# 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 CiMPA was developed for CafeInMaude, the world's second implementation of CafeOBJ. Furthermore, a proof generator called CiMPG was developed to benefit from the strong points of both PSA2FV and CiMPA. Although some case studies have been conducted with CiMPG, 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 in 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 $\quad p \to q : r$
Reply $\quad q \to p : \varepsilon_k(r, q)$

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 $\mathrm{Check}$ message. On receipt of the message, $q$ sends back to $p$ a $\mathrm{Reply}$ message that consists of $r$ and ID $q$ encrypted by the symmetric key $k$ of the $q$'s group. When $p$ receives the $\mathrm{Reply}$ message, $p$ tries to decrypt the ciphertext received with the symmetric key of the $p$'s group. If the decryption succeeds and the plaintext

This work was supported by JST SICORP Grant Number JPMJSC20C2, Japan.

DOI reference number: 10.18293/SEKE2021-037

consists of $r$ and $q$, $p$ then concludes that $q$ is a member of the $p$'s group.

We suppose that the cryptosystem used is perfect, there is only one legitimate group, all members of the group are trustable, and there are also untrustable principals who are not members. Trustable principals exactly follow the protocol, but untrustable ones may do something against the protocol as well. The combination and cooperation of untrustable principals are modeled as the most general enemy (or intruder). The enemy gleans as much information as possible from messages flowing in the network and creates fake 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:

```
op enc : Key Rand Prin -> Cipher .
op k : Cipher -> Key . op r : Cipher -> Rand .
op p : Cipher -> Prin .
```

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 $p$, $\texttt{enc}(k, r, p)$ denotes the ciphertext obtained by encrypting $r$ and $p$ with $k$. Operators `k`, `r` and `p` return the first, second and third arguments of $\texttt{enc}(k, r, p)$, respectively.

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

```
op cm : Prin Prin Prin Rand   -> Msg
op rm : Prin Prin Prin Cipher -> Msg
```

where `Msg` is the sort denoting messages. The first, second and third arguments of each of `cm` and `rm` are the actual creator, the seeming sender and the receiver of the corresponding 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 replay 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 `rands` and `ciphers`, which are declared as follows:

```
op rands   : Network -> ColRands
op ciphers : Network -> ColCiphers
```

where `Network` is the sort denoting networks, `ColRands` is the sort denoting collections of random numbers, and `ColCiphers` is the sort denoting collections of ciphertexts. `ciphers` is defined by the following equations:

```
eq C \in ciphers(void) = false .
ceq C \in ciphers(M , NW) = true if rm?(M)
and C = c(M) .
ceq C \in ciphers(M , NW) = C \in ciphers(NW)
if not(rm?(M) and C = c(M)) .
```

where `void` denotes the empty multiset (or empty network), operator `rm?` checks if a given message is a Reply message, operator `c` takes a Reply message as a parameter and returns its ciphertext (i.e., the fourth argument of `rm` operator), `\in` is an infix operator checking the existence of an element in a collection, and operator `,` in `M , NW` denotes 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`. `rands` can be defined likewise.

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

```
op nw : Sys -> Network . op ur : Sys -> URands
```

where `Sys` is the sort denoting the state space of IFF, `URands` is the sort denoting the sets of random numbers.

We use five transitions together with one constant of `Sys` to represent an arbitrary initial state as follows:

```
op init  : -> Sys {constr}
op sdcm  : Sys Prin Prin Rand -> Sys {constr}
op sdrm  : Sys Prin Msg -> Sys {constr}
op fkcm1 : Sys Prin Prin Rand -> Sys {constr}
op fkrm1 : Sys Prin Prin Cipher -> Sys {constr}
op fkrm2 : Sys Prin Prin Rand -> Sys {constr}
```

`sdcm` and `sdrm` formalize sending Check and Reply messages exactly following the protocol, respectively. The remaining actions `fkcm1`, `fkrm1`, and `fkrm2` are the enemy's faking messages, which can be understood as follows:

- `fkcm1`: a random number `R` is available to the enemy, the enemy fakes and sends a Check message using `R`,
- `fkrm1`: a ciphertext `C` is available to the enemy, the enemy fakes and sends a Reply message using `C`,
- `fkrm2`: a random number `R` is available to the enemy, the enemy fakes and sends a Reply message using `R`.

`sdcm` is defined as follows:

```
ceq nw(sdcm(S,P1,P2,R)) = (cm(P1,P1,P2,R) ,
nw(S)) if c-sdcm(S,P1,P2,R) .
ceq ur(sdcm(S,P1,P2,R)) = (R ur(S))
if c-sdcm(S,P1,P2,R) .
ceq sdcm(S,P1,P2,R) = S
if not c-sdcm(S,P1,P2,R) .
```

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,P1,P2,R)` is put into the network `nw(S)`, `R` is put into `ur(S)` in the state denoted by `sdcm(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:

```
op inv1 : Sys Prin Prin Prin Key Rand
-> Bool .
eq inv1(S,P1,P2,P3,K,R) = ((not(K = k(enemy))
and rm(P1,P2,P3,enc(K,R,P2)) \in nw(S))
implies not(P2 = enemy)) .
```

We describe how to prove that IFF enjoys the property by writing proof scripts and running with CiMPA. In the proof of `inv1`, we need to use a lemma `inv2` that is as follows:

```
op inv2 : Sys Key Rand -> Bool .
eq inv2(S,K,R) = (enc(K,R,enemy) \in
ciphers(nw(S)) implies (K = k(enemy))) .
```

where `k(enemy)` denotes the symmetric key of the group to which the enemy belongs to.

The proof starts with the goals we need to prove:

```
open IFF .
:goal{
eq [iff1 :nonexec] : inv1(S:Sys,P:Prin,
P1:Prin,P0:Prin,K:Key,R:Rand) = true .
eq [iff :nonexec] :
inv2(S:Sys,K:Key,R:Rand) = true . }
```

where `IFF` is the module in which the specification of IFF together with `inv1` 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:

```
: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`, `sdcm`, and `sdrm`, where `si` stands for simultaneous induction. Each goal consists of two equations to prove, corresponding to `inv1` 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 eq [iff1 :nonexec]: inv1(fkcm1(
S#Sys,P#Prin,P0#Prin,R#Rand),P@Prin,
P1@Prin,P0@Prin,K@Key,R@Rand) = true .
1-2. TC 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`, `P1@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:

```
:def c1 = :ctf [R#Rand \in rands(nw(S#Sys)) .]
:apply(c1)
```

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:

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

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

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

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

However, with the sub-goals in which the Boolean term is true, first six equations hold and the last equation does not hold, we need to conduct case splitting more as well as use `inv2` as a lemma:

```
:def c11 = :ctf {eq P1@Prin  = enemy .}
:def c12 = :ctf [enc(K@Key,R@Rand,enemy)
  \in ciphers(nw(S#Sys)) .]
:apply(c11) :apply(c12)
:imp [iff] by {K:Key <- K@Key ;
R:Rand <- R@Rand ;}
:imp [iff1] by {K:Key <- K@Key ;
P0:Prin <- P0@Prin ; P1:Prin <- P1@Prin ;
P:Prin <- P@Prin ; R:Rand <- R@Rand ;}
:apply (rd)
```

The lemma `inv2` is instantiated by replacing the variables `K:Key` and `R:Rand` with the fresh constants `K@Key` and `R@Rand`, and the instance is used as the premise of the implication. The induction hypothesis is instantiated by replacing the variables with the fresh constants, and the instance is used as the premise of the implication. Then, `:apply(rd)` is used to discharge the current goal. The remaining sub-goals of 2 can be discharged directly without using any lemma. The remaining goals from 3 to 6 can be discharged likewise.

### *C. Formal Verification with CiMPG*

The following is the proof score for the case corresponding to goal 1-1-1 in the last section:

```
open 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 inv1(s,a,b,c,k,r1)
implies inv1(fkcm1(s,d,e,r2),a,b,c,k,r1) .
close
```

where `open` makes the module `IFF` available, `close` stops the use of the module and `red` reduces (computes) the given term. `s` and `k` correspond to `S#Sys` and `K@Key` in the last section, respectively. `a`, `b`, `c`, `d`, and `e` correspond to `P@Prin`, `P1@Prin`, `P0@Prin`, `P#Prin`, and `P0#Prin`, respectively. `r1` and `r2` correspond to `R@Rand` and `R#Rand`, respectively. The details of the proof score approach as well as how to write proof scores to conduct formal verification can be found in paper [1]. In comparison with proof scripts, proof scores are often easier to understand for human users, and writing proof scores are also more flexible than writing proof scripts. That is the reason why when conducting formal verification, we prefer to write proof scores rather than proof scripts. However, because of the flexibility, proof scores are subject to human errors. For example, during the verification users may overlook some cases, leading to the flaw verification.

After writing proof scores that IFF protocol enjoys the property, we can confirm that the proof scores are correct by doing the formal verification with CiMPA as described in the last section. Although we are able to conduct the formal verification with CiMPA once we have completed formal verification by writing proof scores in CafeOBJ, it would be preferable to automatically confirm the correctness of proof scores. CiMPG makes it possible to automatically confirm the correctness of proof scores by generating proof scripts for CiMPA from the proof scores.

To use CiMPG, we need to add `:id(iff)` into each open-close proof score fragment. For example, the open-close fragment shown above becomes as follows:

```
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 inv1(s,a,b,c,k,r1)
implies inv1(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 one 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:

```
op enc1 : Prin Nonce Prin -> Cipher1
op enc2 : Prin Nonce Nonce Prin -> Cipher2
op enc3 : Prin Nonce -> Cipher3
```

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 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. enc2 and enc3 can be understood likewise.

We specify three messages used in NSLPK as follows:

```
op m1 : Prin Prin Prin Cipher1 -> Msg
op m2 : Prin Prin Prin Cipher2 -> Msg
op m3 : Prin Prin Prin Cipher3 -> Msg
```

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:

```
op cnonce : Network -> ColNonce
op cenc1 : Network -> ColCipher1
op cenc2 : Network -> ColCipher2
op cenc3 : Network -> ColCipher3
```

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 cenc1 are as follows:

```
eq E1 \in cenc1(void) = false .
ceq E1 \in cenc1(M,NW) = true if m1?(M) and
not(key(cipher1(M)) = intruder) and
E1 = cipher1(M) .
ceq E1 \in cenc1(M,NW) = E1 \in cenc1(NW)
if not(m1?(M) and E1 = cipher1(M))
and not(key(cipher1(M)) = intruder) .
```

where E1 is a CafeOBJ variable of Cipher1. 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 m1 operator). Operator key takes a ciphertext as an argument and returns the principal in which the ciphertext is encrypted with its public key. 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. cnonce, cenc2, and 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:

```
op ur : Sys -> URand . op nw : Sys -> Network
op init : -> Sys {constr}
op sdm1 : Sys Prin Prin Rand -> Sys {constr}
op sdm2 : Sys Prin Rand Msg -> Sys {constr}
op sdm3 : Sys Prin Rand Msg Msg  -> Sys
{constr}
op fkm11 : Sys Prin Prin Cipher1 -> Sys
{constr}
op fkm12 : Sys Prin Prin Nonce -> Sys {constr}
op fkm21 : Sys Prin Prin Cipher2 -> Sys
{constr}
op fkm22 : Sys Prin Prin Nonce Nonce -> Sys
{constr}
op fkm31 : Sys Prin Prin Cipher3 -> Sys
{constr}
op fkm32 : Sys Prin Prin Nonce -> Sys {constr}
```

where URand is the sort denoting sets of random numbers. 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:

- fkm11, fkm21, 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.
- fkm12 and fkm32: a nonce N is available to the intruder, the intruder fakes and sends an Init or Ack message using N, respectively,
- 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 & Q are CafeOBJ variables of Prin. fkm11 is defined as follows:

```
eq  ur(fkm11(S,P,Q,E1)) = ur(S) .
ceq nw(fkm11(S,P,Q,E1)) = m1(intruder,P,Q,E1)
, nw(S) if c-fkm11(S,P,Q,E1) .
ceq fkm11(S,P,Q,E1) = S
if not c-fkm11(S,P,Q,E1) .
```

where c-fkm11(S,P,Q,E1) is E1 \in cenc1(nw(S)), intruder is a constant of Prin denoting the intruder. The equations say that if c-fkm11(S,P,Q,E1) is true, then the Init message m1(intruder,P,Q,E1) is put into the network nw(S), ur(S) does not change in the state denoted by fkm11(S,P,Q,E1); if c-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:

```
eq inv130(S,N) = (N \in cnonce(nw(S))
implies (creator(N) = intruder or
```

```
forwhom(N) = intruder)) .
```

The one-to-many correspondence property is specified by the following two equations:

```
eq inv170(S,P,Q,Q1,R,N) = (not(P = intruder)
and m1(P,P,Q,enc1(Q,n(P,Q,R),P)) \in nw(S)
and m2(Q1,Q,P,enc2(P,n(P,Q,R),N,Q)) \in nw(S)
implies
m2(Q,Q,P,enc2(P,n(P,Q,R),N,Q)) \in nw(S)) .
```

```
eq inv180(S,P,Q,P1,R,N) = (not(Q = intruder)
and m2(Q,Q,P,enc2(P,N,n(Q,P,R),Q)) \in nw(S)
and m3(P1,P,Q,enc3(Q,n(Q,P,R))) \in nw(S)
implies
m3(P,P,Q,enc3(Q,n(Q,P,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 scripts from proof scores. Both of them require the use of the following lemmas:

```
eq inv100(S,E1) = (E1 \in cenc1(nw(S))
implies not(key(E1) = intruder)) .
eq inv110(S,E2) = (E2 \in cenc2(nw(S))
implies not(key(E2) = intruder)) .
eq inv120(S,E3) = (E3 \in cenc3(nw(S))
implies not(key(E3) = intruder)) .
eq inv140(S,E1) = (E1 \in cenc1(nw(S)) and
principal(E1) = intruder
implies nonce(E1) \in cnonce(nw(S))) .
eq inv150(S,E2) = (E2 \in cenc2(nw(S)) and
principal(E2) = intruder
implies nonce2(E2) \in cnonce(nw(S))) .
eq inv160(S,N) = (creator(N) = intruder
implies N \in cnonce(nw(S))) .
```

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) `nw(s) = (m , nw')` to characterize (1) and (ii) `m \in nw(s) = 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 `(nw(s) = (m , nw')) = false`. In this sub-goal, it does not guarantee that `m` is not in `nw(s)` since `m` can be in `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 \in nw(s) = true` and `m \in nw(s) = 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


[1] K. Ogata and K. Futatsugi, "Proof scores in the OTS/CafeOBJ method," in *FMOODS 2003*, 2003, pp. 170–184.

[2] R. Diaconescu and K. Futatsugi, *Cafeobj Report*, ser. AMAST Series in Computing. World Scientific, 1998, vol. 6.

[3] M. Clavel, et al., Ed., *All About Maude*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[4] A. Riesco and K. Ogata, "Prove it! Inferring formal proof scripts from CafeOBJ proof scores," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, pp. 6:1–6:32, 2018.

[5] R. J. Anderson, *Security engineering - A guide to building dependable distributed systems*. Wiley, 2001.

[6] G. Lowe, "An Attack on the Needham-Schroeder Public-Key Authentication Protocol," *Inf. Process. Lett.*, vol. 56, no. 3, pp. 131–133, 1995.

[7] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.