An example of proving UC-realization with formal methods

Suzana Andova, Kristian Gjøsteen, Lillian Kråkmo, Stig Frode Mjølsnes, Saša Radomirovi

263

An example of proving UC-realization with formal methods (extended abstract)

S. Andova[1], K. Gjøsteen[1], L. Kråkmo[2], S. F. Mjølsnes[1], and S. Radomirović[1]

[1] Department of Telematics, Norwegian University of Science, Trondheim, Norway
[2] Department of Mathematical Sciences, Norwegian University of Science, Trondheim, Norway

Abstract. In the universal composability framework we consider ideal functionalities for secure messaging and signcryption. Using traditional formal methods techniques we show that the secure messaging functionality can be UC-realized by a hybrid protocol that uses the signcryption functionality and a public key infrastructure functionality. We also discuss that the signcryption functionality can be UC-realized by a secure signcryption scheme.

1 Introduction

Composing several (possibly identical) protocols into a large protocol, in general, may not preserve security. *Universally composable security* is a framework proposed by Canetti [1] as a way to define security for protocols such that security-preserving composition is possible. This allows for a modular design and analysis of protocols. For each cryptographic task, an *ideal functionality* can be defined, which incorporates, the required properties of a protocol for that task and the allowed actions of an adversary. A protocol *securely realizes* the ideal functionality if any effect caused by an adversary attacking the protocol can be obtained by an adversary attacking the ideal functionality. If, in addition, the environment and the adversary are allowed to interact and exchange information freely, it is said that the protocol *UC-realizes* the ideal functionality. The universal composition theorem guarantees that in a complex protocol in which the involved parties have a secure access to the ideal functionality, it is secure to replace the ideal functionality by a protocol that UC-realizes the functionality. We refer to [1] for a complete overview of this framework.

Signcryption is a cryptographic primitive for achieving both, confidentiality and authenticity of message delivery in a logically single step. This cryptographic primitive was first introduced by Zheng in [6] with the aim to reduce the cost compared to the standard "sign-then-encrypt" approach. Since then, several models have been proposed. A good overview of the different models is provided in [5].

In this paper we consider two primitives, *secure messaging* and *signcryption*, in the universal composability framework. The paper is an extension of [3]. In

[3] the authors define ideal functionalities for these two primitives and show that signcryption can be UC-realized by a secure signcryption scheme, and that the secure messaging can be realized by means of signcryption and public key infrastructure functionality. Unfortunately, the latter proof is not very convincing.

The goal of this paper is to show that using *formal methods* we can give a convincing proof for UC-realization of secure messaging as defined in [3]. As a matter of fact, formal methods are the "right" tool for this type of problems, since they allow a completely rigorous protocol specification. Then, it is easy to reason (formally) about interaction and communication of processes running in parallel, and to determine whether two behaviours are the same.

2 Secure messaging and signcryption

We will write $\mathcal{F}_{SM}$, $\mathcal{F}_{SC}$, and $\mathcal{F}_{PKI}$ for the ideal functionalities for secure messaging, signcryption, and public key infrastructure, respectively, and π_{SM} for the signcryption protocol. The ideal functionalities and the protocol are described in Figs. 2–5 in the appendix.

In general, while a functionality is processing a message from party $\tilde{P}_i$, no further messages from $\tilde{P}_i$ will be accepted by the functionality until the current processing is done. Messages from other parties will be accepted and processed, however.

Under a static corruption model, the functionality $\mathcal{F}_{SC}$ can be securely realized by a secure signcryption scheme. For a precise formulation and more detail we refer to [3]. In order to save space, the functionalities described here do not consider corruption. We also note that the signcryption functionality does not allow signatures on messages to be derived from the ciphertexts (not every signcryption scheme has this property).

The rest of the paper is devoted to the following theorem.

Theorem 1. *The protocol π_{SM} securely realizes $\mathcal{F}_{SM}$ in the $(\mathcal{F}_{PKI}, \mathcal{F}_{SC})$-hybrid model.*

Proof. We need to show that for every adversary $\mathcal{A}$, interacting with parties running π_{SM} in the $(\mathcal{F}_{PKI}, \mathcal{F}_{SC})$-hybrid model, there is an ideal adversary $\mathcal{S}$ such that no environment can tell whether it is interacting with $\mathcal{A}$ and π_{SM}, or with $\mathcal{S}$ and the ideal protocol for $\mathcal{F}_{SM}$. For a given adversary $\mathcal{A}$ the ideal adversary $\mathcal{S}$ is given in Fig. 6. It runs a simulated copy of $\mathcal{A}$ and forwards all messages for the environment to $\mathcal{A}$ and back using two interfaces $\mathcal{A}_{PKI}$ and $\mathcal{A}_{SC}$. The correctness of the theorem follows from Lemma 1 on page 49. □

3 Using formal methods to prove $\mathcal{F}_{SM}$ realization

In order to complete the proof of Theorem 1, we represent all *processes* (behaviours), the functionalities $\mathcal{F}_{SM}$, $\mathcal{F}_{PKI}$, $\mathcal{F}_{SC}$, the ideal adversary $\mathcal{S}$, and the involved running parties $P_1, \ldots, P_n$, whose description is built into protocol π_{SM},

in a formal language. By this, we will be able to derive and formally specify results of the interaction between the processes. Then, we can compare the ways the environment interacts with the two systems, the first containing $\mathcal{A}$ and π_{SM}, denoted by $M_{\pi_{SM}}$, and the second containing $\mathcal{S}$ and $\mathcal{F}_{SM}$, denoted by M_I. Note that we need to compare only the external behaviour of the two systems, i.e. the input/output messages exchanged with the environment.

For this purpose, we use the language μCRL [4, 2]. It is a process algebraic language that can handle abstract data. It is very suitable for the analysis of interacting concurrent processes, since it is supported by a proof theory and proof methodology that can be used for verification purposes, as well as by a toolset for the automatic analysis and manipulation of μCRL specifications.

For lack of space we do not give the specifications of the processes, but only point out the most important bits. The set of atomic actions *Act* consists of two sets, the set of internal actions, Act_I, containing all messages sent and received between the processes, and the set of external actions, Act_E, containing the the messages exchanged between the processes and the environment. We define a set *Nonce* instead of using a random nonce generator. Registered entries within a process are stored in a buffer. The behaviour of each process $\mathcal{P}$ (such as $\mathcal{F}_{SM}$, $\mathcal{F}_{PKI}$, $\mathcal{S}$, $P_1, \ldots, P_n$) is described by a μCRL specification that we denote by $X_{\mathcal{P}}$. Informally, $X_{\mathcal{P}}$ is a μCRL process equation that exactly specifies the read and send activities the process $\mathcal{P}$ is performing, and in which order. From specification $X_{\mathcal{P}}$ one can follow how process $\mathcal{P}$ exchanges information and thus communicates with other processes, and how this information reflects the future state and activities of the process. Therefore, this specification can be seen as a formal expression (using the formal language) that captures exactly the definition of process $\mathcal{P}$ given in the UC-framework (in the appendix). As such, it can be easily derived from the definition of process $\mathcal{P}$.

Once we have obtained formal specifications of all involved processes, we can compose them and we can analyze the behaviour of the composed processes. For our purpose, we need to consider the composition of $\mathcal{F}_{SM}$ and $\mathcal{S}$ when run in parallel with "dummy" processes $\tilde{P}_1 \ldots \tilde{P}_n$ on one side, and the parallel composition of $\mathcal{F}_{SC}, \mathcal{F}_{PKI}, P_1, \ldots, P_n$ on the other side. Formally, we write

$$M_I = X_{\mathcal{F}_{SM}} \,||\, X_{\mathcal{S}} \,||\, X_{\tilde{P}_1} \,||\ldots||\, X_{\tilde{P}_n}$$

and

$$M_{\pi_{SM}} = X_{\mathcal{F}_{SC}} \,||\, X_{\mathcal{F}_{PKI}} \,||\, X_{P_1} \,||\ldots||\, X_{P_n}.$$

A set of rules in the μCRL language expresses how process expressions, when composed, ought to be manipulated in order to obtain the behaviour of their composition. Thus, from the specifications of the separate processes ($\mathcal{F}_{SM}$, $\mathcal{S}$, $\mathcal{F}_{SC}$, etc.) we are able to derive the process expressions of their compositions, M_I and $M_{\pi_{SM}}$.

The next step in our analysis is to compare the interaction of the two composed systems with the environment, ignoring the way the built-in parts interact with each other. Therefore, we shall distinguish between the external and the internal behaviour of the systems. All communications between the processes

within the system, either M_I or $M_{\pi_{SM}}$, are considered to be internal (the subset of actions Act_I). The interaction with the environment makes the external behaviour. Our aim is to compare the external behaviours of M_I and $M_{\pi_{SM}}$ after abstraction from the internal activities. The μCRL language contains a so-called *hiding operator* used to model abstraction from internal activities. Without going into detail about its derivation, we claim that the two systems exhibit the same external behaviour (i.e. both interact with the environment in the same way) which, for the registration phase, can be informally described as follows:

The system receives $(i, \mathbf{SM.Register})$ from the environment:

1. If P_i has already been registered, send $(i, \mathbf{SM.Already.Registered})$ to the environment and stop.
2. Otherwise, send $(i, \mathbf{SC.KeyGen})$ to the signcryption module.
3. Receive $(i, \mathbf{SC.Key}, (pk_i^s, pk_i^r))$ from the signcryption module.
4. Send $(i, \mathbf{PKI.Register}, (pk_i^s, pk_i^r))$ to the PKI module.
5. Receive $(i, \mathbf{PKI.Registered})$ from the PKI module.
6. Send $(i, \mathbf{SM.Registered})$ to the environment.

Fig. 1. Interaction with the environment in the registration phase for both systems.

If we denote the external behaviour of M_I and $M_{\pi_{SM}}$ by EX_I and $EX_{\pi_{SM}}$, respectively, we have the following lemma:

Lemma 1. *EX_I and $EX_{\pi_{SM}}$ specify branching bisimilar processes.*

Note that branching bisimulation is a rather strong equivalence for our needs here. Indeed, it is sufficient to consider trace equivalence or weak bisimulation. However, the proof methodology of this language is investigated with respect to this equivalence.

References

1. R. Canetti, *Univerally composable security: A new paradigm for cryptographic protocols*, Cryptology ePrint archive, Report 2000/067, 2005, available at `http://eprint.iacr.org/2000/067`.
2. W. J. Fokkink, J. Pang, *Cones and foci for protocol verification revisited*, In: Proceedings of the 6th Conference on Foundations of Software Science and Computation Structures, LNCS 2620, Springer, pp 267281, 2003.
3. K. Gjøsteen, L. Kråkmo, *Universally composable signcryption*, manuscript, 2005.
4. J. F. Groote, A. Ponse, *The sintax and semantics of μCRL*, in Algebra of Communicating Processes '94, Workshop in Computing Series, A. Ponse, C. Verhoef, S.F.M. van Vlijmen, (eds.), pp. 26-62, 1995.
5. J. C. Malone-Lee, *On the security of signature schemes and sighcryption schemes*, PhD. thesis, University of Bristol, 2004.

6. Y. Zheng, *Digital signcryption or How to achieve Cost(Signature & Encryption) ¡¡ Cost(Signature) + Cost(Encryption)*, Proc. of the 17th Annual International Cryptology Conference - CRYPTO '97, B.S. Kaliski Jr. (Ed.), LNCS 1294, Springer-Verlag, pp. 165-179, 1997.

A Functionalities and protocols

When (**PKI**.**Register**, v) is received from $\tilde{P}_i$:

1. If $\mathcal{F}_{PKI}$ has already processed a **PKI**.**Register** message from $\tilde{P}_i$, send **PKI**.**Already**.**Registered** to $\tilde{P}_i$ and stop.
2. Send (**PKI**.**Register**, P_i, v) to $\mathcal{A}$ and wait for (**PKI**.**Register**.**OK**, P_i) from $\mathcal{A}$.
3. Store (P_i, v) in the registration buffer and send (**PKI**.**Registered**) to $\tilde{P}_i$.

When (**PKI**.**Retrieve**, P_i) is received from $\tilde{P}_j$:

1. Send (**PKI**.**Retrieve**, P_j, P_i) to $\mathcal{A}$ and wait for (**PKI**.**Retrieve**.**OK**, P_j, P_i) from $\mathcal{A}$.
2. If there is an entry (P_i, v) in the registration buffer, send (**PKI**.**Retrieved**, v) to $\tilde{P}_j$, otherwise send (**PKI**.**Retrieved**, $\perp$) to $\tilde{P}_j$.

Fig. 2. The PKI functionality $\mathcal{F}_{PKI}$.

When **SC.KeyGen** is received from $\tilde{P}_i$:

1. If $\mathcal{F}_{SC}$ has already processed an **SC.KeyGen** message from $\tilde{P}_i$, send **SC.Key.Already.Generated** to $\tilde{P}_i$ and stop.
2. Send $(\textbf{SC.KeyGen}, P_i)$ to the adversary $\mathcal{A}$ and wait for $(\textbf{SC.Key}, P_i, pk_i^s, pk_i^r)$ from $\mathcal{A}$. (We note that $\mathcal{A}$ is free to choose (pk_i^s, pk_i^r), subject to the restriction that pk_i^s or pk_i^r should not have appeared before in messages between $\mathcal{A}$ and $\mathcal{F}_{SC}$.)
3. Store (P_i, pk_i^s, pk_i^r) in the public key buffer and send $(\textbf{SC.Key}, pk_i^s, pk_i^r)$ to $\tilde{P}_i$.

When $(\textbf{SC.Encrypt}, pk^r, m)$ is received from $\tilde{P}_i$:

1. If an entry $(P_i, pk_i^s, \cdot)$ is not stored in the public key buffer, send **SC.Key.Not.Generated** to $\tilde{P}_i$ and stop.
2. If an entry $(P_j, \cdot, pk^r)$ is stored in the public key buffer:
   (a) Send $(\textbf{SC.Encrypt}, pk_i^s, pk^r, |m|)$ to $\mathcal{A}$ and wait for $(\textbf{SC.Ciphertext}, pk_i^s, pk^r, c)$ from $\mathcal{A}$. (We note that $\mathcal{A}$ is free to choose c, subject to the restriction that c should not have appeared before in an **SC.Ciphertext** message from $\mathcal{A}$ to $\mathcal{F}_{SC}$, nor in an **SC.Decrypt** message to $\mathcal{A}$ from $\mathcal{F}_{SC}$.)
   (b) Store (pk_i^s, pk^r, c, m) in the ciphertext buffer, send $(\textbf{SC.Ciphertext}, c)$ to $\tilde{P}_i$ and stop.
3. Send $(\textbf{SC.Encrypt}, pk_i^s, pk^r, m)$ to $\mathcal{A}$ and wait for $(\textbf{SC.Ciphertext}, pk_i^s, pk^r, c)$ from $\mathcal{A}$. (We note that the above restrictions on c also apply here.)
4. Send $(\textbf{SC.Ciphertext}, c)$ to $\tilde{P}_i$.

When $(\textbf{SC.Decrypt}, pk^s, c)$ is received from $\tilde{P}_j$:

1. If an entry $(P_j, \cdot, pk_j^r)$ is not stored in the public key buffer, send **SC.Key.Not.Generated** to $\tilde{P}_j$ and stop.
2. Send $(\textbf{SC.Decrypt}, pk^s, pk_j^r, c)$ to $\mathcal{A}$ and wait for $(\textbf{SC.Plaintext}, pk^s, pk_j^r, m')$ from $\mathcal{A}$.
3. If there is an entry (pk^s, pk_j^r, c, m) in the ciphertext buffer, send $(\textbf{SC.Plaintext}, m)$ to $\tilde{P}_j$ and stop.
4. If there is an entry $(P_i, pk^s, \cdot)$ in the public key buffer, send $(\textbf{SC.Plaintext}, \perp)$ to $\tilde{P}_j$ and stop.
5. Send $(\textbf{SC.Plaintext}, m')$ to $\tilde{P}_j$.

Fig. 3. The signcryption functionality $\mathcal{F}_{SC}$.

When P_i receives **SM**.**Register** from the environment:

1. If P_i has already processed **SM**.**Register**, send **SM**.**Already**.**Registered** to the environment and stop.
2. Send **SC**.**KeyGen** to $\mathcal{F}_{SC}$ and wait for $(\mathbf{SC}.\mathbf{Key}, pk_i^s, pk_i^r)$ from $\mathcal{F}_{SC}$.
3. Send $(\mathbf{PKI}.\mathbf{Register}, (pk_i^s, pk_i^r)$ to $\mathcal{F}_{PKI}$ and wait for **PKI**.**Registered** from $\mathcal{F}_{PKI}$.
4. Send **SM**.**Registered** to the environment.

When P_i receives $(\mathbf{SM}.\mathbf{Encrypt}, m)$ from the environment:

1. If P_i has not processed **SM**.**Register**, send **SM**.**Not**.**Registered** to the environment and stop.
2. Send $(\mathbf{PKI}.\mathbf{Retrieve}, P_j)$ to $\mathcal{F}_{PKI}$ and wait.
3. If $\mathcal{F}_{PKI}$ replies with $(\mathbf{PKI}.\mathbf{Retrieved}, \perp)$, send **SM**.**Recipient**.**Not**.**Registered** to the environment and stop.
4. Otherwise $\mathcal{F}_{PKI}$ replies with $(\mathbf{PKI}.\mathbf{Retrieved}, (\cdot, pk_j^r))$.
5. Send $(\mathbf{SC}.\mathbf{Encrypt}, pk_j^r, m)$ to $\mathcal{F}_{SC}$ and wait for $(\mathbf{SC}.\mathbf{Ciphertext}, c)$ from $\mathcal{F}_{SC}$.
6. Send $(\mathbf{SM}.\mathbf{Ciphertext}, c)$ to the environment.

When P_j receives $(\mathbf{SM}.\mathbf{Decrypt}, c)$ from the environment:

1. If P_j has not processed **SM**.**Register**, send **SM**.**Not**.**Registered** to the environment and stop.
2. Send $(\mathbf{PKI}.\mathbf{Retrieve}, P_i)$ to $\mathcal{F}_{PKI}$.
3. If $\mathcal{F}_{PKI}$ replies with $(\mathbf{PKI}.\mathbf{Retrieved}, \perp)$, send **SM**.**Sender**.**Not**.**Registered** to the environment and stop.
4. Otherwise $\mathcal{F}_{PKI}$ replies with $(\mathbf{PKI}.\mathbf{Retrieved}, (pk_i^s, \cdot))$.
5. Send $(\mathbf{SC}.\mathbf{Decrypt}, pk_i^s, c)$ to $\mathcal{F}_{SC}$ and wait for $(\mathbf{SC}.\mathbf{Plaintext}, m)$ from $\mathcal{F}_{SC}$.
6. Send $(\mathbf{SM}.\mathbf{Plaintext}, m)$ to the environment.

Fig. 4. The secure messaging protocol π_{SM}.

When **SM.Register** is received from $\tilde{P}_i$:

1. If $\mathcal{F}_{SM}$ has already processed an **SM.Register** message from $\tilde{P}_i$, send **SM.Already.Registered** to $\tilde{P}_i$ and stop.
2. Send (**SM.Register**, P_i) to the adversary $\mathcal{A}$ and wait for (**SM.Register.OK**, P_i) from $\mathcal{A}$.
3. Store P_i in the registration buffer and send **SM.Registered** to $\tilde{P}_i$.

When (**SM.Encrypt**, P_j, m) is received from $\tilde{P}_i$:

1. If P_i is not stored in the registration buffer, send **SM.Not.Registered** to $\tilde{P}_i$ and stop.
2. Send (**SM.Encrypt**, $P_i, P_j, |m|$) to the adversary $\mathcal{A}$ and wait for (**SM.Ciphertext**, P_i, P_j, c) from $\mathcal{A}$. (We note that $\mathcal{A}$ is free to choose c, subject to the restriction that c should not have appeared before in an **SM.Ciphertext** message from $\mathcal{A}$ to $\mathcal{F}_{SM}$, nor in an **SM.Decrypt** message to $\mathcal{A}$ from $\mathcal{F}_{SM}$ that resulted in an **SM.Plaintext** being sent from $\mathcal{F}_{SM}$.)
3. If P_j is not stored in the registration buffer, send **SM.Recipient.Not.Registered** to $\tilde{P}_i$ and stop.
4. Store (P_i, P_j, c, m) in the ciphertext buffer and send (**SM.Ciphertext**, c) to $\tilde{P}_i$.

When (**SM.Decrypt**, P_i, c) is received from $\tilde{P}_j$:

1. If P_j is not stored in the registration buffer, send **SM.Not.Registered** to $\tilde{P}_j$ and stop.
2. Send (**SM.Decrypt**, P_j, P_i, c) to the adversary $\mathcal{A}$ and wait for (**SM.Plaintext**, P_j, P_i, m') from $\mathcal{A}$.
3. If P_i is not stored in the registration buffer, send **SM.Sender.Not.Registered** to $\tilde{P}_j$ and stop.
4. If an entry (P_i, P_j, c, m) is stored in the ciphertext buffer, send $(SM.Plaintext, m)$ to $\tilde{P}_j$, otherwise send $(SM.Plaintext, \perp)$ to $\tilde{P}_j$.

Fig. 5. The secure messaging functionality $\mathcal{F}_{SM}$.

When $\mathcal{S}$ receives $(\textbf{SM.Register}, P_i)$ from $\mathcal{F}_{SM}$:

1. Send $(\textbf{SC.KeyGen}, P_i)$ to $\mathcal{A}_{SC}$ and wait for $(\textbf{SC.Key}, P_i, pk_i^s, pk_i^r)$.
2. Store (P_i, pk_i^s, pk_i^r) in the public key buffer.
3. Send $(\textbf{PKI.Register}, (P_i, pk_i^s, pk_i^r))$ to $\mathcal{A}_{PKI}$ and wait for $(\textbf{PKI.Register.Ok}, P_i)$ from $\mathcal{A}_{PKI}$.
4. Send $(\textbf{SM.Register.Ok}, P_i)$ to $\mathcal{F}_{SM}$.

When $\mathcal{S}$ receives $(\textbf{SM.Encrypt}, P_i, P_j, |m|)$ from $\mathcal{F}_{SM}$:

1. Send $(\textbf{PKI.Retrieve}, P_i, P_j)$ to $\mathcal{A}_{PKI}$ and wait for $(\textbf{PKI.Retrieve.Ok}, P_i, P_j)$ from $\mathcal{A}_{PKI}$.
2. If there is no entry $(P_j, \cdot, \cdot)$ in public key buffer, send $(\textbf{SM.Ciphertext}, P_i, P_j, \perp)$ to $\mathcal{F}_{SM}$ and stop.
3. Find entries $(P_i, pk_i^s, \cdot)$ and $(P_j, \cdot, pk_j^r)$ in public key buffer, send $(\textbf{SC.Encrypt}, pk_i^s, pk_j^r, |m|)$ to $\mathcal{A}_{SC}$, and wait for $(\textbf{SC.Ciphertext}, pk_i^s, pk_j^r, c)$ from $\mathcal{A}_{SC}$.
4. Send $(\textbf{SM.Ciphertext}, P_i, P_j, c)$ to $\mathcal{F}_{SM}$.

When $\mathcal{S}$ receives $(\textbf{SM.Decrypt}, P_j, P_i, c)$ from $\mathcal{F}_{SM}$:

1. Send $(\textbf{PKI.Retrieve}, P_j, P_i)$ to $\mathcal{A}_{PKI}$ and wait for $(\textbf{PKI.Retrieve.Ok}, P_j, P_i)$ from $\mathcal{A}_{PKI}$.
2. If there is no entry $(P_i, \cdot, \cdot)$ in public key buffer, send $(\textbf{SM.Plaintext}, P_j, P_i, \perp)$ to $\mathcal{F}_{SM}$ and stop.
3. Find entries $(P_i, pk_i^s, \cdot)$ and $(P_j, \cdot, pk_j^r)$ in public key buffer, send $(\textbf{SC.Decrypt}, pk_i^s, pk_j^r, c)$ to $\mathcal{A}_{SC}$, and wait for $(\textbf{SC.Plaintext}, pk_i^s, pk_j^r, m')$ from $\mathcal{A}_{SC}$.
4. Send $(\textbf{SM.Plaintext}, P_j, P_i, m')$ to $\mathcal{F}_{SM}$.

Fig. 6. The ideal adversary $\mathcal{S}$.