrow \text{Sys} \{\text{constr}\}
\end{align*}
\]

where \( \text{URand} \) is the sort denoting sets of random numbers. \( \text{ur, nw, and init} \) can be understood as in the last section. The first three transitions formalize sending messages exactly following the protocol, while the remaining formalize the intruder’s faking messages, which can be understood as follows:

- \( \text{fkm11}, \text{fkm21}, \text{and fkm31}: \) a ciphertext \( C \) is available to the intruder, the intruder fakes and sends a/an Init, or Resp, or Ack message using \( C \), respectively.
- \( \text{fkm12} \) and \( \text{fkm32} \): a nonce \( N \) is available to the intruder, the intruder fakes and sends an Init or Ack message using \( N \), respectively.
- \( \text{fkm22} \): two nonces \( N1 \) and \( N2 \) are available to the intruder, the intruder fakes and sends a Resp message using \( N1 \) and \( N2 \).

Let \( S \) be a CafeOBJ variable of Sys, and \( P \) and \( Q \) are CafeOBJ variables of Prin. \text{fkm11} is defined as follows:

\[
\begin{align*}
\text{eq ur(fkm11(S, P, Q, E1))} & = \text{ur(S)} . \\
\text{ceq nw(fkm11(S, P, Q, E1))} & = \text{m1(intruder, P, Q, E1)} \\
& \text{, nw(S) if c-fkm11(S, P, Q, E1)} . \\
\text{ceq fkm11(S, P, Q, E1)} & = S \\
& \text{if not c-fkm11(S, P, Q, E1)} .
\end{align*}
\]

where \( \text{c-fkm11}(S, P, Q, E1) \) is \( E1 \in \text{cenc1(nw(S))} \), intruder is a constant of Prin denoting the intruder. The equations say that if \( c\text{-fkm11}(S, P, Q, E1) \) is true, then the Init message \( m1(\text{intruder, P, Q, E1}) \) is put into the network \( \text{nw(S)} \), \( \text{ur(S)} \) does not change in the state denoted by \( \text{fkm11}(S, P, Q, E1) \); if \( c\text{-fkm11}(S, P, Q, E1) \) is false, nothing changes. The remaining transitions can be defined likewise.

### B. Formal Verification with CiMPA and CiMPG

There are two properties of NSLPK that we would like to verify namely nonce secrecy property and one-to-many correspondence property. The former says that all nonces available to the intruder are those created by the intruder or those created for the intruder. Let \( N \) be a CafeOBJ variable of Nonce, we specify the nonce secrecy property as follows:

\[
\text{eq inv130(S, N) = (N \in cnonce(nw(S)) \implies creator(N) = intruder)}
\]
The one-to-many correspondence property is specified by the following two equations:

$$\text{eq\ inv100}(S,E1) = (E1 \in \text{cenc1}(nw(S))) \implies \text{not}(\text{key}(E1) = \text{intruder})$$

$$\text{eq\ inv110}(S,E2) = (E2 \in \text{cenc2}(nw(S))) \implies \text{not}(\text{key}(E2) = \text{intruder})$$

$$\text{eq\ inv120}(S,E3) = (E3 \in \text{cenc3}(nw(S))) \implies \text{not}(\text{key}(E3) = \text{intruder})$$

$$\text{eq\ inv130}(S,E3) = (E3 \in \text{cenc3}(nw(S))) \implies \text{not}(\text{key}(E3) = \text{intruder})$$

$$\text{eq\ inv140}(S,E1) = (E1 \in \text{cenc1}(nw(S)) \land \text{principal}(E1) = \text{intruder})$$

$$\text{eq\ inv150}(S,E2) = (E2 \in \text{cenc2}(nw(S)) \land \text{principal}(E2) = \text{intruder})$$

$$\text{eq\ inv160}(S,N) = (\text{creator}(N) = \text{intruder}) \implies \text{not}(\text{key}(E2) = \text{intruder})$$

$$\text{eq\ inv170}(S,P,Q,R,N) = (\text{not}(P = \text{intruder}) \land m1(P,P,Q,\text{enc1}(Q,n(P,Q,R),P)) \in nw(S) \land m2(Q,P,\text{enc2}(P,n(P,Q,R),N,Q)) \in nw(S) \implies m3(P,P,Q,\text{enc3}(Q,n(P,Q,R))) \in nw(S))$$

$$\text{eq\ inv180}(S,P,Q,R,N) = (\text{not}(Q = \text{intruder}) \land m1(P,P,Q,\text{enc1}(Q,n(P,Q,R),P)) \in nw(S) \land m3(Q,Q,P,\text{enc2}(P,n(P,Q,R),Q,Q)) \in nw(S) \implies m3(P,P,Q,\text{enc3}(Q,n(P,Q,R)))) \in nw(S)$$

where P1 & Q1 are CafeOBJ variables of Prin, R is a CafeOBJ variable of Rand. inv170 says that whenever P successfully sent an Init message to Q, and received a corresponding Resp seemingly from Q, the principal that P is communicating with is really Q even though there are malicious principals (e.g., Q1). inv180 can be understood likewise.

To verify the nonce secrecy property, we prove that inv130 is an invariant of the OTS formalizing NSLPK. The formal verification is also conducted in two ways: by writing proof scripts with CiMPA and by using CiMPG to generate proof scores from proof scripts. Both of them require the use of the following lemmas:

$$\text{eq\ inv100}(S,E1) = (E1 \in \text{cenc1}(nw(S))) \implies \text{not}(\text{key}(E1) = \text{intruder})$$

$$\text{eq\ inv110}(S,E2) = (E2 \in \text{cenc2}(nw(S))) \implies \text{not}(\text{key}(E2) = \text{intruder})$$

$$\text{eq\ inv120}(S,E3) = (E3 \in \text{cenc3}(nw(S))) \implies \text{not}(\text{key}(E3) = \text{intruder})$$

$$\text{eq\ inv130}(S,E3) = (E3 \in \text{cenc3}(nw(S))) \implies \text{not}(\text{key}(E3) = \text{intruder})$$

$$\text{eq\ inv140}(S,E1) = (E1 \in \text{cenc1}(nw(S)) \land \text{principal}(E1) = \text{intruder}) \implies \text{not}(\text{key}(E2) = \text{intruder})$$

$$\text{eq\ inv150}(S,E2) = (E2 \in \text{cenc2}(nw(S)) \land \text{principal}(E2) = \text{intruder}) \implies \text{not}(\text{key}(E3) = \text{intruder})$$

$$\text{eq\ inv160}(S,N) = (\text{creator}(N) = \text{intruder}) \implies \text{not}(\text{key}(E2) = \text{intruder})$$

where E2 and E3 are CafeOBJ variables of Cipher2 and Cipher3, respectively.

In each way of verification, what we need to do is quite similar to what we have described in the last section with formal verification of IFF. However, with CiMPG, we also need to make some modifications to the existing proof scores. Let us consider an example in which we want to split the current case into two sub-cases: (1) message m is in nw(s), which is the network of the current state, and (2) m is not in nw(s). CafeOBJ allows us to write proof scores to conduct case splitting by introducing two equations: (i) \(\text{nw}(s) = (m , \text{nw}')\) to characterize (1) and (ii) \(m \not\in \text{nw}(s) = \text{false}\) to characterize (2), where nw' is a constant denoting an arbitrary network (or list of messages). With CiMPA, if we declare equation (i) and apply for case splitting, then it will automatically split the current goal into two sub-goals in which (i) holds in the first sub-goal, while it does not hold in the second one. Thus, the second sub-goal is characterized by the equation \((\text{nw}(s) = (m , \text{nw}')) = \text{false}\). In this sub-goal, it does not guarantee that m is not in \(\text{nw}(s)\) since m can be in \(\text{nw}'\). CiMPG also can not recognize that the use of two equations (i) and (ii) for case splitting is correct. In the existing proof scores of formal verification of NSLPK, there are many times in which case splitting is “flexibly” applied in the same way as based on two equations (i) and (ii) mentioned above. This flexible case splitting is an advantage of the CafeOBJ/proof score method but also is a disadvantage because we need to ensure that the equations used for case splitting cover every case and do not overlap each other. However, to make it possible for CiMPG to generate the proof scripts, the existing proof score needs to be modified. With the example mentioned above, two equations used for case splitting should be \(m \not\in \text{nw}(s) = \text{true}\) and \(m \not\in \text{nw}(s) = \text{false}\).

IV. CONCLUSION

This paper has presented the formal verifications with proof assistant CiMPA and with proof generator CiMPG. In comparison with the proof score approach, each verification method has advantages as well as disadvantages. While proof scores are flexible to write, they are subject to human errors since human users can overlook some cases during the verification. The proof scripts are reliable, but they are not easy to develop, especially with non-expert users. CiMPG combines the flexibility of the proof score approach and the reliability of CiMPA. However, it often takes time for CiMPG to generate proof scripts when the size of input proof scores is large. Two case studies are presented in which we formally verify that IFF protocol enjoys the identifiable property, and NSLPK enjoys the nonce secrecy and one-to-many correspondence properties.

REFERENCES


