Security Analysis of Parlay/OSA Framework

R. Corin[1], G. Di Caprio[3], S. Etalle[1], S. Gnesi[2], G. Lenzini[1,2], and C. Moiso[3]

[1] Departement of Computer Science, University of Twente
7500 AE Enschede, The Netherlands
{coring,etalle,lenzinig}@cs.utwente.nl
[2] Istituto di Scienza e Tecnologie dell'Informazione, ISTI-CNR
Area della Ricerca di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy
{gnesi,lenzini}@isti.cnr.it
[3] Telecom Italia Lab
Via G. Reiss Romolo 274, 1048 Torino, Italy
{corrado.moiso,gaetano.dicaprio}@tlab.com

Abstract. This paper analyzes the security of the Trust and Security Management (TSM) protocol, an authentication protocol which is part of the Parlay/OSA Application Program Interfaces (APIs). Architectures based on Parlay/OSA APIs allow third party service providers to develop new services that can access, in a controlled and secure way, to those network capabilities offered by the network operator. Role of the TSM protocol, run by network gateways, is to authenticate the client applications trying to access and use the network capabilities features offered. For this reason potential security flaws in its authentication strategy can bring to unauthorized use of network with evident damages to the operator and to the quality of services. This paper shows how a rigorous formal analysis of TSM underlines serious weaknesses in the model describing its authentication procedure. The paper relates about the design activity of the formal model, the tool-aided verification performed and the security flaws discovered. This will allow us to discuss about how the security of the TSM protocol can be generally improved.

1 Introduction

Service architectures based on Parlay/Parlay X Application Program Interfaces (APIs) [1], allow network operators to supply service capability features to third party service providers. Parlay/Parlay X APIs propose an attractive framework where programmers, not necessary experts in telephony or telecommunication, can develop innovative resources or design new services.

With a wider accessibility to network resources it becomes crucial the definition of access rules between the entities that make network capabilities and the entities that access to them, so that the operator could maintain the full control over the usage of their resources and on the quality of the services offered. For this reason quality parameters need to be carefully evaluated before any service is effectively executed. In particular, the security checks becomes of primary importance to avoid that unauthorized entities can reach to use the network.

Security in a distributed setting it is usually achieved by the use cryptographic protocols, but unfortunately their design is a formidably difficult task. In some cases subtle flaws were found only years after the publication of a protocol [5, 8], and experience teaches that security protocols need to be carefully checked. In this task Formal Methods [6] can help. Generally speaking formal methods provide engineers with valuable methodologies and tools for the analysis of complex, concurrent and distributed systems. They require the main characteristics of a system be represented by a formal (i.e., with a rigorous semantics) model.

Recent y formal methods have been profitably applied in the verification of many authentication security protocols (e.g., see [8]). They have been especially used in the formal analysis of security properties such as confidentiality and authenticity. Confidentiality guarantees the secrecy of sensitive information (e.g., personal information, session keys, reference to private interfaces etc.). Authenticity [9] provides a way of checking identities of agents involved in a communication, thereby avoiding for instance that an impostor uses a service while charging it on someone else. This paper reports and discusses the application of Formal Methods for verifying the security of the Trust and Security Management protocol in Parlay/OSA APIs [1]. This protocol is designed to protect telecommunication capabilities from unauthorized access and it implements an authentication procedure. The formal validation experience, conducted within a joint project between Universities and Industry, has underlined some security flaws in the authentication mechanism. Moreover from the analysis of the traces showing the attacks, we are able to suggest possible solutions to fix the security weaknesses discovered.

2 The Parlay/OSA Architecture

The Parlay/OSA specifications define an architecture that enables service application developers to make use of network functionality through an open standardized interface. Parlay/OSA APIs [1] provide an abstract and coherent view of heterogeneous network capabilities, and they allow a developer to interface its applications via distributed processing mechanisms.

The Parlay/OSA (Figure 1) architecture consists of:

- *a set of Client Applications* accessing the network resources;
- *a set of Service Interfaces*, or Service Capability Features (SCFs), that represent APIs to control the network capabilities provided by network resources they allows a developer to interface its applications via distributed processing mechanisms and offers potential guarantees of compliance with respect to some quality of service parameters. (e.g., controlling the routing of voice calls, sending/receiving SMSs, locating a terminal, etc.).
- *a Framework*, that provides a modular and "controlled" access to the SCFs.
- *Network Resources*, in the telecommunication network, implementing the network capabilities.

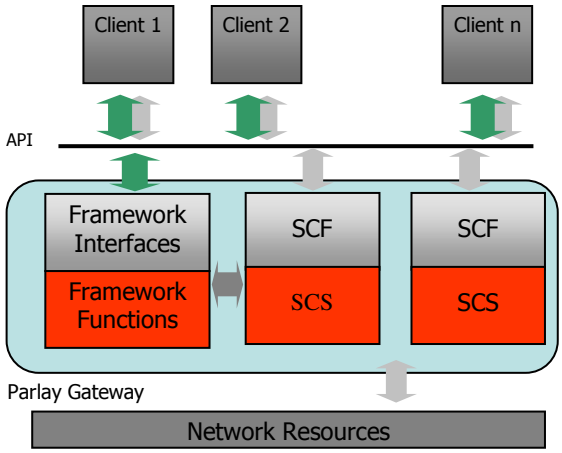

Fig. 1. The Parlay/OSA Architecture, and the points where Trust and Security Management protocol is involved.

A Parlay Gateway includes the framework functions and the Service Capability Services (SCSs), i.e., the modules implementing the SCFs: it is a logical entity that could be implemented in a distributed way by several systems.

Since the applications could be deployed in an administrative domain different from the one of the Parlay Gateway, the secure and controlled access to SCFs is a paramount aspect for the Parlay architecture.

In order to get the references of the required SCFs, an application must perform several interactions with the framework interfaces. For example the application must perform an authentication phase and then select the SCFs required, as described in Section 2.1. The framework verifies whether the application is authorized to use them, according to a subscription profile. Finally, an agreement is digitally signed, and the framework returns to the application the references to the required SCFs (e.g., as CORBA interface reference): these references are valid only for a single session of the application.

When the Framework has to return an SCF reference to an application it contacts the SCS which implements it, by passing all the configuration parameters, e.g., representing Service Level Agreement conditions, stored in the subscription profile of the application. The SCS creates a new instance of the SCF, configured with the received parameters, and returns its reference to the Framework. Each time the application invokes a method on the SCF instance, the SCS executes it by taking into account the configuration parameters received at instantiation time.

2.1 Trust and Security Management protocol

One of the critical steps for guarantee the controlled access to the SCFs is the authentication phase between the Gateway and the application. It is supported

by the protocol implemented by the Trust and Security Management API. The paper is focusing on the analysis of the properties of this security protocol, whose behavior is described by the MSC in Figure 2. The main steps of the protocol are:

- Initiate Authentication: the client invokes "`initiateAuthenticationWithVersion`" on the Framework's public interface (e.g., an URL) to initiate the authentication process. The client provides a reference to its own authentication interface, and the Framework returns a reference to its authentication interface;
- Select Authentication Mechanism: the client invokes "`selectAuthenticationMechanism`" on the Framework authentication interface, to negotiate the authentication algorithm (e.g., a hash function) that will be used in the authentication steps;
- The client and the framework authenticate each other . Each authentication step is performed following a one-way Challenge Handshake Authentication Protocol (CHAP) [7], i.e. by issuing a challenge in the "challenge" method, and checking if the partner returns the correct response. The Framework could authenticate the client before the client authenticates the Framework, or afterwards, or the two authentication processes could be interleaved. However, the client shall respond immediately to any challenge issued by the Framework, as the Framework might not respond to any challenge issued by the client until the Framework has successfully authenticated the client. An invocation of the method "`authenticationsucceeded`" signals the success of the challenge;
- Request an access session: when authenticated by the Framework, the client is permitted to invoke "`requestAccess`" to start an access session. The client provides a reference to its own Access interface, and the Framework returns a reference to Access interface, unique for this client;
- The Access interface is used to negotiate the signing algorithm to be used in the session and to obtain references to other Framework interfaces (we will call them, service framework interfaces), such as service discovery, service agreement management, etc. .

With the reference to a service framework interface the TSM finishes. Note that if an intruder obtained the reference to these interfaces it could arrive to have the interfaces to access to the services. For this reason we will limit our analysis to the secrecy of the only service framework interfaces.

In fact, after TSM ends a client selects the required SCFs by invoking the "`selectService`" method on the service agreement management interface, optionally after invoking the Discovery interface to obtain a list of the SCFs supported by the Framework. The client obtains a service token, which can be signed as part of the service agreement by the client and the Framework, through "`signServiceAgreement`" and the "`signAppServiceAgreement`" methods. The service token generally has a limited lifetime: if the lifetime of the service token expires, a method accepting the service token will return an error code. If the

sign service agreement phase succeeds, the Framework returns to the client a reference to the selected SCF, personalized with the client configuration parameters.

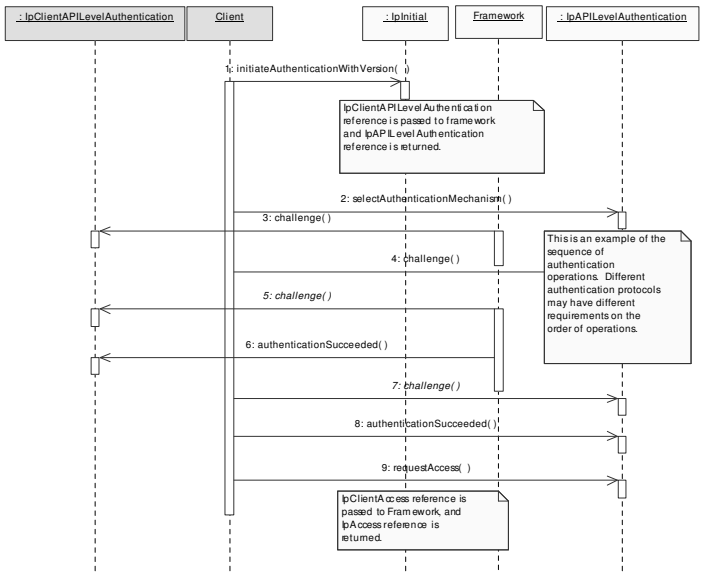

Fig. 2. Message Sequence Chart of the Trust and Security Management Protocol from the UML specification of the protocol [1]

3 Security Formal Analysis

This section explains in detail our experience of formal analysis of security properties over the TSM protocol. To carry out the verification phase we have used "CoProVe" [4], a tool developed at the Twente University (NL), which is specifically designed for model-checking security protocols[1] . Basically model- checking [3] is a validation methodology used to verify automatically if a certain desired behavior is satisfied by a formal and finite-state specification of a system (called model). If an error is recognized a counter-example showing under which circumstances the error happens, is usually generated. A counter-example provides evidence that the model is faulty and it needs to be revised. This means that, if the model is sound with respect the system it represents, that the system itself is prone to show the same error.

We anticipate that our analysis of the TSM protocol has lead to spotting (and to suggest possible patches) important security flaws.

[1] CoProVe is also freely accessible via the web at `http://wwwes.cs.utwente.nl/24cqet/coprove.html`

3.1 Formal Models

One of the challenges in applying these tools to industrial architectures lies in translating the (usually less formal) architecture specification into a formal model. This experience has been very instructive in many respects. In the first place, translating a complex system design into a formal protocol specification involves many non-trivial steps: software technology concepts such as method invocation and port specification have to be "encoded" into a straightforward protocol specification. This encoding phase forces the engineer to reason about the security implication of using these constructs.

The Parlay/OSA Framework API specification consists of many pages, and hence it is extremely difficult to have a good overview of the security mechanisms. By extracting a short formal specification out of those pages we were able to have a concise yet complete summary of the security core of the system. To translate the TSM specification into a protocol of a few lines we had to take some modelling choices:

- a reference to a (new) private interface f was modeled by a (new) encryption shared key K_f;
- calling of a method with parameter m over a new interface f was translated into a sending message m encrypted with the relative key K_f. Dually, getting a result from a method invocation was translated into a receiving message still encrypted with K_f;

The use of an encryption key reflects the fact that an intruder that does not know the interface reference is not allow to infer anything that comes from any method's invocation over that interface. Notice that this latter modeling choice is intentionally optimistic.

In the following we report the obtained formal model, written in the usual abstract representation of cryptographic protocols (we called abstract model).

* initiate *
step 1. $C \longrightarrow F : C; KC$
step 2. $F \longrightarrow C : KF$

* select authentication methods *
step 3. $C \longrightarrow F : \{h; h'; h''\}_{KI}$
step 4. $F \longrightarrow C : \{h\}_{KI}$

* challenge *
step 5. $F \longrightarrow C : \{\{F, N\}_{pk(C)}\}_{KC}$
step 6. $C \longrightarrow F : \{\{C, h(N, SCF)\}_{pk(F)}\}_{KF}$
step 7. $F \longrightarrow C : \{ok/fail\}_{KC}$

* request access *
step 8. $C \longrightarrow F : \{req, KC\}_{KF}$
step 9. $F \longrightarrow C : \{KA/fail\}_{KF}$

* select signing methods *
step 10. $C \longrightarrow F : \{s; s'; s''\}_{KI}$
step 11. $F \longrightarrow C : \{s\}_{KI}$

* request framework service interface *
step 12. $C \longrightarrow F : \{req\ \prime\}_{KA}$
step 13. $F \longrightarrow C : \{KS/fail\}_{KA}$

In the abstract model, C represents a client and F the Framework, while $C \longrightarrow F : M$ depicts a communication C and F involving a message M. With $\{M\}_K$ we indicate the message M encrypted with a key K, while with $h(M)$ we write the result of applying a hash function h to a message M.

Precisely step 1 a represents the client that initiates the protocol over the public interface of the framework, by providing its name and a reference to its interface. As a reply the framework displays a new interface KF. In step 3 and 4 (resp., 10 and 11) the client asks the framework to chose an authentication procedure (resp., signing methods). Step 5 and 6 model the authentication challenge, based on the CHAP, between F and C using the hash function h selected. Here SCF represent the secret, shared between C and F, required by the CHAP [7].

We want to underline that it is not clear, reading the specification, if this shared secret is indeed implemented in the "challenge" method, and if it is the case, when/how this secret should have to be exchanged. As a consequence we designed more models each containing a different implementations of the CHAP steps (without secret, with two secrets, with messages signed). We performed our analysis over all these models, obtaining for all the same results. In this paper we will describe the model which seems to be intuitively, the most secure, precisely that one which assumes that a shared secret between the client and the framework exists. We anticipate that even in this choice will not be sufficient to avoid a security flaw.

In steps 8 and 9 the client requests for an interface where to invoke the request access for a service. Finally, steps 12 and 13 represent the client asking for a request and receiving back the reference to the framework interface. The abstract model in Figure 3, or part of it, has been then translated into the specific language required in input by the tool CoProVe. In other words the abstract model has been used as basis for specifying more concrete formal models.

In the following we comment one of them, precisely the one used in the analysis of the secrecy of the interface that the client uses to request the access to a service. It implements the steps (1-2);(5-9);(12-13) of the abstract model:

```
% Initiator role specification
client(C,F,Kc,Kf,N,Req,Ka,Scf,[
   send([C,Kc]),
   recv(Kf),
   recv([F,N]+Kc),
   send([C,sha([N,Scf])]+Kf),
   send(Req+Kf),
   recv(Ka+Kf)]).

% Responder role specification
framework(C,F,Kc,Kf,N,Req,Ka,Scf[
   recv([C,Kc]),
   send(Kf),
   send([F,N]+Kc),
   recv([C,sha([N,Scf])]+Kf),
   recv(Req+Kf),
   send(Ka+Kf)]).

% Secrecy check (it is a singleton role)
secrecy(N, [ recv(N) ] ).

% scenario specification pairs [name, Name]
% [label for the role; actual role]
scenario
([[c,Client1],[f,Framew1],[sec,Secr1]]):-
   client(c,f,kc,_,_,req,_,scf,Client1),
   framework(c,f,_,kf,n,_,ka,scf,Framew1),
   secrecy(ka, Secr1).

% The initial intruder knowledge
initial_intruder_knowledge([c,f,e]).

% specify which roles we want to force
% to finish (only sec in this example)
has_to_finish([sec]).
```

The previous specification involves three principals: one client (`c`), one framework (`f`) and eavesdropping agent (`sec`) that receives any message passing in the net. Each role is specified by a sequence of send/receive actions that mimic exactly the steps of the abstract model. Symbol "`+`" is used to denote symmetric encryption using shared keys and "`*`" for asymmetric encryption using public/private key pairs. Formal parameters (e.g., in the client role `C,F,Kc,Kf`, `N,Req,Ka,Scf`) are used to denote all the objects used in the role specification. In a scenario those parameter are instantiated with actual constant representing real object (i.e., `c,f,\_,kf,n,\_,ka,scf`). Here "`_`" is used when no instantiation is required, that is when a free variable is involved. The intruder is assumed

to know only the client and framework names plus its own name "`e`". Verification of secrecy (i.e., the quest for a secrecy flaw) consists in asking if there is a trace bringing the eavesdropper to know a secret. So for example, for the secrecy of `Ka`(which models the reference of the interface used by the client to request an access session) we check if agent `sec`knows `Ka`.

3.2 Formal Analysis and Detected Weakness

The analysis we done upon the model of TSM protocol, pointed out the following weaknesses in the security mechanism. In the following we will describes the flaws discovered as a commented list of items. In one case we also show the output produced by our tool and we comment how the flaw can be ricognized from it. In the next section we will discuss about from what weaknesses in the TSM the flaws have origin and how it can be possibile to avoid them.

Flaw 1. An intruder can impersonate a client and start an authentication challenge with the framework.

Informally a malicious application can obtain the reference to the method used to start the authentication procedure. Formally an intruder can obtain the reference to the interface used by the client to start the authentication challenge (key `Kf`).

Flaw 2. An intruder can impersonate a client, authenticate itself to the framework and obtain the reference to the interface used to request an access to a service.

This is a serious flaw that compromises the main goal of the protocol itself. Informally a malicious application can pass the authentication phase instead of an honest client. Our analysis shows also that, as a consequence, an intruder can even obtain a reference to the interface used to request a service (key `Ka`). In order to understand better this weakness, let us study the output of CoProVe showing the attack:

```
[c,send([c,kc])]
[f,recv([c,kc])]
[c,recv(_h325)]
[f,send(kf)]
[f,send([f,n] + kc)]
[c,recv([f,n] + kc)]
[c,send([c,sha([n,scf])] + _h325)]
[f,recv([c,sha([n,scf])] + kf)]
[c,send(req + _h325)]
[c,recv(_h391 + _h325)]
[f,recv(req + kf)]
[f,send(ka + kf)]
[sec,recv(ka)]
```

The previous trace can be read in the following way. Each row represents a communication action. For example `[c,send(req+_h314)]` represents the action "send" that the client "`c`" executes with message "`req`" encrypted with a new name (generated by the intruder) "`_h134`". The sequence of actions describes the attack. By analyzing the traces in this way, it comes out that an intruder can use an honest client as an oracle to obtain the right answer required by the authentication procedure and after that to make the framework reveal the secret "`Ka`". In fact if we write the sequence of trace in the informal notation the attach may be better visualized as

$$
\begin{array}{ll}
\text{step 1.} & C \longrightarrow I(F) : C, KC \\
\text{step 1'.} & I(C) \longrightarrow F : C, KC \\
\text{step 2.} & I(F) \longrightarrow C : Ke \\
\text{step 2'.} & F \longrightarrow I(C) : KF \\
 & \\
\text{step 5'.} & F \longrightarrow I(C) : \{F, N\}_{KC} \\
\text{step 5.} & I(F) \longrightarrow C : \{F, N\}_{KC} \\
\text{step 6.} & C \longrightarrow I(F) : \{C, h(N, SCF)\}_{Ke} \\
\text{step 6'} & I(C) \longrightarrow F : \{C, h(N, SCF)\}_{KF} \\
 & \\
\text{step 8.} & C \longrightarrow I(F) : \{req\}_{Ke} \\
\text{step 9} & I(F) \longrightarrow C : \{fail\}_{Ke} \\
\text{step 8'.} & I(C) \longrightarrow F : \{req\}_{KF} \\
\text{step 9'.} & F \longrightarrow I(C) : \{KA\}_{KF}
\end{array}
$$

The attach shows two parallel runs of the protocol, where the intruder play, respectively, the role of the client against the framework ($I(C)$ in steps i') and the framework against the client ($I(F)$ in steps i), where i is the step number w.r.t. the abstract model. K_e is the intruder's key. It is worth to underline that this attack exists even if we assume a perfect encryption assumption [5] (i.e., the cryptographic engine is assumed to be unbreakable).

Flaw 3. An intruder can impersonate a client, authenticate itself to the framework, send a request of a service and obtain the reference to a service framework interface.

This is also a serious flaw that compromises the main goal of the protocol itself. An intruder can obtain the reference to the framework interface (key `Kf`). Mainly it is a consequence of flaw 1 and 2: once an intruder authenticated itself instead of the client, it can easily obtain the reference also. By the way it can be shown that the intruder can obtain that reference by a man-in-the-middle attack, where the intruder simply listen the communication between the client and the framework and it steals the reference when it is passed in clear. In out model this attack can be explained in the following way: the intruder steals, by eavesdropping, the message $\{KS\}_{KA}$, encrypted with KA, and it decrypts it. This is possible because the encryption key KF is passed in clear and, by eavesdropping the intruder can easily obtain $\{KA\}_{KF}$, and hence KA.

Flaw 4. An intruder can force the framework to use an authentication mechanism of her choice.

This last attack shows how an intruder, by the use of a replay attack, can force the Framework to choose a particular authentication mechanism (or a particular digital signature procedure), later used along the protocol.

4 Discussion

This section discusses the weaknesses here presented, and argues about possible solutions to increase the overall security.

We start with some preliminary considerations: first of all, the common practice in protocol engineering [2] suggests the use of session keys to protect the confidentiality of sensitive information, which in TSM case are the references to interfaces. In fact, from our analysis arises that the core of the security weaknesses we found, lays in the fact that the references to interfaces are passed in clear. Session keys are indeed missing at all in the present implementation [2], while their use could avoid that an intruder can access to a reference interface itself by simply eavesdropping, with a men-in-the-middle attack, the communication between a client and a framework (see flaw 1 to 3). Note that unfortunately it is not sufficient to establish a session key as a result of the "challenge" phase. Flaw 2 shows, in fact, that an intruder can pass the authentication challenge instead of a client; as a consequence it can also agree a session key with the framework. A check with CoProVe formally confirms this argument. This means that the security strategy needs to be globally reviewed. An additional discussion regards the correct use of a CHAP-based authentication. Reading the specification it seems that security can be ensured if the "challenge" is frequently invoked by the framework to authenticate the client that, in turn, must reply "immediately". In fact

> "The Framework authenticates the client. The sequence diagram illustrates one of a series of one or more invocations of the challenge method on the client's API Level Authentication interface. In each invocation, the Framework supplies a challenge and the client returns the correct response. The Framework could authenticate the client before the client authenticates the Framework, or afterwards, or the two authentication processes could be interleaved. However, *the client shall respond immediately to any challenge issued by the Framework*, as the Framework might not respond to any challenge issued by the client until the Framework has successfully authenticated the client" ([1], page 19)

These qualitative adjectives give only an apparent image of security. In fact our analysis prove that not only the intruder can acts as a client towards the framework (e.g., flaw 2), but it can passively observe, as man-in-the-middle,

[2] Do not confuse them with the session keys that appear in the abstract model. Those are only part of the model and represent private references to interfaces.

the framework and a client to authenticate each other, and eventually steal the reference to the service framework interfaces when it is transmitted in clear (see flaw 3). At this point the intruder can substitute itself the client.

Flaw 4 has a different nature. It teaches that particular care must be reserved on the choice of the encryption algorithms or digital signature procedures offered by the Framework considering that if there is one which is known particular weak or flawed then the intruder can force the use it. For example it must be avoided the offer of communication in clear-text, or encryption procedures that are known to be particularly weak.

5 Conclusions

This paper discusses an experience of formal analysis of security aspects of the Parlay/OSA Trust and Security Management protocol. The protocol aims to authenticate clients before they access to network services. Our experience confirms that formal methods can help to discover serious security flaws. More precisely, this is evidenced in two respects. First, the use of a formal model, where only the relevant security features are expressed, helps in pointing out what are the critical parts for security. In an informal description, on the other hand, this information is usually dispersed and difficult to gather. Second, the use of an automatic tool provides us with unambiguous proofs of attacks, and from the analysis of the events conducting to them, useful guidelines to improve security.

6 Acknowledgement

S. Gnesi and G. Di Caprio, and C. Moiso were supported by the MIUR-CNR Project SP4. R. Corin and S. Etalle were supported also, by the project "Privacy in an Ambient World" (PAW) a TUD DIES KUN TNO EIB TNO FEL collaboration funded by IOP GenCom under project nr. IGC-03001-IOP. G. Lenzini was supported by both SP4 and PAW Projects.

References

[1] *Open Service Access (OSA) - Application Programming Interface (API) Mapping for OSA*. http://www.3gpp.org/ftp/Specs/archive/29_series. Release 5.

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. Technical Report Research Report 125, Digital Systems Research Center, 1994.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Transaction on Programming Languages and Systems*, 2(8):244–263, 1986.

[4] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In *Proc. of s 9th International Static Analysis Symposium (SAS)*, number 2477 in LNCS, pages 326–341. Springer-Verlag, 2002.

[5] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.
[6] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 5(7):11–19, 1990.
[7] G. Leduc. Verification of two versions of the challenge handshake authentication protocol (chap). *Annals of Telecommunications*, 1-2(55):18–30, 1999.
[8] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. *Software Concepts and Tools*, 3(17):93–102, 1997.
[9] G. Lowe. A hierarchy of authentication specifications. In *Proc. of the 10th Computer Security Foundations Workshop (CSFW)*, pages 31–44. IEEE Computer Society, 1997.