Formal verification of the mERA-based eServices with trusted third party protocol

Maria Christofi[1,2] and Aline Gouget[1]

[1]Gemalto - 6, rue de la Verrerie - 92447 Meudon sur Seine - France
[2]Versailles Saint-Quentin-en-Yvelines University - France
{maria.christofi, aline.gouget}@gemalto.com

Abstract. Internet services such as online banking, social networking and other web services require identification and authentication means. The European Citizen card can be used to provide a privacy-preserving authentication for Internet services enabling e.g. an anonymous age verification or other forms of anonymous attribute verification. The Modular Enhanced Symmetric Role Authentication (mERA) - based eServices with trusted third party protocol is a privacy-preserving protocol based on eID card recently standardized at CEN TC224 WG16. In this paper, we provide a formal analysis of its security by verifying formally several properties, such as secrecy, message authentication, unlinkability, as well as its liveness property. In the course of this verification, we obtain positive results about this protocol. We implement this verification with the ProVerif tool.

Keywords: privacy, authentication, mERA, formal verification, cryptographic protocol, ProVerif

1 Introduction

Cryptographic protocols can be seen as small programs which are designed to guarantee the security of exchanges between participants. Even if some protocols look very simple, the design of a secure cryptographic protocol is often difficult and error-proning. This is the case for many protocols, for which attacks were found many years after their design. It is thus very important to formally analyze the properties fulfilled by these protocols.

A protocol verification procedure can be seen as a procedure with two inputs and one output. The procedure takes as inputs a protocol definition and a statement describing a property of protocol execution and then outputs a “yes” or “not sure” answer. A “yes” answer means that the input property is indeed a property of the input protocol. A “not sure” answer means that some effort to establish this proposition has failed. This means that either the proposition is false and so cannot be established, or it may be true but more efforts are needed in order to establish it.

The verification of security properties of protocols has been the subject of many recent research works. The general idea is to give a description of the protocol in the formal style of Dolev-Yao model [8], a description of security properties such as secrecy and message authentication. Then verifiers, such as ProVerif [7] or AVISPA [5], can be used to nearly automatically check various security properties of protocols.

This paper describes the use of one of these tools (mainly ProVerif) for the verification of the mERA-based eServices with trusted third party protocol. This is a recently standardized protocol (EN 14890) presented in [12]. This protocol is used when a user wants to have access to some services where he has to prove that he fulfills some criteria. The aim is that the user preserves his privacy concerning his identity or personal data while being able to prove the fulfillment of these criteria.

The verification yields assurance about the secrecy, the message authentication and the unlinkability during the protocol. The reasoning could also be used in other related protocols.

Structure of the paper In this paper, we review in Section 2 the description of the mERA-based eServices with trusted third party protocol. In Section 3, we briefly present the verification tool. We explain our formal specification in Section 4 and then prove the security properties in Section 5.

Related work It is quite common to reason informally about security protocols. For instance, in [9] Krawczyk define the protocol SKEME and he gave some informal arguments about the properties SKEME. Generally, such arguments are informative, but far from being complete and fully reliable. Formal proofs could be necessary according to the critical impacts of security flows. Nevertheless, formal proofs for concrete, practical protocols remain relatively rare. We state some of these results. The work of Abadi and Blanchet in [1] reports on the formal specification of a non trivial protocol for certified email, as well as on the verification of its main security properties. In [2], Abadi, Blanchet and Fournet combine manual and automated proofs for analyzing the Just Fast Keying (JFK) protocol which was one of the candidates to replace IKE as the key exchange protocol in IPSec. They verify classical properties such as secrecy and authenticity, but also properties like forward secrecy and identity protection. Both [10] and [6] formalize and analyze privacy properties for electronic voting. Indeed, in [10] Kremer and Ryan provide the first definitions in the symbolic model of vote-privacy, receipt-freeness and coercion-resistance. However, this work does not consider forced-abstention attacks and do not apply to remote voting protocols. These two last properties have been studied by Backes et al. in[6].

In this paper, we use ProVerif to formally verify some properties of the privacy-preserving authentication protocol called mERA-based eServices with trusted third party protocol.

2 mERA-based eServices with trusted third party protocol

This section recalls the description of the mERA-based eServices with trusted third party protocol (mERA stands for "modular Enhanced Symmetric Role Authentication"). The reader can also see the original description in [4] for additional details.

The participants of this protocol are: a smart card (denoted by ICC for Integrated Circuit Card) with its owner and a local reader, an identity provider (denoted by IdP), a service provider (denoted by SP) and a certification authority (denoted by CA).

The aim of the mERA-based eServices with trusted third party protocol is that a user could remotely use a privacy-preserving credential without disclosing his identity or personal data to the service provider, while proving he fulfills the profile access criteria and without disclosing any knowledge about the service to the identity provider. The privacy-preserving credential is issued by an identity provider to the card. Indeed, a user who has a card and wants to access a service from a service provider has to prove first that his profile fulfills different criteria required by the service provider. A card can get a privacy-preserving credential from an identity provider by proving the compliance of its profile stored in the card with the profile required by the service provider while preserving the user privacy. The identity provider learns nothing about the service delivered by the service provider, except the criteria required by the service provider to access the service. The information delivered to the service provider by the card, which is included in the privacy-preserving credential, is under the control of the user.

2.1 Cryptographic primitives

The protocol relies on a number of cryptographic primitives. The corresponding notations are as follows:

- an encryption function (the encryption of m using the key k) : $\{m\}_k$
- a decryption function: dec
- a signature: $sign$
- a Message Authentication Code function: MAC
- a key derivation function : KDF
- two secure hash functions: H and H_{dh}

For this protocol, we use also the concatenation. The concatenation of the m_i is denoted by: $m_1 \mid m_2 \mid ... \mid m_n$.

2.2 Description of the protocol

The mERA protocol is a modular authentication protocol which means that it can be used in two different ways called mERA1-3 and mERA1-7. The mERA-based eServices with trusted third party protocol involves three parties: a smart card (ICC), a service provider (SP) and an identity provider (IdP). The goal of this protocol is to grant user access to some eServices while preserving user-privacy.

A smart card owns a master secret key which is shared by a large number of cards for user-privacy purpose, i.e. in order to prevent the identification by a service provider of a specific card within the group of cards sharing the same master key. Every service provider owns a specific secret key which can also be computed on the card side from both the master key and the service provider's identifier. In addition, every service provider is equipped with a long term Diffie-Hellman key pair and a pseudo-certificate in order to prevent cards to impersonate a service provider.

In this protocol, we can distinguish three main phases:

- **Phase 1 (mERA1-3):** Exchanges between a smart card and a service provider. The card first requests to the service provider a set of criteria conditioning access to a service. It then authenticates the service provider while remaining anonymous. The authentication in this phase is performed using the mERA1-3 protocol.
- **Phase 2 (privacy-preserving mutual authentication):** a mutual authentication happens between the identity provider and the card. The protocol used for the mutual authentication during this phase is not part of the specification of the mERA-based eServices protocol. For the purpose of our verification we used the "device authentication with privacy protection" protocol described in [12]. At the end of this phase, the identity provider delivers a privacy preserving credential.
- **Phase 3 (mERA1-7):** Exchanges between the card and a service provider. During this phase, the card requests access to the service provider. There is a mutual authentication using the mERA1-7 protocol and the card securely transmits the privacy preserving credential to the service provider.

These exchanges are detailed in Figure 1.

Key setup The following setup of the system concerns only phases 1 and 3. For the second phase, we consider that it is performed using the "device authentication with privacy protection" protocol [12].

- A master key mk is generated by a probabilistic algorithm and transmitted to any legitimate card (ICC).
- A new identifier id_i is generated for the service provider (SP) and the corresponding sk_i is computed : $sk_i = KDF(mk, id_i)$.
- A private Diffie-Hellman element x_i, a public Diffie-Hellman element g^{x_i} and a certificate σ_i are generated and securely transmitted to SP. Then, SP securely receives its personal secret values (sk_i, x_i) and its personal public values $(id_i, g^{x_i}, \sigma_i)$.

Phase 1 Get the Profile

1. The user first requests the service provider to get the profile (access conditions associated to the service) and he checks if it is acceptable. Afterwards, the user authenticates himself via PIN presentation to notify his content; this step is not included in our formal verification.
2. The service provider is authenticated by the card in order to prove to the card that it is authorized to store a profile in the card.
3. The card generates an ephemeral key pair (u, g^u) serving to link the profile to the service provider and to securely connect later to this service provider.

During this phase, the profile is transmitted encrypted and the ciphertext is transmitted with a Message Authentication Code (MAC)

Phase 2 Getting the privacy-preserving credentials with mutual authentication

4. A mutual authentication takes place between the identity provider and the card, and a secure channel is established protecting all following commands.
5. The identity provider may verify the revocation status of the card.
6. The identity provider recovers the public part of the ephemeral key g^u and the profile. It asks for the user's consent and makes the necessary verifications about the questions and answers.
7. The identity provider delivers to the card a privacy preserving credential including the public part of the ephemeral key g^u and the profile, and signed with its private key.

During this phase, the profile, the public part of the ephemeral key g^u and the privacy-preserving credential are transmitted in a secure channel. The user also checks the profile.

Phase 3 Use of the privacy-preserving credentials

8. A mutual authentication takes place between the card and the service provider.
9. At the end of the authentication, the card and the service provider share a common DH-based ephemeral secret, based on their public parameters exchanged during the mutual authentication. A secure channel is established between the card and the service provider. The card sends the privacy-preserving credential to the service provider, within the secure channel.
10. The service provider verifies the elements of the privacy preserving credential and that this credential has been signed by a recognized identity provider. If the verification is successful, the service provider knows that the card is compatible with its use condition, and so the service provider grants access to the service.

3 The formal verification tool: ProVerif

In this section, we review the verification tool that we use for our analysis.

The verification tool ProVerif (Protocol Verifier) [7] enables to perform a static analysis for cryptographic protocols under the assumption that the cryptographic primitives are idealized. The tool requires expressing protocols and the properties that we want to prove in a formal language. The tool is sound with respect to the semantics of this language. The formal verification using this tool guarantees the absence of the attacks as captured by the language semantics, but not necessarily of other attacks.

3.1 Proof engine

The proof-engine uses a resolution-based solving algorithm in order to determine properties of the protocol. Actually the verifier has to describe the protocol as well as the properties that (s)he wants to prove in a formal language. The tool proposes two formal languages for the protocol description; both languages are equivalents and are presented later on. ProVerif translates both languages into a set of Horn clauses before using a resolution-based algorithm (see [7] for more details).

In case the property we want to verify is not true, the output of ProVerif is a *trace* explaining the failure. For example, if we request the verification of the secrecy of a variable whereas this variable does not remain secret, the trace explaining the failure would be an attack path leading to reach this variable. In most cases, the output of ProVerif is either a confirmation that the property checked is verified, or an attack as a counterexample to robust safety. In rare cases that ProVerif does not terminate we cannot say anything.

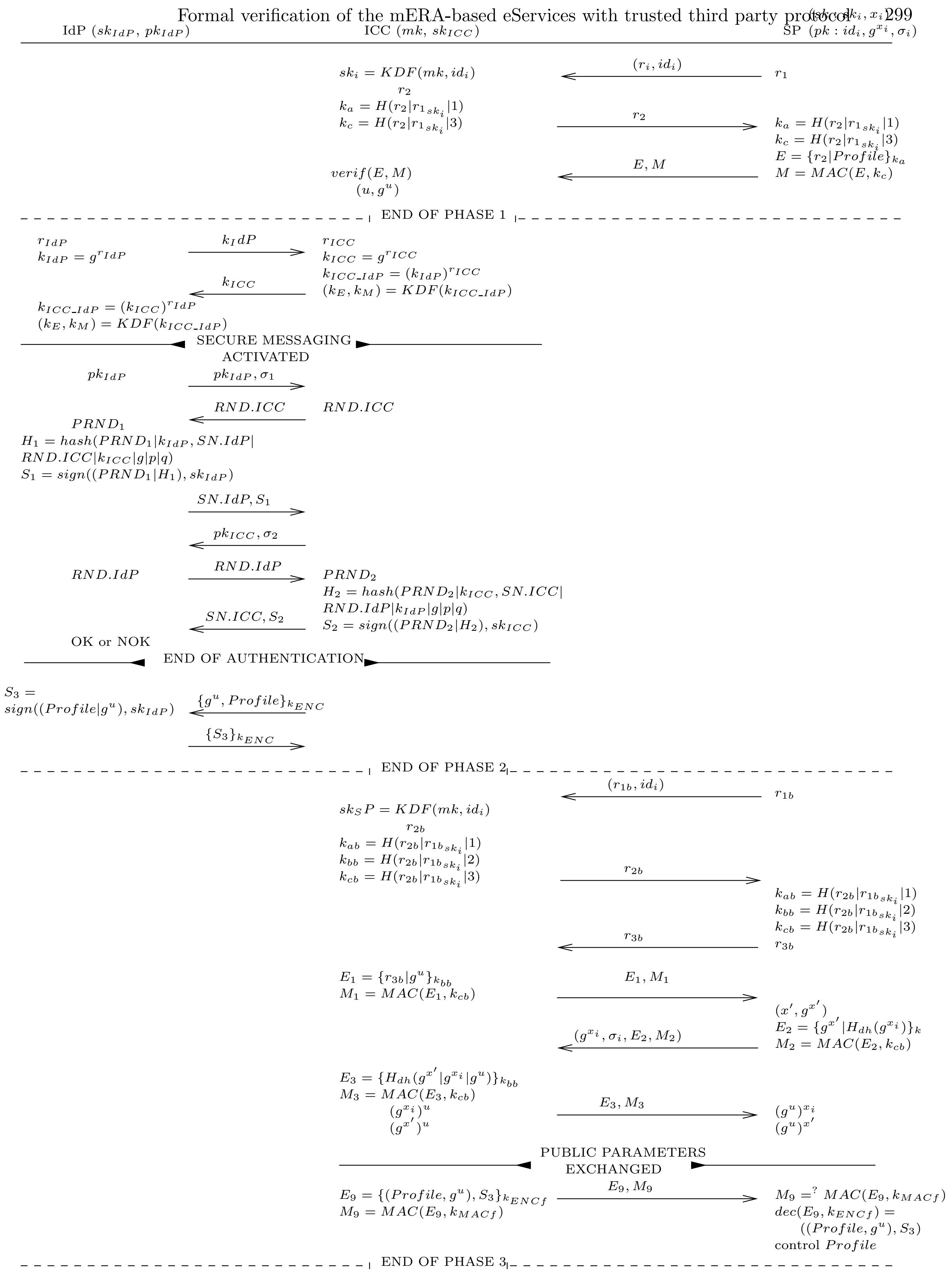
IdP (sk_{IdP}, pk_{IdP})
ICC (mk, sk_{ICC})
(sk_i, x_i)
SP ($pk : id_i, g^{x_i}, \sigma_i$)
(r_i, id_i)
r_1
$sk_i = KDF(mk, id_i)$
r_2
$k_a = H(r_2|r_{1_{sk_i}}|1)$
$k_c = H(r_2|r_{1_{sk_i}}|3)$
r_2
$k_a = H(r_2|r_{1_{sk_i}}|1)$
$k_c = H(r_2|r_{1_{sk_i}}|3)$
$E = \{r_2|Profile\}_{k_a}$
$M = MAC(E, k_c)$
E, M
$verif(E, M)$
(u, g^u)
END OF PHASE 1
r_{IdP}
$k_{IdP} = g^{r_{IdP}}$
$k_I dP$
r_{ICC}
$k_{ICC} = g^{r_{ICC}}$
$k_{ICC\_IdP} = (k_{IdP})^{r_{ICC}}$
$(k_E, k_M) = KDF(k_{ICC\_IdP})$
k_{ICC}
$k_{ICC\_IdP} = (k_{ICC})^{r_{IdP}}$
$(k_E, k_M) = KDF(k_{ICC\_IdP})$
SECURE MESSAGING ACTIVATED
pk_{IdP}
pk_{IdP}, σ_1
$RND.ICC$
$RND.ICC$
$PRND_1$
$H_1 = hash(PRND_1|k_{IdP}, SN.IdP|$
$RND.ICC|k_{ICC}|g|p|q)$
$S_1 = sign((PRND_1|H_1), sk_{IdP})$
$SN.IdP, S_1$
pk_{ICC}, σ_2
$RND.IdP$
$RND.IdP$
$PRND_2$
$H_2 = hash(PRND_2|k_{ICC}, SN.ICC|$
$RND.IdP|k_{IdP}|g|p|q)$
$SN.ICC, S_2$
$S_2 = sign((PRND_2|H_2), sk_{ICC})$
OK or NOK
END OF AUTHENTICATION
$S_3 =$
$sign((Profile|g^u), sk_{IdP})$
$\{g^u, Profile\}_{k_{ENC}}$
$\{S_3\}_{k_{ENC}}$
END OF PHASE 2
(r_{1b}, id_i)
r_{1b}
$sk_SP = KDF(mk, id_i)$
r_{2b}
$k_{ab} = H(r_{2b}|r_{1b_{sk_i}}|1)$
$k_{bb} = H(r_{2b}|r_{1b_{sk_i}}|2)$
$k_{cb} = H(r_{2b}|r_{1b_{sk_i}}|3)$
r_{2b}
$k_{ab} = H(r_{2b}|r_{1b_{sk_i}}|1)$
$k_{bb} = H(r_{2b}|r_{1b_{sk_i}}|2)$
$k_{cb} = H(r_{2b}|r_{1b_{sk_i}}|3)$
r_{3b}
r_{3b}
$E_1 = \{r_{3b}|g^u\}_{k_{bb}}$
$M_1 = MAC(E_1, k_{cb})$
E_1, M_1
$(x', g^{x'})$
$E_2 = \{g^{x'}|H_{dh}(g^{x_i})\}_k$
$M_2 = MAC(E_2, k_{cb})$
$(g^{x_i}, \sigma_i, E_2, M_2)$
$E_3 = \{H_{dh}(g^{x'}|g^{x_i}|g^u)\}_{k_{bb}}$
$M_3 = MAC(E_3, k_{cb})$
$(g^{x_i})^u$
$(g^{x'})^u$
E_3, M_3
$(g^u)^{x_i}$
$(g^u)^{x'}$
PUBLIC PARAMETERS EXCHANGED
$E_9 = \{(Profile, g^u), S_3\}_{k_{ENCf}}$
$M_9 = MAC(E_9, k_{MACf})$
E_9, M_9
$M_9 =^? MAC(E_9, k_{MACf})$
$dec(E_9, k_{ENCf}) =$
$((Profile, g^u), S_3)$
control $Profile$
END OF PHASE 3

Fig. 1. mERA-based eServices with trusted third party protocol

3.2 The language

As seen in Section 3.1 there are two options for the input language. The first one is an abstract representation of the protocol by a set of Horn clauses:

- predicate logic formulas of the form :
  $(\forall \boldsymbol{x})(P_1(\boldsymbol{p_1}) \wedge ... \wedge P_n(\boldsymbol{p_n}) => Q(\boldsymbol{q}))$, where $n \geq 0$ and $P_1, ..., P_n, Q$ are predicate symbols from a fixed set and with fixed arity
- the clauses represent deduction rules for protocol participants and adversary
- one special clause, the goal, represents a property
- ProVerif checks, using resolution, whether {goal, protocol, clauses} is a satisfiable set

The second option is the description of the protocol in a programming language which corresponds to an extension of the *pi calculus* (named applied pi-calculus) introduced by M. Abadi and C. Fournet in [3]. This calculus represents messages by terms M, N, ... and programs by processes P, Q, ... Identifiers are partitioned into names, variables, constructors and destructors.

A function symbol with arity 0 is a constant symbol. Functions are distinguished in two categories: constructors and destructors. Constructors serve for building terms. Thus, the terms are variables, names, and constructors applications of the form $f(M_1, ..., M_n)$. A constructor of arity n is introduced with the declaration *fun* f/n. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. Typical examples of constructors are encryption and pairings. As for the destructors, we can consider decryption and projections. More generally, we can represent data structures and cryptographic operations using constructors and destructors (as we will see below in our coding of this protocol).

The grammar for processes of the ProVerif process language is the following:

P, Q, R ::=		plain processes
	0	null process
	P | Q	parallel composition
	!P	replication
	new n;P	name restriction
	if M = N then P else Q	conditional
	let x = g $(M_1, ..., M_n)$ in P else Q	destructor application
	event M [;P]	events
	in (M,x); P	message input
	out(M,N); P	message output
	phase(n) [;P]	phase indication

The null process does nothing. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction *new* $a; P$ creates a new name a, then executes P. The conditional is actually defined in terms of *let*. A destructor *equals* is defined, with the reduction $equals(x, x) \rightarrow (x)$. The *if* construct stands for *let x = equals (M,N) in P else Q*. We also use some syntactic sugar to allow the let-construction to introduce abbreviations such as *let x = M in P*.

The process calculus includes auxiliary events that are useful in specifying security properties. A query of the form *query ev: M ⇒ ev: N*, allows to express and to verify a property of the form "if the event M has been executed, then the event N must have been executed before". For example, authentication between A and B can be seen as a query of the form "if B receives a message m, then A must have sent it before" and so it can be proved using the events.

Secrecy assumptions correspond to an optimization of our verifier. The declaration *not M* indicates to the verifier that M is secret. The verifier can then use this information to speed up the solving process. At the end of the process, the verifier checks whether the secrecy assumption is true, so that a wrong secrecy assumption is leading to an error message but not to an incorrect result.

The input process *in (M, x);P* inputs a message on channel M, then runs P with the variable x bound to the input message. The output process *out (M, N);P* outputs the message N on the channel M, then runs P.

The phase separation command *phase n; P* indicates the beginning of phase n. When a process starts running it is in *phase 0.* When a process enters the next phase, all remaining instructions from the previous phase are discarded. In that sense, one can understand the phase separators as global synchronization commands. The adversary obviously keeps its knowledge when changing phases.

We generally assume that processes execute in the presence of an adversary, which is itself a process in the same calculus and so it is allowed to execute the same actions as any other process. The adversary needs not to be programmed explicitly; we usually establish results with respect to all adversaries.

4 Verifying mERA protocol

In order to analyze the mERA protocol (see Figure 1 for an informal description), we program it using the input language described in section 3.2. We give in Appendix A, an extract of the code which describes the exchanges that SP participates under the name of SP process.

4.1 Modelisation of cryptographic primitives

In the code, we firstly declare some cryptographic primitives. For example,

∗ the constructor *encrypt* represents the encryption function, which takes two parameters, a public key and a message, and returns a ciphertext
∗ the destructor *decrypt* represents the corresponding decryption function. From a ciphertext encrypt(x, y) and the corresponding secret key y, it returns the ciphertext x. Hence we give the rule: *decrypt(encrypt (x, y), y) = x.* We assume perfect cryptography, so the cleartext can be obtained from the ciphertext only when one has the decryption key
∗ the constructor *sign* representing a signature scheme, which takes two parameters, a secret key and a message, and returns a signature
∗ the destructor *checksign*, checks the signature, while *getmess* returns the message without checking the signature. (In particular, the adversary may use *getmess* in order to obtain message contents from signed messages). *checksign* takes two parameters, the signature and the corresponding public key, and returns the message only if its signature is valid, whereas *getmess* has only one parameter, the signature, and returns the signed message without checking its signature.
∗ the constructor MAC for the MAC function
∗ the constructor *KDF* for the key derivation function
∗ two constructors *hash* and $hash_{dh}$ for the two hash functions that we need
∗ the constructor *pk* in order to compose the public keys for the authentication from the secret keys, as well as to compose id_i from the sk_i
∗ the constructor g used for the exponentiation of the generator of the group to a power

We analogously define the Diffie-Hellman function. Concatenation is represented by tuples, which are pre-defined by default in the tool with all the necessary operations.

4.2 Declarations and channels

In ProVerif we can declare two types of variables. The *free* type variable which declares public free names and the *private free* variables which declares private free names (not known by the adversary).

Communication channels are often public free variables. So we declare them using the "free" declaration. For this protocol, we need three channels:

- `c` which is a public channel used for exchanges between the card and the service provider during the first and the third phase
- `auth` which is a second public channel used only between the card and the identity provider during the second phase of the protocol and
- `cert` which is a private channel used in order to communicate to a certification authority whenever we need to verify a certificate using the public keys of the corresponding entities. The certificates used are σ_1 -which is the identity provider's certificate- , σ_2 -which is the card's certificate- and σ_i -which is the service prodiver's certificate- .

4.3 Processes

Until now, we have described the declaration of the cryptographic primitives and the different variables that we need for this protocol. So we are ready to describe the protocol itself.

Every entity of the protocol (in this case: ICC, IdP and SP) represents a process that describes every action of this entity using the language described before. For example (see the code in Appendix A), the process SP first generates a random number and then sends it as well as the SP identifier (noted pkSP) on the public channel c. Then it executes the steps of the protocol description.

The process *let* $x = g(M_1, ..., M_n)$ *in* P *else* Q tries to evaluate $g(M_1, ..., M_n)$; if this succeeds then x is bound to the result and P is run, else Q is run. A pattern $=M$ matches only the term M. So, on the example of Appendix A, we can find the pattern *let (=r3, pkDHICCr) = decrypt(enc1, kBb) in [...]* . This tries to evaluate *decrypt(enc1, kBb)*; if this succeeds, then *r3* is bound to the result *decrypt(enc1, kBb)* and the rest of the code is run. So the destructor *decrypt* above succeeds if and only if $decrypt(enc1, kBb) = r3$.

In addition of the three processes corresponding to the three entities, we define two processes `lookup_cert` and `lookup_certSP` to verify the certificates.

All these processes are composed in the last part of the protocol specification (see Appendix B). This last part computes SP's, ICC's and IdP's encryption keys from their secret keys by the constructor `pk`. For example: pkSP = pk(skSP) corresponds to the computation of public identifiers called here id_i using the constructor pk and the sk_i (pkSP := id_i and skSP := sk_i). Then the corresponding public keys are output in the public channel c, so that the adversary can have them. We proceed similarly for both the authentication keys (of ICC and IdP), as well as for SP's Diffie-Helmann pair of keys (skDHSP, pkDHSP) but using the constructor `g` this time.

5 Properties formally verified

In this section, we present the properties that we have formally verified.

5.1 Liveness property

The liveness property of a protocol consists in proving that the protocol is not in a permanent deadlock situation. That means that at least an exchange will occur during its life time.

In this case, we have successfully verified the protocol completion. It is verified that both an exchange between the card and the service provider and an exchange between the card and the identity provider will occur at some time in the life of the card.

This is done with the help of a *query* which verifies the existence of the events "ICC accepts SIG_3" (sent by IdP) and "SP accepts the signature SIG_3" (sent by the card). The first event verifies that at least one exchange between IdP and ICC will occur, and the second one that at least one exchange between ICC and SP will occur. So the verifier has to prove the existence of both events in order to ensure us that at least one exchange will occur between every entity who participates in the protocol.

5.2 Security properties

Secrecy The secrecy is then a correctness property which must be an invariant of the protocol. The property must be true at the beginning of the protocol execution and must be true at all instances of the protocol execution. In ProVerif, this is done with the appropriate query *attacker "variable"*.

In particular,

* the secret keys of all the parties of the protocol, that is: sk_i, x_i, sk_{ICC_AUTH}, sk_{IdP_AUTH}, remain secrets all over the protocol
* the master key mk remains secret all over the protocol (if not the attacker could recover the sk_i and so run all the protocol instead of a legitimate card)
* the secret part of the ephemeral DH keys (generated by both ICC -named u- during the first phase and SP -named x'- during the third phase of the protocol) remain secret throughout all the exchanges occurred in the protocol
* the keys chosen randomly by both IdP and ICC during the DH of the second phase of the protocol (that is r_{IdP} and r_{ICC}) remain secrets all over the protocol
* the common key between ICC and IdP used in order to build the keys of encryption and MAC during the second phase of the protocol ($g^{r_{ICC} \cdot r_{IdP}}$) remains secret all over the protocol
* the profile of ICC remains secret to the attacker throughout the second phase of the protocol
* the keys shared between SP and ICC used at the end of the protocol in order to build the final keys of encryption and MAC (i.e. $g^{x_i \cdot u}$ and $g^{x' \cdot u}$) remain secrets throughout the protocol
* even the public part of the DH key generated by ICC remains secret throughout the protocol and finally
* the unique identifiers (e.g. unique public key, unique serial number) of both ICC and IdP remain secrets throughout the protocol

Message authentication Authentication is used to verify that a party is indeed who (s)he claims to be.

The message authentication is required in different points of the protocol (i.e. verify that if an information is received by an entity, then only the corresponding entity could have sent it). More precisely:

* the authentication of the Profile sent by ICC to SP during the first phase
* the authentication of the public keys of both ICC and IdP and their certificates as well as the authentication of the signatures exchanged during the mutual authentication of the second phase of the protocol
* the authentication of the public part of the ephemeral key of ICC (g^u) and the Profile sent by ICC to IdP, as well as the one of the credential sent by IdP to ICC (during the second phase of the protocol)
* the DH-message authentication during the third phase of the protocol, i.e. if SP accepts a valid public part of an ephemeral key, then only the ICC that has the secret part of the key can have sent it and respectively for the public part of the SP's ephemeral key
* during the third phase, we also verify the validity of the SP's DH key and its certificate
* the DH-message authentication of the three public keys : g^u, $g^{x'}$ and g^{x_i}, at the end of the third phase of the protocol; this is verified through the establishment of a secure channel by using k_{ENCf} and k_{MACf}
* the authentication of the final signature sent by ICC to SP during the third phase of the protocol and which includes the public part of the ephemeral key of ICC (g^u) and the Profile

All these properties are formalized and verified using *queries* about *events*, as described in Section 3. For example, the query:

$$\text{query evinj: ICCaccepts(t)} \Longrightarrow \text{evinj: SPsends(t).}$$

is verified during the first phase of the protocol and check the authentication of the profile, that is if the event “ICC receives a profile” occurs, then the event “this profile is the one sent by SP” has occured before.

Unlinkability The notion of the unlinkability [11] of two or more items of interest means that within the system (including these and possibly other items), from the attackers perspective, these items of interest are no more and no less related after his observation than before according to his initial knowledge.

In ProVerif, this property can be proved either using the *choice* construction (which helps us prove an observational equivalence between two processes) or the *noninterf* query. Using the choice-construction, we can express two processes in one. For example, if we want to verify the unlinkability about the message, we can introduce two messages m_1 and m_2 and using the *choice[*m_1, m_2*]* ask ProVerif if m_1 and m_2 are equivalents from an observer point of view. If it is the case, this means that either we use m_1 or m_2, an observer cannot see any difference, which is what we want to prove in the case of the unlinkability. On the other hand, “noninterf x_1, x_2,...” is supposed to check if a process preserves secrecy of variables $x_1, x_2, ...$ in the strong sense. Strong secrecy means that an adversary cannot see any difference when the value of the secret changes, which is exactly what we are looking to prove for the unlinkability. The choice between the *choice*-construction and the *noninterf*-command is up to the user according to the elements for which (s)he wants to prove the unlinkability and the complexity of the protocol.

In the case of mERA, we verified the unlinkability for two different executions of the phase 1, phase 2 and phase 3, the unlinkability between the phase 1 and the phase 2, between the phases 1 and 3 and between the phases 2 and 3. For example, for the case of the unlinkability between the phases 1 and 3, we had to prove it for the profile as well as the Diffie-Hellman key produced at the end of the phase 1 - u, g^u-.

6 Conclusion

In this paper, we present the formal verification of the mERA-based eServices with trusted third party protocol, as well as the verification of its main security properties, that is secrecy, authentication and unlinkability. This is done using an automatic protocol verifier. The aim of using an automatic verifier is to reduce the human interaction in a so complicated proof and so reduce the risk of error.

In our case, we used ProVerif. An automatic verifier which, unlike other verifiers, does not limit the number of concurrent protocol runs that a modeled attacker may execute. So, it is able to find attacks that require any number of concurrent protocol runs. ProVerif is completely automatic, in the sense that after starting the verifier, it does not need any user input to complete its task. However, a user has to completely understand and model the whole protocol as well as describe the properties that (s)he wants to prove. This step is often the most difficult part of the verification, mainly for complicated protocols like mERA.

Acknowledgments

Maria Christofi’s research is a PhD research work partly funded by ANRT (CIFRE).

References

1. Martín Abadi and Bruno Blanchet. Computer-assisted verification of a protocol for certified email. *Sci. Comput. Program.*, 58(1-2):3–27, 2005.
2. Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just fast keying in the pi calculus. *ACM Trans. Inf. Syst. Secur.*, 10(3), 2007.

3. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.
4. ANTS, Gemalto, Oberthur Technologies, and Safran Morpho. Access to e-services with privacypreserving credentials. Technical Report `http://www.ants.interieur.gouv.fr/IMG/pdf/IAS/TR-Privacy\_Preserving\_Credentials\_10082011\_v2.0.pdf`, 10.08.2011.
5. Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.
6. Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 195–209. IEEE Computer Society, 2008.
7. Bruno Blanchet. From secrecy to authenticity in security protocols. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359. Springer, 2002.
8. Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.
9. H. Krawczyk. SKEME: a versatile secure key exchange mechanism for Internet. *Network and Distributed System Security, Symposium on*, 0:114, 1996.
10. Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2005.
11. Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology. In *Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2001.
12. CEN/TC224 prEN 14890-1:2008. Application interface for smart cards used as secure signature creation devices, 2008.

A Part of the proof: the SP process

```
let SP =    new r1;
   out (c, (r1, pkSP));
   in (c, NX);
        let zz = encrypt ((NX,r1), skSP) in
  let kA = hash((zz,1)) in
  let kC = hash((zz,3)) in
  let sh = (NX, Profile) in
   event SPsends(Profile);
      let E = encrypt(sh, kA) in
   out (c, (E, MAC (E, kC)));
phase 1; (* empty because SP doesn't participate in phase 1 *)
phase 2;
      new r1b;
   out(c, (r1b, pkSP));
   in(c, NXb);
  let zzb = encrypt((NXb,r1b), skSP) in
   let kAb = hash ((zzb,1)) in
   let kBb = hash ((zzb,2)) in
   let kCb = hash ((zzb,3)) in
   new r3;
   out(c, r3);
   in (c, (enc1, mac1) );
  if mac1 = MAC( enc1, kCb) then
  let ( =r3, pkDHICCr) = decrypt (enc1, kBb) in
```

```
      event SPaccepts(pkDHICCr);
    new x1; (* x' *)
    let gx1 = g(x1) in
      event SPsendsDH(gx1);
           let ENC2 = encrypt ( (gx1, hash_dh(pkDHSP)), kAb) in
      out( c, (pkDHSP, sigmaSP, ENC2, MAC ( ENC2, kCb)));
      in ( c, (enc3, mac3));
    if mac3 = MAC( enc3, kCb) then
    if hash4 ( gx1, pkDHSP, pkDHICCr) = decrypt (enc1, kBb) then
       out (c, ok);
            let k1 = f (skDHSP, pkDHICCr ) in
    let k2 = f (x1, pkDHICCr) in
      let ( kENCf, kMACf) = KDF3 (k1 k2) in
in (c, (enc9, mac9));
    if mac9 = MAC( enc9, kMACf) then
    let (Profile, =pkDHICCr, SIG3) = decrypt ( enc9, kENCf) in
    if (Profile, pkDHICCr ) = checksign ( SIG3, pkIdP) then
     event SignAccepts(SIG3).
```

B Part of the proof: the composition of all processes

```
process
new skSP;
    let pkSP = pk(skSP) in out (c, pkSP);
new skICC;
new skDHSP;
    let pkDHSP = g(skDHSP) in out (c, pkDHSP );
new skDHICC;
new SN_ICC;
new skICC_AUTH;
    let pkICC_AUTH = pk (skICC_AUTH) in out (c, pkICC_AUTH);
new SN_IdP;
new skIdP_AUTH;
    let pkIdP_AUTH = pk (skIdP_AUTH) in out (c, pkIdP_AUTH);
new skIdP;
    let pkIdP = pk (skIdP)in out (auth, pkIdP);
( !lookup_cert
| !lookup_certSP
| !SP
| !ICC
| !IdP )
```