APS Files for Selected Authentication Protocols

D42.8

Document Identification	
Date	18/04/2015
Status	Final
Version	1.0

Related SP/WP	SP4/WP42.6	**Document Reference**	LiveLink
Related Deliverable(s)	D42.1-D42.5	**Dissemination Level**	PU
Lead Participant	DTU	**Lead Author**	Omar Almousa(DTU)
Contributors	Sebastian Mödersheim (DTU Moritz Horsch(TUD) Paolo Modesti (UNEW) Tobias Wich(ECS) Omar Almousa(DTU)	**Reviewers**	Jaap-Henk Hoepman(RU) Tobias Wich(ECS)

This document is issued within the frame and for the purpose of the *FutureID* project. This project has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318424.

1 Executive Summary

We introduce the APS files for some selected protocols that are highly relevant to the electronic identity systems (eID) in general and FutureID in particular. The selected protocols are: (1) the Extended Access Control protocol (EAC) that is used in the European e-passports to protect sensible data. (2) The Password Authenticated Connection Establishment protocol (PACE) that establishes a secure channel between an eID card chip and a terminal. PACE is the recommended first step in EAC. (3) The Transport Security Layer protocol (TLS) that is one of the most widely used protocols to establish secure channels between a client and a server. (4) A group of ISO standard authentication protocols (ISO 9798).

First we give in a glance the Authentication Protocol Specification language (APS) including its connections to formal verification back-end tools, namely Proverif and OFMC. Then we present each of our selected protocols in a separate section but following the same structure for all protocols, i.e., for each of the selected protocols, we introduce it briefly, then explain in details its specification in APS. After that, we discuss the protocol *formats* (how data is structured in a protocol). Finally, we give the formal verification results for that protocol.

2 Document information

2.1 Contributors

Name	Partner
Sebastian Mödersheim	DTU
Moritz Horsch	TUD
Tobias Wich	ECS
Paolo Modesti	UNEW
Omar Almousa	DTU

2.2 History

0.1	2015-03-24	Omar Almousa	1st Draft Outline
0.2	2015-4-25	Omar Almousa	sections drafts
0.3	2015-4- 30	Sebastian Mödersheim	Introduction
0.4	2015-4-30	Paolo Modesti	ISO9798
0.5	2015-5-6	Moritz Horsch	PACE formats and adapt spec. accordingly
0.6	2015-5-8	Sebastian Mödersheim	Feedback
0.7	2015-5-11	Tobias Wich	EAC formats and adapt spec. accordingly
0.8	2015-5-11	Paolo Modesti	Conclusion
0.9	2015-5-11	Omar Almousa	Formal verification results and finalize
1.0	2015-5-18	Omar Almousa	Reviewers comments

2.3 Table of Contents

2.4 List of Tables

3 Introduction

APS stands for *Authentication Protocol Specification language* and it is a formal language designed within the FutureID project in WP42: Universal Authentication Service. The language is based on the popular informal Alice-and-Bob style notation (aka. protocol narration) but with a precise formal semantics, including a specification of assumptions (like initial knowledge) and security goals. The details can be found in our project deliverable D42.3 [15] and we will give below only a short summary of the language so that the rest is hopefully self-explanatory.

The role of APS in the FutureID project is to have a common ground for specifying protocols that is usable for several purposes:

- It is a simple and intuitive way to communicate protocols and discuss several possible alternatives.
- Using a translator we can directly turn them into executable code in JavaScript. In fact, many people may argue that JavaScript is not a very "secure" language due to, for instance, its untyped nature. However we show that our translator generates very robust implementations, e.g., not suffering from type-flaw attacks or buffer overflows, if one only implements the formats appropriately (as explained below), so that the use of JavaScript does not pose an inherent threat to security at all.
- We can also translate APS into the applied π-calculus and the AVISPA Intermediate Format, the input languages used by two major verification tools (ProVerif and AVISPA/AVANTSSAR). This helps to quickly detect and fix vulnerabilities in protocols and then verify that the final result does not have an attack in a reasonable formal model. Note that the formal model is indeed very close to the actual implementation generated by the translator, as both are obtained from the same "abstract" code representation called *operational strands*.

As writing an APS specification is not too difficult, we have done this for quite a number of classical authentication protocols, also to test our tool chain that the known vulnerabilities are indeed detected by the formal verification tools. They are summarized in Appendix A. In this deliverable, we discuss in detail only a selected number of protocols that are of particular relevance for the project:

EAC The Extended Access Control protocol [16] that is used in the European e-Passports to protect sensible data like fingerprints and other biometric features. It is also used in the German identity card system to provide mutual authentication between the card and a remote terminal.

PACE The Password Authenticated Connection Establishment protocol [16, 17] is the first step before running EAC. It establishes a secure channel between an eID card chip and a terminal based on a previously shared password.

TLS The Transport Security Layer protocol [11, 12], which is one of the most widely used protocols to establish secure channels, and of course used in several places in FutureID as one possible way to establish an encrypted connection between two parties [14].

ISO 9798 A number of ISO standard authentication protocols presented in [1–6].

There are two further, larger, applications of APS that are still ongoing and out of scope for this deliverable as they cannot be done completely within APS:

- As part of WP12, we perform an analysis of security and privacy properties of the FutureID system and its components. It turns out that many aspects can indeed be formalized in APS, namely the general exchange prescribed by the architecture, when abstracting from the concrete credential technologies (such as X.509 or SAML bearer assertions) and when using an abstract concept of channels (e.g., avoiding a detailed model of TLS and rather assume its properties).
- Related to that and WP24, we currently slightly extend APS to support compositional reasoning better. For instance one needs to declare long-term secrets and long-term public values to facilitate several of our compositionality results, and we are developing also tool support for compositional reasoning based on APS specifications. In fact, we can already show that many of the APS specifications we have at this point meet the sufficient conditions for compositionality.

The rest of this report is structured as follows: In Section 4, we briefly summarize the APS language. Section 5 gives the presentation structure for our selected protocols. Sections 6, 7, 8, and 9 are dedicated for the selected protocols in this order: EAC, PACE, TLS, and ISO 9798. We conclude in Section 10.

4 APS in a Nutshell

The Authentication Protocol Specification language (APS) is a formal language that describes protocols in the Alice-and-Bob style notation. It also allows to specify the security goals (like secrecy and authentication) and the specification assumptions of protocols (like what agents know initially).

Now, we give in short how protocols are specified in APS in order to help the reader in the later sections (when we explain the selected protocols). Let us first informally[1] clarify what a message is in APS. A message in APS is either an *atomic* message or a *composed* message. Each atomic message is given a type of the following set of types $\{\text{Number}, \text{Agent}, \text{PublicKey}, \text{PrivateKey}, \text{SymmetricKey}\}$. Atomic messages of type Agent represent the agents of a protocol. Agents with names that start with a capital letter represent *roles* that can be played by honest or dishonest agents, whereas agents with names that start with a small letter represent honest agents. We also use this convention of capital/small letters with the rest of atomic messages (whose type is not Agent): identifiers with capital letters represent freshly-generated values (short-term values) and we call them *variables*, and identifiers with small letters represent long-term values and we call them *constants*.

The composed messages are the result of the application of any of the APS *operators* to other messages (atomic or composed). APS operators include (1) the *cryptographic operators* that represent the basic cryptographic operations in protocols e.g., the symmetric encryption scrypt and the hash function hash, those operations can be performed by all agents, i.e., if an agent knows a message m then he knows the hash of that message: $\mathsf{hash}(m)$. These operators are *public* as they are available to all agents. One last thing about these operators is that they have some deduction rules that govern them, e.g., reflecting that if an agent knows an encrypted message $\mathsf{scrypt}(k, m)$ and the key k, then he can get the message m. The full details of operators and their properties are found in [15]. The rest of the APS operators are either (2) *mappings*, or (3) *formats* that we now discuss in order.

A mapping represents a relationship between objects, i.e., to model the shared key between two agents A and B, we use $\mathsf{shk}(A, B)$ where shk maps a pair of agents to a symmetric key. The concept of mappings is very crucial for the formal models when we want to verify a protocol for several sessions. Mappings are not public (unlike the cryptographic operators), i.e., if an agent knows A and B then he cannot construct $\mathsf{shk}(A, B)$, he either receives it during a protocol run or has it initially by the protocol specifications. Here the instantiation of roles for each of the sessions must be typically different, i.e., A is played by a concrete honest agent in one session, but by the intruder (who we call i) in another session. For example, consider a role A that knows initially a shared key with another role B: $\mathsf{shk}(A, B)$. In the session that i plays the role of A and b plays the role of B then all occurrences of A and B are instantiated with i and b respectively, and so the intruder will know initially $\mathsf{shk}(\mathsf{i}, b)$; meaning that the intruder in this session has a shared key with b which is completely fine to model dishonest agents, but he never gets $\mathsf{shk}(a, b)$ in another session where A is instantiated with the concrete agent a. If we use a constant instead (to model the mapping $\mathsf{shk}(A, B)$), then it is sufficient for the intruder

[1] the formal definition of APS including its message model is found in [15]

to get that constant in one session and use it in all other sessions. Moreover, if we use any public operator instead, say $\mathsf{hash}(A, B)$, then it is sufficient for the intruder to know A and B to get their shared secret. Finally, we do not model that agents share variables initially, as variables must be created during a protocol run and are not initially known. One last thing about mappings is that they are not real operations, i.e., in real implementations, there is no such function that computes the shared key for two agents given their names. Thus mappings will not appear in the implementation.

Now we proceed with formats, the last class of operators in APS. In the formal model, a protocol format is a public operator that represents how a protocol, in the implementation, structures a certain message (XML-tags, fixed-lengths, etc.). We use formats to reflect that protocols structure their messages in various ways, we declare different formats and use them (to compose messages) as public operators in the formal model. For the implementation side, formats are the key to allow for implementations that are interoperable with existing standards, because we can integrate any desired mechanism for structuring, like modern XML-based formats in EAC, as well as classic formats with tagging and length fields like in TLS, or even combinations of modern and classic formats. For each format in a protocol, we require a Java class that basically implements a parser and pretty-printer for it (i.e., serializer and de-serializer in protocol implementation slang). In the formal model, we handle formats as public *transparent* functions; meaning that all agents can construct messages with formats, and all agents can get the fields of any formatted message (without the need for a key), e.g., consider a format f, if an agent knows m_1 and m_2 then he can construct $\mathsf{f}(m_1, m_2)$, and if he knows $\mathsf{f}(m_3, m_4)$ then he also knows m_3 and m_4. A proper implementation of formats that avoids type-flaws and buffer overflows, is the guarantee for a robust implementation generated by the APS compiler.

After we explained how messages are constructed in APS, now we discuss how protocols are specified. In APS, a protocol specification is given by several sections, such that each of these sectios specifies an aspect about the protocol, i.e., the **Actions** section specifies how agents exchange messages, the **Knowledge** section specifies the initial knowledge for each of the protocol participants, and the **Goals** section specifies what goals the protocol aims to achieve. The list of APS specification sections is as follows:

Protocol In this section we give a name for the protocol, e.g., PACE.

Types In this section we declare all atomic messages of a protocol (except fresh-values that are declared in the **Actions** section as we will see shortly). These messages are declared with types from the set of types: {Number, Agent, PublicKey, PrivateKey, SymmetricKey} as we explained earlier. For example, we say: `Agent A,B;` to declare that a protocol has two agents A and B. The reader can refer to the coming protocol specifications in Listings 1,9, and 11 for more detailed examples.

Formats In this section, we give the abstract terms of the protocol formats. For example, if a protocol uses a certificate structure that includes an agent name and his public key, then in APS we abstract away the fine details of how this certificate is structured (XML-tags or fixed-lengths) and say in the **Formats** section: `certificate(Agent,PublicKey);`. After this declaration, we can use it in the rest of the APS specification as a public operator.

Knowledge In this section we give the *initial knowledge* for each of the protocol *participants*; a participant is an agent involved in the message exchange of a protocol. Note that not all agents declared in the **Types** section are participants, for example a certificate authority that issues certificates for other agents but never engaged in message exchanging is not a participant. The initial knowledge of participants is driven from the protocol assumption, e.g., a participant has a public/private key pair or two participants share a secret key. Participants use the messages that they have in their initial knowledge to compose the messages they send or to decompose the messages they receive.

Actions In this section we specify what are the messages exchanged among protocol participants, and what data is freshly created during a protocol run. If a participant A creates a fresh public-key (and call it X), then we say: `A : PublicKey X`, then if A wants to send this key to B then we say: `A` $\rightarrow$ `B : X`, where the $\rightarrow$ denotes an insecure channel between A and B. We support different types of channels in APS including secure, confidential and authentic channels that we will explain later when they occur in the coming specifications. Finally, the **Actions** section must follow a token-passing order, i.e., the receiver of the previous message is the sender of the next one.

Goals In this section we specify the goals that the protocol aims at achieving. In APS, we support several protocol goals including secrecy and authentication.

4.1 APS Connections

The APS compiler enables the formal verification of protocols via the connection to several back-end tools such as the static analysis tool Proverif [9,10] and the model checker OFMC [20]. These connections are made possible by the translations to Applied-π [7] and AVISPA IF [8] language. Next we give in a glance some background about both.

4.2 Applied-π for Proverif

Proverif allows for the verification of protocols for an unbounded number of sessions. Proverif uses abstraction and produces sound security proofs, i.e., the absence of attacks in the given protocol model. However, the abstraction may lead to false positive attacks due to the over-approximation, i.e., some attacks may be unrealistic. Proverif accepts as an input specifications in applied-π that our APS compiler translates to.

4.3 AVISPA IF for OFMC

OFMC, on the other hand, allows for the verification of protocols for a bounded number of sessions, but without abstraction, i.e., OFMC tells if a given protocol is secure for a bounded number of sessions (e.g., protocol A is secure for 3 sessions), and OFMC found attacks are not the result of an over-approximation. One of the input languages of OFMC is the AVISPA IF languages that our compiler translates to.

5 Case Studies Structure

In order to improve the readability of this report, we follow the same structure in presenting the selected protocols. We introduce each protocol in a separate section that has the following subsections:

- An introduction that gives a brief informal description of the protocol.
- The APS specification of the protocol, in which we explain how we modeled the different aspects of the protocol, why we made certain design choices, and what limitations we have. As a natural choice, we present the protocol's APS specification following its structure as explained in the previous Section 4.
- The protocol formats, in which discuss the message formats of the protocol.
- The analysis results, in which we discuss the formal verification part of the protocol. We first present the most important aspects of the auto-generated code (we give the full version of this code in the appendix). Then we summarize the results obtained from running the back-end tools (ProVerif and OFMC) on the auto-generated code of the protocol.

6 EAC

The Extended Access Control (EAC) protocol is used in ePassports to secure sensitive data like fingerprints. EAC is a two-party protocol that aims to provide a mutual authentication between an eID card and a terminal. The protocol takes place in the environment of the German ID card. Through EAC (1) a terminal (Proximity Coupling Device, PCD) authenticates itself to the ID card (Proximity Integrated Circuit Card, PICC) to get access to the personal data stored on the card and (2) the PICC proves its authenticity to the PCD. In brief, EAC works as follows: (1) PCD sends his certificate (issued by a certificate authority) to PICC, (2)PICC replies with a freshly generated random number say R_1. Then, (3) PCD sends back his Diffie-Hellman half key, and (4) PICC replies with another freshly generated random number say R_2. Now, (5) PCD signs R_1 and his Diffie-Hellman half key, and sends the signature to PICC. Finally, (6) PICC sends his Diffie-Hellman half key signed by a certificate authority that PCD accepts. Both PICC and PCD now can construct a shared authenticated key.

6.1 APS

Now we give the complete specification of EAC followed by a detailed explanation for this specification.

```
Protocol: EAC
Types:
 Agent PCD, PICC, ca;
 Number g, certDesc, rChat, oChat, auxData, noCert, cHAT, cAR, eFC, idPICC, null,
    handle,  didName;
Formats:
 eac1input(Msg, Msg, Msg, Number, Number, Number, Number);
 eac1output(Number, Number, Number, Number, Number);
 eac2input(Msg, Msg, Number, Number, Number);
 eac2output(Number, Msg, Msg, Number);
 eacadditionalinput(Msg, Msg, Msg);
 certForm(Agent, PublicKey);
 x59d(Agent, Number, PublicKey);
Knowledge:
 PCD:  PCD, sign(inv(pk(ca)),certForm(PCD,pk(PCD))), pk(PCD), pk(ca),
    inv(pk(PCD)), g, handle, didName, certDesc, rChat, oChat, auxData, noCert, null;
 PICC: PICC, sign(inv(pk(ca)),certForm(PICC,exp(g,sk(PICC)))), pk(ca),
    sk(PICC), g, cHAT, cAR, eFC, idPICC, null;
  where PCD!=ca, PICC !=ca;
Actions:
 [PCD]*->*[PICC]: eac1input(handle,  didName, sign(inv(pk(ca)), certForm(PCD, pk(PCD))),
      certDesc, rChat, oChat, auxData)

 PICC: Number  RC
 [PICC]*->*[PCD]: eac1output(cHAT, cAR, eFC, idPICC, RC)

 PCD: Number X
 let PK_PCD=exp(g,X)
 [PCD]*->*[PICC]: eac2input(handle,  didName, noCert, PK_PCD, null)

 PICC: Number Rpicc
 [PICC]*->*[PCD]: eac2output(Rpicc, null, null, null)
```

```
[PCD]*->*[PICC]: eacadditionalinput(handle,  didName, sign(inv(pk(PCD)), x59d(PICC,
    Rpicc, PK_PCD)))

PICC: Number Rmac
let PK_PICC=exp(g,sk(PICC))
let DHKey= exp(PK_PCD,sk(PICC))
[PICC]*->*[PCD]: eac2output(null, sign(inv(pk(ca)),certForm(PICC,PK_PICC)), mac(hash(
    DHKey, Rmac), PK_PCD), Rmac)

Goals:
PICC authenticates PCD on DHKey
PCD authenticates PICC on DHKey
DHKey secret of PICC, PCD
```

Listing 1: EAC in APS

Protocol Here we give the name of the protocol, EAC.

Types In EAC, we have three agents: (1) `PCD` represents a terminal (Proximity Coupling Device), (2) `PICC` represents an eID card (Proximity Integrated Circuit Card), and (3) `ca` that represents a trusted certificate authority that issued certificates for the other two agents. We also declare a number `g` to represent a base to use it later for Diffie-Hellman key agreement. We assume that g is a primitive root modulo some prime p that both agents agreed upon; we later use $\mathsf{exp}(g, X)$ to denote g to the power X modulo p. Finally, we declare several identifiers like `certDesc`, `rChat`, `oChat`, `eFC`. Those are used later by the agents as parameters in their exchanged messages. We explain them when they appear in the messages in the **Actions** section.

Formats EAC has several formats, e.g., the format `eac1input` represents an XML structuring for seven fields: a certificate and six numeric values that describe the certificate and provide auxiliary information; we describe these six fields below when `eac1input` is used in the protocol message exchange. Note that in this section we only name each protocol format and specify the data types for its fields, the details of the format structure are abstracted away and left for the implementation (a Java class for each of the protocol formats). More about the formats of EAC is given in Section 6.2 that is dedicated to this purpose.

Knowledge In this section we give the initial knowledge for each of the participants of EAC:

`PCD` knows initially his name, a certificate issued by the certificate authority `ca` in which `ca` signs the public key of `PCD`. In EAC a certificate chain may be used instead of a single one, but in our specification we model them as a single certificate as it is a realistic abstraction. `PCD` also knows his public and private keys, the public key of `ca`, and the Diffie-Hellman base g. Additionally, `PCD` knows `handle`, `didName`, `certDesc`, `rChat`, `oChat`, `auxData`, `noCert` and `null` that he uses later as parameters in his messages, we explain them when they appear in the exchanged messages in the **Actions** section.

`PICC` knows initially his name, a certificate issued by the certificate authority `ca` in which the half key of `PICC` is signed. `PICC` also knows the public key of `ca`, his secret key

sk(PICC), and finally the base g. Finally, PICC knows cHAT, cAR, eFC, idPICC and null; he uses these constants as parameters in his messages, we explain them when they appear in the exchanged messages in the **Actions** section.

Actions This section specifies what messages are exchanged among protocol participants as well as what data is created freshly during an EAC run.

1. PCD initiates the EAC protocol; he sends a handle handle and a didName that are both used to identify him. PCD also sends his certificate signed by ca along with a certificate description certDesc, a required and an optional Certificate Holder Authorization Template (CHAT) represented here with rCHAT and oCHAT, and some auxiliary data auxData. PCD wraps this data using the eac1input format before he sends them to PICC.

   In the original specification of the EAC protocol [16], PCD sends a certificate chain, in here we use a single certificate to model the chain. Finally, PCD sends this message over a *secure pseudonymous channel* that we denote by $[PCD]* \rightarrow *[PICC]$. By secure we mean that the intruder can not get the exchanged messages over this channel, and by pseudonymous we mean that they both do not know each other. Due to the lack of tool support, we simulate this channel between any two agents A and B (that do not know each other), by: A sends first a fresh public-key to B, then B replies with a fresh symmetric key encrypted with the public-key of A. After this, they can both exchange message securely using the symmetric key to encrypt all exchanged messages among them; still without knowing each other. More about secure pseudonymous channels can be found in [21]. This channel is the result of performing the PACE protocol step before EAC (we discuss PACE protocol in Section 7). This channels is used in all the steps of EAC.
2. After receiving the first message from PCD, PICC verifies the received certificate and extracts the public key of PCD. Then, PICC generates a random number (challenge) RC and sends it along with his identifier idPICC, a Certificate Holder Authorization Template cHAT, a Certification Authority Reference cAR, and eFC. The message has the format eac1output. Note that this message is dedicated to PCD as he is the only one able to decrypt it.
3. PCD then computes an ephemeral Diffie-Hellman half-key by generating the nonce X. Then he send his half key and the parameter noCert to PICC both formatted with eac2input. The let is an in-line macro used to improve readability.
4. PICC generates the random number Rpicc and send it to PCD formatted with eac2output.
5. Now PCD signs (with his private key) his name, the received random number Rpicc and his half key exp(g, X), then sends them to PICC. null is used to model the null value for any optional parameter. Note that the signed message is formatted with the format x59d and the whole message is formatted with eac2additionalinput.
6. Finally, PICC performs checks on the values of PCD half-key and Rpicc. Then he generates the random number Rmac, and computes: mac(hash(sk($PICC$), PK_PCD, Rmac), PK_PCD) where mac is a keyed hash, hash is a hash function, sk($PICC$) is his long-term secret, PK_PCD is the ephemeral half-key of PCD the PICC received previously, i.e., exp(g, X).

After this computation, PICC sends the certificate issued by ca that signs his static half-key along with the mac of above and the random number Rmac. Note that PCD can compute the full Deffie-Hellman key using the half-key of PICC and X that he previously generated, thanks to the algebraic properties of the modular exponentiation exp. Note that the users of APS do not have to specify how agents decrypt the received messages nor how they are verified (if a message is of a certain format); these details are defined in the semantics of APS (defined in [15]) and implemented by APS compiler.

Goals Here we specify the goals of the EAC protocol. EAC aims at achieving mutual authentication between its two participants, PCD and PICC. The goals are explained later in the verification tools section (to check if EAC achieves them or not). We discuss the verification of EAC in Section 6.3.

6.2 Formats

The EAC protocol has several formats for its XML-messages as shown in Listings 2- 6. Note that the messages are slightly simplified, i.e., Name-spaces and organizational XML attributes were left out.

```
EAC1INPUT(handle, did-name, certs, cert-desc, req-chat, opt-chat, aux-data, transact) =
<DIDAuthenticate>
  handle
  <DIDName>did-name</DIDName>
  <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
    {certs: c | <Certificate>c</Certificate>}
    <CertificateDescription>cert-desc</CertificateDescription>
    {if req-chat: <RequiredCHAT>req-chat</RequiredCHAT>}
    {if opt-chat: <OptionalCHAT>res-chat</OptionalCHAT>}
    {if aux-data: <AuthenticatedAuxiliaryData>aux-data</AuthenticatedAuxiliaryData>}
    {if transact: <TransactionInfo>transact</TransactionInfo>}
  </AuthenticationProtocolData>
</DIDAuthenticate>
```

Listing 2: EAC1INPUT format

```
EAC1OUTPUT(chat, cars, ef-ca, id-picc, challenge) =
<DIDAuthenticateResponse>
  <Result>
    <ResultMajor>http://www.bsi.bund.de/ecard/api/1.1/resultmajor#ok</ResultMajor>
  </Result>
  <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
    {if chat: <CertificateHolderAuthorizationTemplate>chat</
        CertificateHolderAuthorizationTemplate>}
    {cars: c | <CertificationAuthorityReference>c</CertificationAuthorityReference>}
    <EFCardAccess>ef-ca</EFCardAccess>
    <IDPICC>id-picc</IDPICC>
    <Challenge>challenge</Challenge>
  </AuthenticationProtocolData>
</DIDAuthenticateResponse>
```

Listing 3: EAC1OUTPUT format

```
EAC2INPUT(handle, did-name, certs, key, sig) =
<DIDAuthenticate>
  handle
  <DIDName>did-name</DIDName>
  <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
    {certs: c | <Certificate>c</Certificate>}
    <EphemeralPublicKey>key</EphemeralPublicKey>
    {if sig: <Signature>sig</Signature>}
  </AuthenticationProtocolData>
</DIDAuthenticate>
```

Listing 4: EAC2INPUT format

```
EAC2OUTPUT(ef-cs, token, nonce, challenge) =
<DIDAuthenticateResponse>
  <Result>
    <ResultMajor>http://www.bsi.bund.de/ecard/api/1.1/resultmajor#ok</ResultMajor>
  </Result>
  <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
    {if challenge: <Challenge>challenge</Challenge>
     else:
    <EFCardSecurity>ef-cs</EFCardSecurity>
    <AuthenticationToken>token</AuthenticationToken>
    <Nonce>nonce</Nonce>}
  </AuthenticationProtocolData>
</DIDAuthenticateResponse>
```

Listing 5: EAC2OUTPUT format

```
EACADDITIONALINPUT(handle, did-name, sig) =
<DIDAuthenticate>
  handle
  <DIDName>did-name</DIDName>
  <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
    <Signature>sig</Signature>
  </AuthenticationProtocolData>
</DIDAuthenticate>
```

Listing 6: EACADDITIONALINPUT format

For each of the formats, the APS compiler provides a Java class skeleton. Here we give as an example the class for `eac1input` format, in which we have a parser, a pretty printer and other methods of `eac1input` format.

```
//Auto-generated by APS
//Format class prototype
 public class eac1input{
     private byte[] field1;
     private byte[] field2;
     private byte[] field3;
     private byte[] field4;
     private byte[] field5;
     public eac1input( byte [] input){//insert your format parser code
    }
     public eac1input(byte[] x1, byte[] x2, byte[] x3, byte[] x4, byte[] x5){//insert
         your format setter here
```

```
        field1=x1;
        field2=x2;
        field3=x3;
        field4=x4;
        field5=x5;
    }
     public boolean verify(){
     //insert your verifier code here
    }
    public byte[] get1(){
     //insert your getter code here
     return field1;
    }
     public byte[] get2(){
     //insert your getter code here
     return field2;
    }
     public byte[] get3(){
     //insert your getter code here
     return field3;
    }
     public byte[] get4(){
     //insert your getter code here
     return field4;
    }
     public byte[] get5(){
     //insert your getter code here
     return field5;
    }
    public byte[] encode(){//insert your format pretty-printer code here
    }
}
```

Listing 7: eac1input Class

As shown Listing 7 we have the following methods:

- Two constructors, one that parses a byte array into an `eac1input` object (this corresponds to a deserializer), and another that constructs such an object from the given fields values.
- A `verify` method to check whether a message is a valid `eac1input` object.
- An `encode` method that returns the object in concrete syntax, also referred as a serializer or a pretty-printer.
- For each element of the object, a `get` method that returns that element.

6.3 Analysis Results

Here we present the formal verification aspects and results of EAC, i.e., Proverif proofs or/and OFMC attacks or bounded verification. We use EAC as our key example, i.e., we discuss in details different aspects of the analysis of this protocol. However, in the other protocols we only point out the differences if they exists, in order to avoid any redundancy.

6.3.1 Proverif

Using the APS compiler, we translated the EAC specification into an applied-π code. The generated code is shown in B.1 and is composed from three main parts. The first part is the *declarations* part in which we declare all identifiers and functions used in the rest of the code. Listing 8 shows a chunk of this part, where one can find the declaration of the following:

- A channel `ch` (Line 1) that agents use to exchange message.
- The intruder name (Line 2).
- The used mappings like inv (Line 3) and their only property of being `private`, i.e., agents (including the intruder) cannot use it to compose messages.
- The used functions such as the symmetric encryption function scrypt (Line 3) followed by its decryption property; namely that if the key is known then the message is also known. Proverif considers all functions to be public unless they are followed by `[private]` as in the mapping inv, thus another property of scrypt is that it is public (all agents including the intruder can use it to compose messages).
- The used formats, e.g., `certform` (Line 6) followed its properties (Lines 7 and 8), i.e., if one has a formatted message then he can get the fields of that formats (decompose the format). Formats are also public.
- Finally, the goals. (Line 10) is a query for Proverif to answer if the attacker can get the enclosed secret `expexpgxskpiccPICCPCD` that is declared to be private in the previous line.

```
1  free ch: channel.
2  free i:bitstring.
3  fun inv(bitstring):bitstring [private].
4  fun scrypt(bitstring,bitstring):bitstring.
5   reduc forall k: bitstring, m: bitstring;  sdecrypt(k,scrypt(k,m))=m.
6  fun certform(bitstring,bitstring):bitstring.
7   reduc forall a:bitstring,b:bitstring;certformget1(certform(a,b))=a.
8   reduc forall a:bitstring,b:bitstring;certformget2(certform(a,b))=b.
9
10 free expexpgxskpiccPICCPCD:bitstring[private].
11 query attacker(expexpgxskpiccPICCPCD).
```

Listing 8: Declaration part of EAC in applied-π

The second part The second part is the processes of the EAC participants, i.e., `PCD` and `PICC`. These processes contain all actions each participant performs such as sending and receiving messages, performing checks and what event he issues. The events are used later to verify goals accordingly.

The third part is the "global" process that instantiates the above processes with their initial knowledges and allows the intruder to instantiate some of these process to model the dishonest agents bahavior. One important feature here is that our generated code verifies EAC for unbounded number of session and agents. The summary of Proverif analysis of EAC is shown in Table 1

Table 1: Proverif Analysis Summary for EAC

Goal of EAC	Proverif result
Secrecy Goal `DHKey secret of PICC, PCD`	Proof found
Authentication Goals `PCD authenticates PICC on DHKey`	Proof found
`PICC authenticates PCD on DHKey`	Attack found

6.3.2 OFMC

APS compiler also generates AVISPA IF specifications that can be checked with the OFMC. The summary of the OFMC results for EAC are shown in Table 2.

Table 2: OFMC Analysis Summary for EAC

Goal of EAC	OFMC result
Secrecy Goal `DHKey secret of PICC, PCD`	No attack found
Authentication Goals `PCD authenticates PICC on DHKey`	No attack found
`PICC authenticates PCD on DHKey`	Attack found

As shown in the Tables 1 and 2, the back-end tools agreed that `PICC` can not authenticate `PCD` on the shared key `DHKey`. This attack is a Man-in-the-Middle attack that is well-know flaw in Diffie-Hellman key agreement, but it is not considered as a serious attack since it happens only if the intruder can impersonate the `PCD` to the `PICC` and vice-versa, which is only possible if intruder controls the `PCD`. In case we assumed the intruder cannot control the `PCD` then we do not have such an attack. This assumption is realistic since it holds a certified from the trusted third party `ca`.

7 PACE

The Password Authenticated Connection Establishment (PACE) [13] protocol establishes a strong session key for secure communication based on a shared, weak secret used for authentication. PACE is used by the German identity card (i.e. `PICC`) to (1) protect the communication over the contactless interface between the card and the terminal (i.e. `PCD`) and (2) authenticate the legitimate card holder using a PIN. It was invented by the German Federal Office for Information Security as a replacement for the insecure Basic Access Control (BAC).

In brief, PACE works as follows: (1) The card user `User` enters the PIN into the card reader `PCD`. (2) `PCD` requests the card `PICC` to send him a freshly generated random number encrypted with the PIN. (3) `PICC` creates a random number, encrypts it with the shared secret (i.e. PIN), and sends it to the `PCD`. (4) `PCD` decrypts received message to get the random number, creates an ephemeral key pair, and sends the public key to the `PICC`. (5) `PICC` also generates an ephemeral key pair and responses with the public key. (6) Both perform a Diffie-Hellman key agreement and compute a common shared secret. (7) Both compute a new Diffie-Hellman generator based on the common shared secret and the random number and perform an additional Diffie-Hellman key agreement using the new generator. Finally, (8) both derive session keys for encryption and authentication and create an authentication token to authentication each other and verify the session keys. In case of a wrong shared secret, i.e. a wrong PIN, that verification fails. The channel between *User* and `PCD` is a pseudonymous secure channel, i.e., they can exchange messages securely without knowing each other (relying on a pseudonym instead of their real names). For the lack of tool support we model this channel by: first, `User` generates a public/private key pair (we only model the generation of the public part and the private part is implicit, i.e., if an agent creates a public key, then the private key is added to his knowledge implicitly), then he sends his key and the PIN to `PCD` encrypted with the public key of `PCD`. We also add a step in the end of the protocol to enable the check authentication between `PICC` and `User`.

Please note that the `PICC` and `PCD` exchange information about key sizes, encryption and hash algorithms, Diffie-Hellman generators, and so forth beforehand; and therefore we include this information in their initial knowledges as shown shortly.

7.1 APS

In this section we present the PACE protocol in APS (cf. Listing 9) and describe it in detail.

```
1  Protocol: PACE
2  Types:
3   Agent   PCD, PICC, User;
4   Number g, three, oid, secretid, null;
5
6  Formats:
7   mseSetATRequest(Msg, Msg);
8   mseSetATResponse(Msg);
9   encryptedNonceRequest(Number);
10  encryptedNonceResponse(Number);
11  mapNonceRequest(PublicKey);
```

```
12  mapNonceResponse(PublicKey);
13  performKeyAgreementRequest(PublicKey);
14  performKeyAgreementResponse(PublicKey);
15  mutualAuthenticationRequest(Number);
16  mutualAuthenticationResponse(Number);
17
18 Knowledge:
19  PICC: PICC, User, g, three, shk(User,PICC), oid, secretid, null;
20  PCD:  PCD, g, three, oid, secretid, pk(PCD), inv(pk(PCD)), null;
21  User: User, PICC, PCD, shk(User,PICC), pk(PCD), null;
22  where PCD != i;
23
24 Actions:
25  User: PublicKey PkA
26  User -> PCD: crypt(pk(PCD), (shk(User, PICC), PkA))
27
28  PCD -> PICC: mseSetATRequest(oid, secretid)
29
30  PICC -> PCD: mseSetATResponse(null)
31
32  PCD -> PICC: encryptedNonceRequest(null)
33
34  PICC: Number R
35  PICC -> PCD: encryptedNonceResponse(scrypt(shk(User,PICC), R))
36
37  PCD: Number X1
38  PCD -> PICC: mapNonceRequest(mult(X1,g))
39
40  PICC: Number Y1
41  PICC -> PCD: mapNonceResponse(mult(Y1,g))
42
43  PCD: Number X2
44
45  let PCD_K1 = mult(mult(Y1,g), X1)
46  let PCD_g2 = add(mult(R,g), PCD_K1)
47  PCD -> PICC: performKeyAgreementRequest(mult(X2, PCD_g2))
48
49  PICC: Number Y2
50
51  let PICC_K1 = mult(mult(X1,g), Y1)
52  let PICC_g2 = add(mult(R,g), PICC_K1)
53  PICC -> PCD: performKeyAgreementResponse(mult(Y2, PICC_g2))
54
55  let PCD_K2 = mult(X2, mult(Y2, PICC_g2))
56  PCD -> PICC: mutualAuthenticationRequest(mac(hash(PCD_K2, three), mult(Y2,PICC_g2)))
57
58  let PICC_K2 = mult(mult(X2, PCD_g2), Y2)
59  let Key = mult(X2, PCD_g2)
60  PICC: Number NPICC
61  PICC-> PCD: mutualAuthenticationResponse(mac(hash(PICC_K2,three),mult(X2, PCD_g2))),
        scrypt(Key ,NPICC)
62
63  PCD-> User: crypt(PkA, (shk(User, PICC), NPICC))
64
65 Goals:
66  Key secret of PCD, PICC
67  NPICC secret of PICC, User
68  User authenticates PICC on NPICC
```

Listing 9: PACE in APS

Protocol Specifies the name of the protocol, PACE.

Types This section specifies the three agents that are involved in the PACE protocol: (1) the `PCD` representing the terminal, (2) the `PICC` representing the ID card, and (3) the user `User`. Furthermore, the section specifies the numbers `g`, `three`, `oid`, `secretid`, and `null`. `g` represents the generator for the Diffie-Hellman key exchange that takes place in PACE and `three` simply represents the number three and is used as a constant in the key derivation. `oid` is the object identifier that tells what algorithms should be used. `secretid` tells what secret should be used, i.e., PIN in our case, but it could be PUK or CAN as well. Finally, *null* is to represent the empty value for optional parameters.

Formats The APS defines various formats for the PACE protocol. The `mseSetAT`[2] is used to initialize the PACE protocol. The `encryptedNonce` formats represent the exchange of the encrypted random number. The `mapNonce` message formats are used for the first Diffie-Hellman key exchange and the `performKeyAgreement` formats for the second key exchange. The `mutualAuthentication` are used to convey the authentication token.

Knowledge This initial knowledge of the participants is as follows:

`PICC` knows his name, the user name `User`, and the constants `g`, `three`, `oid`, and `secretid`. He also has `shk(User, PICC)` a shared key between him and the user `User`; this shared key models the PIN.

`PCD` knows his name, and the constants `g`, `three`, `oid`, and `secretid`. He also has a public/private key pair represented by `pk(PCD)` and `inv(pk(PCD))`.

`User` knows his name name, the names of `PCD` and `PICC`. He also knows the public key of `PCD` and the PIN, i.e., `shk(User, PICC)`. The reason behind the user knowing the public key of `PCD` is to model the secure pseudonym channel between both of them as mentioned earlier.

Actions We explain this APS section step-by-step.

1. The user `User` sends the PIN (`shk(User, PICC)`) to `PCD`. For lack of tool support and to model that `User` and `PCD` are not known to each other, but the user can send the PIN securely to `PCD`; we model the channel between them by: first, the user generates a public/private key pair (we only model the generating of the public part and the private part is implicit), then he sends his key and the PIN to the terminal `PCD` encrypted with the public key of `PCD`.
2. `PCD` derives the PIN (`shk(User, PICC)`) from the received message, then sends the `mseSetAT` message to the `PICC`. The `oid` is an object identifier that specifies the PACE version and cryptographic algorithms that should be used in the protocol run.

[2] Manage Security Environment Set Authentication Template

The `secretid` specifies the type of the secret (PIN, PUK, or CAN), in our case, we assume it is always the PIN.

3. Subsequently, the `PCD` requests an encrypted random number from the `PICC`.
4. The `PICC` generates the random number `R`, encrypts it using the shared key $\mathsf{shk}(\texttt{User}, \texttt{PICC})$, and sends it to the `PCD`.
5. The `PCD` decrypts the message and retrieves `R`. It generates a secret `X1`, computes $\mathsf{mult}(\texttt{X1}, \mathsf{g})$, and sends it to `PICC`. Please note that is step is the first part of the Diffie-Hellman key agreement and that mult represents the modular multiplication.
6. In response, `PICC` generates the secret `Y1` and sends $\mathsf{mult}(\texttt{Y1}, \mathsf{g})$ back.
7. The `PCD` finishes the first key agreement by computing the first Diffie-Helman shared secret `PCD_K1`. Now, `PCD` starts the second key agreement. He computes the new generator `PCD_g2`, generates `X2`, computes $\mathsf{mult}(\texttt{X2}, \mathsf{g2})$, and sends it to `PICC`.
8. The `PICC` acts similar. It computes `PICC_K1`, generates `Y2`, computes $\mathsf{mult}(\texttt{Y2}, \mathsf{g2})$, and sends it to `PCD`.
9. Now both participants can calculate a shared authentication token. First we show how `PCD` does that (in the next step we show it for `PICC`):
   (a) `PCD` calculates a common shared key $\texttt{mult}(\mathsf{mult}(\mathsf{g}, \texttt{X2}), \texttt{Y2})$.
   (b) Then he derives a MAC key using the hash function and the constant three $\mathsf{hash}(\texttt{mult}(\texttt{X2}, \texttt{mult}(\texttt{Y2}, \mathsf{add}(\texttt{mult}(\texttt{R}, \mathsf{g}), \texttt{mult}(\texttt{mult}(\texttt{X1}, \mathsf{g}), \texttt{Y1})))), \texttt{three})$.
10. Finally, `PICC` calculates the shared authentication token in a similar way to what `PCD` did in the last step, i.e.:
    (a) `PICC` calculates a common shared key $\texttt{mult}(\mathsf{mult}(\mathsf{g}, \texttt{X2}), \texttt{Y2})$.
    (b) Then he derives a MAC key using the hash function and the constant three $\mathsf{hash}(\texttt{mult}(\texttt{Y2}, \texttt{mult}(\texttt{X2}, \mathsf{add}(\texttt{mult}(\texttt{R}, \mathsf{g}), \texttt{mult}(\texttt{mult}(\texttt{Y1}, \mathsf{g}), \texttt{X1})))), \texttt{three})$.

Goals The goal of PACE is to establish a secure channel between the `PICC` and the `PCD`. More about PACE goals is found in Section 7.3.

7.2 Formats

This section provides more details about the formats for PACE. The communication between the `PICC` and the `PCD` is based an Application Protocol Data Units (APDU) [18]. The formats for request messages (e.g. `mseSetAT`) as specified in Listing 10 defines the APDU header of 4 bytes followed by the data. In Listing 10 we use (1) $\mathsf{byte}(n)$ to represent one-byte constant n; (2) the $\cdot$ to represent the concatenation operator; and (3) `tlv(tag, value)` to represent the tag-length-value encoding. For example, the formats for response messages may contain data followed by 2 bytes representing the result of the request. A successful request is for instance confirmed by `0x9000`, i.e. $\mathsf{byte}(140) \cdot \mathsf{byte}(0)$.

```
mseSetATRequest(oid, secretID)
        = byte(0) · byte(32) · byte(193) · byte(164) · tlv(byte(128), oid) · tlv(byte(131), secretID)

mseSetATResponse() = byte(140) · byte(0)

encryptedNonceRequest()
        = byte(16) · byte(134) · byte(0) · byte(0) · byte(2) · byte(124) · byte(0) · byte(0)

encryptedNonceResponse(random) = random · byte(140) · byte(0)

mapNonceRequest(X1)
        = byte(16) · byte(134) · byte(0) · byte(0) · tlv(byte(129), tlv(byte(124), X1)) · byte(0)

mapNonceResponse(Y1)
        = tlv(byte(130), tlv(byte(124), Y1)) · byte(140) · byte(0)

performKeyAgreementRequest(X2)
        = byte(16) · byte(134) · byte(0) · byte(0) · tlv(byte(131), tlv(byte(124), X1)) · byte(0)

performKeyAgreementResponse(Y2)
        = byte(16) · byte(134) · byte(0) · byte(0) · tlv(byte(132), tlv(byte(124), Y1)) · byte(0)

mutualAuthenticationRequest(token)
        = byte(16) · byte(134) · byte(0) · byte(0) · tlv(byte(133), tlv(byte(134), token))

mutualAuthenticationResponse(token)
        = byte(16) · byte(134) · byte(0) · byte(0) · tlv(byte(134), tlv(byte(134), token))
```

Listing 10: PACE Formats

7.3 Analysis Results

7.3.1 Proverif

We use the APS compiler to generate applied-π code from PACE specification; so we can use Proverif to verify PACE. Unfortunately, Proverif was not able to verify PACE in a reasonable time, it took several hours and only giving the next message without any noticed progress.

Linear part:
mult(a_26,mult(b_27,c)) = mult(mult(a_26,b_27),c)
mult(a_24,b_25) = mult(b_25,a_24)
Completing equations...

Currently, we are working on a solution for this issue.

7.3.2 OFMC

The results of checking PACE with OFMC is summarized in Table 3.

Table 3: OFMC Analysis Summary for PACE

Goal of PACE	OFMC result
Secrecy	
`Key secret of PCD, PICC`	No attack found for 2 sessions
`NPICC secret of PICC, User`	No attack found for 2 sessions
Authentication	
`User authenticates PICC on NPICC`	No attack found for 2 sessions

8 TLS

The Transport Layer Security (TLS) protocol [11,12] is a widely known protocol that establishes a secure channel between a client and a server. TLS is widely used in many applications including eID systems; several European eID systems use it for authentication, e.g., the Italian and Swiss eID systems [14].

8.1 APS

Here we present TLS specification in APS based on [19]

```
Protocol: TLS
Types:
 Agent A,B,ca;
 Number eps,cipher,compr,t20,t22,t23;
Formats:
 clientHello(Msg, Msg, Msg, Msg, Msg);
 serverHello(Msg, Msg, Msg, Msg, Msg);
 serverCert(Msg);
 serverHelloDone(Msg);
 clientKeyExchage(Msg);
 finished(Msg);
 pmsForm(Msg);
 masterForm(Msg,Msg);
 clientFinished(Msg,Msg,Msg);
 serverFinished(Msg,Msg,Msg);
 keyBlock(Msg, Msg);
 changeCipher(Msg);
    ####
 record(Number, Msg);
 x509(Agent, PublicKey);

Knowledge:
 A: A,B,pk(ca),eps,cipher,compr, shk(A,B), t20,t22,t23;
 B: B, pk(B),pk(ca),inv(pk(B)), sign(inv(pk(ca)),x509(B,pk(B))),eps,cipher,compr, shk(A,
     B), t20,t22,t23;
Actions:
# A generates Ra and T1
 A: Number Ra, T1
 let AM1= record(t22, clientHello(T1,Ra,eps,cipher,compr))
```

```
29 A->B: AM1
30 #B generates Rb, Id T2 and
31 B: Number Rb, T2, Id
32 let BM1=record(t22, serverHello(T2,Rb,Id,cipher,compr)),
33         record(t22, serverCert(sign(inv(pk(ca)),x509(B,pk(B)))),
34         record(t22, serverHelloDone(eps))
35 B->A: BM1
36 #A checks the certificate for a TTP ca
37 #A extracts the public key of B
38 #A generates the Pre-Mster Secret PMS.
39 #A computes MS=PRF(mster-form(PMS;RA + RB))
40 let MS = prf(masterForm(PMS,add(Ra,Rb)))
41 A: Number PMS
42 let AM2= record(t22, clientKeyExchage(crypt(pk(B), pmsForm(PMS)))),
43         record(t20, changeCipher(eps)),
44         record(t22, finished(prf(clientFinished(MS,add(Ra,Rb),hash(AM1, BM1)))))
45 A->B: AM2
46 B->A: record(t20, changeCipher(eps)), record(t22, finished(prf(serverFinished(MS,add(Ra
       ,Rb),hash(AM1, BM1, AM2)))))
47 #A computes the key clntK=extractCK(key block(MS;RA + RB))
48 #A and B exchange payload messages as follows:
49 A: Number PAYLOADA
50 A->B: record(t23, scrypt(extCK(keyBlock(MS, add(Ra,Rb))), shk(A,B)))
51 #B computes srvrK=extractSK(key block(MS;RA + RB))
52 B: Number PAYLOADB
53 B->A: record(t23, scrypt(extSK(keyBlock(MS, add(Ra,Rb))), PAYLOADB))
54
55 Goals:
56 MS secret of A,B
57 B authenticates A on MS
```

Listing 11: TLS in APS

Now we explain the TLS specification given above by section.

Protocol In this section we give the name of the protocol, TLS.

Types In TLS, we have three agents: (1) `A` represents a client, (2) `B` represents a server, and (3) `ca` represents a trusted certificate authority that issued a certificate for the server `B`. We also declare the cipher suite `cipher`, the compression algorithm `compr`, the empty string `eps`, and the tags `t20, t22` and `t23`. We later explain how agents use them.

Formats TLS has several formats that structure its messages. For instance, the format `helloClient` has five fields of type Msg: a time-stamp, a freshly generated number, a session-id, cipher suites and compression methods. Later we provide more details about TLS formats.

Knowledge The initial knowledge for each of the participants of TLS is as follows:

- `A` knows initially his name, the name of `B`, and the public key of the trusted third party `ca`. She also knows the cipher suite `cipher`, the compression algorithm `compr` and the empty string `eps`. Finally she knows the tags `t20, t22` and `t23`.
- `B` knows initially his name, the name of `A`, his private and public keys and the public key of the trusted third party `ca` (certificate authority). He also has a certificate issued for

him by ca and he knows the cipher suite cipher, the compression algorithm compr and the empty string eps. Finally he knows the tags t20, t22 and t23.

Actions In this section, we specify the exchange messages between protocol participants as well as the freshly generated data during a TLS execution.

1. The client A initiates the protocol; she generates a fresh random number Ra and a time-stamp T1 and sends them to the server B along with the emptystring eps, her preferences for encryption cipher, and her preferences for compression algorithms compr. She formats this data with the format clientHello, and envelopes this message with the format record. The tag t22 denotes that this message belongs to the handshake sub-protocol. We refer to the whole message by AM1 for later reference.
2. The server B, in response, generates the fresh random number Rb, the time-stamp T2, and the session-id Id. Then with the received cipher, and compr he formats them using the serverHello format. He also sends his certificate that has the x509 format and enveloped with serverCert format. (This certificate is issued by the trusted third party ca.) Each of these messages (the serverHello, serverCert, and the serverHelloDone) is enveloped with the format record then sent to the client A. The tag t22 denotes that this message belongs to the TLS-handshake sub-protocol. We refer to this whole message by BM1 for later reference as well.
3. Now, A checks the certificate issued by ca and extracts the public key of B. Then he generates the Pre-Master Secret PMS and computes $\mathtt{MS} = \mathtt{prf}(\mathtt{masterForm}(\mathtt{PMS}, \mathtt{add}(\mathtt{Ra}, \mathtt{Rb})))$ using the pseudo-random function prf. He encrypts MS with the public key of B and formats it with clientKeyExchange.
   Then each of: (1) this message, (2) the change-cipher message $\mathtt{changeCipher}(\mathtt{eps})$, and (3) the message $\mathtt{finished}(\mathtt{prf}(\mathtt{clientFinished}(\mathtt{MS}, \mathtt{add}(\mathtt{Ra}, \mathtt{Rb}), \mathtt{hash}(\mathtt{AM1}, \mathtt{BM1}))))$; is enveloped with record and sent to B. Note that A included the hashes of the previous messages AM1 and BM1. Also note that the finished format is nesting another format, namely clientFininshed, and that the tag of the second message (2) is t20 denoting that it belongs to the TLS sub-protocol: change-cipher specification.
4. B computes the pseudo-random function of $\mathtt{serverFinished}(\mathtt{MS}, \mathtt{add}(\mathtt{Ra}, \mathtt{Rb}))$ and the hash of all previous messages, i.e., AM1, BM1 and AM2. He wraps the computed value with the format finished and sends it to A as his finished message.
5. A computes the key $\mathtt{extCK}(\mathtt{keyBlock}(\mathtt{MS}, \mathtt{add}(\mathtt{Ra}, \mathtt{Rb})))$ using the client extract-key function extCK and uses it to encrypt the payload message PAYLOADA, then sends the encrypted message to B. Here (and in the next) the record message is tagged with t23 to denote that it belongs to the TLS sub-protocol: application data.
6. B also computes the key $\mathtt{extSK}(\mathtt{keyBlock}(\mathtt{MS}, \mathtt{add}(\mathtt{Ra}, \mathtt{Rb})))$ using the server extract-key function extCK and uses it to encrypt the payload message PAYLOADB, then sends the encrypted message to A. Note that both extCK and extSK are publicly known functions, so both the client and the server can compute the keys of the encrypted messages sent to them.

Goals TLS aims at establishing a secure channel between A and B so they can securely exchange payloads. The goals are explained later in the analysis results of TLS (Section 8.3).

8.2 Formats

The formats used in the TLS protocol as specified in [19] are shown in Table 4. In this table we use the following notation:

- $\mathsf{byte}(n)$ denotes one-byte constant n, e.g., $\mathsf{byte}(2)$ means the constant 2 represented in a byte.
- $\cdot$ denotes the concatenation operator, e.g., $\mathsf{byte}(3) \cdot \epsilon$ means the concatenation of the constant 3 with the empty string ϵ.
- $\mathsf{off}_n(data) \cdot data$ means an offset (with a fixed length of n bytes) that tells how many bytes the following *data* is supposed to be, e.g., $\mathsf{off}_3(comp\_methods) \cdot comp\_methods \cdot comp\_methods$ means that the number presented in the first 3 bytes tells what is the length of the following *comp_methods*.

For example, consider the very first format of Table 4, `clientHello` that has five fields: a time-stamp, a freshly generated random number, a session-id, cipher suites and compression methods. This format is structured as follows:

- Three bytes with the values $1, 3$ and 3, where the later two bytes indicate the TLS version number, namely 1.2.
- A fixed-length field for the time-stamp.
- A fixed-length field for the random number.
- One-byte offset that tells the length of the following field reserved for the session-id.
- Two-byte offset that tells the length of the following field reserved for the cipher suites.
- One-byte offset that tells the length of the following field reserved for the compression methods.

The Java class skeleton for the `clientHello` format is given in Listing 13.

```
1  public class clientHello{
2      private byte[] field1;
3      private byte[] field2;
4      private byte[] field3;
5      private byte[] field4;
6      private byte[] field5;
7      public clientHello( byte [] input){
8      //insert your format parser code
9     }
10     public clientHello(byte[] x1, byte[] x2, byte[] x3, byte[] x4, byte[] x5){
11     //insert your format setter here
12         field1=x1;
13         field2=x2;
14         field3=x3;
15         field4=x4;
```

Table 4: TLS formats

$\texttt{clientHello}(time, random, session\_id, cipher\_suites, comp\_methods)$
$= \text{HANDSHAKE}(\mathsf{byte}(1), \mathsf{byte}(3) \cdot \mathsf{byte}(3) \cdot time \cdot random \cdot \mathsf{off}_1(session\_id) \cdot session\_id \cdot \mathsf{off}_2(cipher\_suites) \cdot cipher\_suites \cdot \mathsf{off}_1(comp\_methods) \cdot comp\_methods)$

$\texttt{serverHello}(time, random, session\_id, chosen\_cipher, chosen\_comp)$
$= \text{HANDSHAKE}(\mathsf{byte}(2), \mathsf{byte}(3) \cdot \mathsf{byte}(3) \cdot time \cdot random \cdot \mathsf{off}_1(session\_id) \cdot session\_id \cdot chosen\_cipher \cdot chosen\_comp)$

$\texttt{serverCert}(certificate\_tls\_vec)$
$= \text{HANDSHAKE}(\mathsf{byte}(11), \mathsf{off}_3(certificate\_tls\_vec) \cdot certificate\_tls\_vec)$

$\texttt{serverHelloDone}() = \text{HANDSHAKE}(\mathsf{byte}(14), \epsilon)$

$\texttt{clientKeyExchange}(EncrPreMasterSecret) = \text{HANDSHAKE}(\mathsf{byte}(16), EncrPreMasterSecret)$

$\texttt{finished}(encr\_finished) = \text{HANDSHAKE}(\mathsf{byte}(20), encr\_finished)$

$\texttt{pmsForm}(secret) = \mathsf{byte}(3) \cdot \mathsf{byte}(3) \cdot secret$

$\texttt{masterForm}(PMS, R) = PMS \cdot \texttt{"master secret"} \cdot R$

$\texttt{clientFinished}(MS, R, H) = MS \cdot \texttt{"client finished"} \cdot R \cdot H$

$\texttt{serverFinished}(MS, R, H) = MS \cdot \texttt{"server finished"} \cdot R \cdot H$

$\texttt{keyBlock}(MS, R) = MS \cdot \texttt{"key expansion"} \cdot R$

$\texttt{changeCipher}() = \mathsf{byte}(1)$
$\texttt{certRequest}(cert\_type, supp\_alg, cert\_auths)$
$= \text{HANDSHAKE}(\mathsf{byte}(13), \mathsf{off}_1(cert\_type) \cdot cert\_type \cdot$

$\texttt{clientCert}(certificate\_tls\_vec)$
$= \text{HANDSHAKE}(\mathsf{byte}(11), \mathsf{off}_3(certificate\_tls\_vec) \cdot certificate\_tls\_vec)$

$\texttt{certVerify}(signed\_handshake) = \text{HANDSHAKE}(\mathsf{byte}(15), signed\_handshake)$

$\text{RECORD}(sub, data) = sub \cdot byte(3) \cdot byte(3) \cdot offco2data \cdot data$

```
16          field5=x5;
17      }
18       public boolean  verify(){
19       //insert your verifier code here
20      }
21       public byte[] get1(){
22       //insert your getter code here
23       return field1;
24      }
25       public byte[] get2(){
26       //insert your getter code here
27       return field2;
28      }
29       public byte[] get3(){
30       //insert your getter code here
31       return field3;
32      }
33       public byte[] get4(){
34       //insert your getter code here
35       return field4;
36      }
37       public byte[] get5(){
38       //insert your getter code here
39       return field5;
40      }
41      public byte[] encode(){
42      //insert your format pretty-printer code here
43          }
44 }
```

Listing 12: clientHello Class

TLS with client authentication We also consider another version of TLS with client authentication. For brevity, we only point out the difference with the previously detailed version of TLS. In this version, the client `A` has a certificate issued by the trusted third party `ca` that he sends to the server `B` in the third step of the protocol in response to a certificate request from the server in the step before. One further difference between the two versions of TLS is that the later one (TLS with client authentication) we can achieve an extra goal, namely that the client `A` can be authenticated to the server `B`. It is well know that this goals cannot be achieved in TLS without client certificate.

,

```
1 Protocol: TLS-CA #with client certificate
2 Types:
3  Agent A,B,ca;
4  Number eps,cipher,compr,t20,t22,t23;
5 Formats:
6  clientHello(Msg, Msg, Msg, Msg, Msg);
7  serverHello(Msg, Msg, Msg, Msg, Msg);
8  serverCert(Msg);
9  serverHelloDone(Msg);
10 clientKeyExchage(Msg);
11 finished(Msg);
12 pmsForm(Msg);
13 masterForm(Msg,Msg);
```

```
14  clientFinished(Msg,Msg,Msg);
15  serverFinished(Msg,Msg,Msg);
16  keyBlock(Msg, Msg);
17  changeCipher(Msg);
18  certRequest(Msg, Msg, Msg);
19  clientCert(Msg);
20  certVerify(Msg)
21     ###
22  record(Number, Msg);
23  x509(Agent, PublicKey);
24
25 Knowledge:
26  A: A,B,pk(A),pk(ca),inv(pk(A)), sign(inv(pk(ca)),x509(A,pk(A))),eps,cipher,compr, t20,
        t22,t23;
27  B: B, pk(B),pk(ca),inv(pk(B)), sign(inv(pk(ca)),x509(B,pk(B))),eps,cipher,compr,t20,t22
        ,t23, certType, suppAlg, certAuth;
28
29 Actions:
30 # A generates Ra and T1
31  A: Number Ra, T1
32  let AM1= record(t22, clientHello(T1,Ra,eps,cipher,compr))
33  A->B: AM1
34 #B generates RB T2 and Id
35  B: Number Rb, Id, T2
36  let BM1= record(t22, serverHello(T2,Rb,Id,cipher,compr)),
37          record(t22, serverCert(sign(inv(pk(ca)),x509(B,pk(B)))),
38          record(t22, certRequest(certType,suppAlg,certAuth)),
39          record(serverHelloDone(eps))
40  B->A: BM1
41 #A checks the certificate for a TTP ca
42 #A extracts the public key of B
43 #A generates the Pre-Mster Secret PMS.
44 #A computes MS=PRF(mster-form(PMS;RA + RB))
45 # Here in certVerify(eps), eps replaces signed_hanshake
46  let MS = prf(masterForm(PMS,add(Ra,Rb)))
47  A: Number PMS
48  let AM2= record(t22, clientCert(sign(inv(pk(ca)),x509(A,pk(A)))),
49          record(t22, clientKeyExchage(crypt(pk(B), pmsForm(PMS)))),
50          record(t22, certVerify(eps)),
51          record(t20, changeCipher(eps)),
52          record(t22, finished(prf(clientFinished(MS,add(Ra,Rb),hash(AM1, BM1)))))
53  A->B: AM2
54
55  B->A: record(t20, changeCipher(eps)), record(t22, finished(prf(serverFinished(MS,add(Ra
        ,Rb),hash(AM1, BM1, AM2)))))
56 #A computes the key clntK=extractCK(key block(MS;RA + RB))
57 #A and B exchange payload messages as follows:
58  A: Number PAYLOADA
59  A->B: record(t23, scrypt(extCK(keyBlock(MS, add(Ra,Rb))), PAYLOADA))
60 #B computes srvrK=extractSK(key block(MS;RA + RB))
61  B: Number PAYLOADB
62  B->A: record(t23, scrypt(extSK(keyBlock(MS, add(Ra,Rb))), PAYLOADB))
63 Goals:
64  MS secret of A,B
65  A authenticates B on MS
66  B authenticates A on MS
```

Listing 13: TLS with Client authentication in APS

8.3 Analysis Results

As aforementioned, we provide in this section the formal verification results that we obtained from running the back-end verification tools on our auto-generated code from the APS compiler.

8.3.1 Proverif

Now we show the results obtained from Proverif. We first translated the APS specifications of TLS into applied-π code and we get the results summarized in Table 5.

Table 5: Proverif Analysis Summary for TLS

Goal of TLS	Proverif result no client auth.	Proverif result with client auth.
Secrecy `MS secret of A,B`	Proof found	Proof found

Authentication
`A authenticates B on MS`
`B authenticates A on MS`
	—
	Proof found
		Proof found
		Proof found

8.3.2 OFMC

Here we show the results from running the OFMC tool on the AVISPA IF code generated from the APS files listed above.

Table 6: OFMC Analysis Summary for TLS

Goal of TLS	OFMC result no client auth. (for 2 sessions)	OFMC result with client auth. (for 2 sessions)
Secrecy `MS secret of A,B`	No attack found	No attack found

Authentication
`A authenticates B on MS`
`B authenticates A on MS`
	—
	No attack found
		No attack found
		No attack found

9 ISO/IEC 9798-4

The ISO/IEC 9798 Standard specifies a family of entity authentication protocols. It comprises six main documents. The Part 1 [4] is general and describes the main notions which are common for the other parts. The protocols are grouped into five parts. Part 2 [2] describes six protocols using symmetric encryption, Part 3 [5] seven protocols using digital signatures, Part 4 [1] four protocols using cryptographic check functions such as MACs, Part 5 [3] considers protocols using zero knowledge and Part 6 [6] eight entity authentication mechanisms based on manual data transfer between authenticating devices techniques.

9.1 APS

Part 4 [1] describes two mechanisms that are concerned with the authentication of a single entity (unilateral authentication), while the remaining are mechanisms for mutual authentication of two entities. The mechanisms specified in this part of ISO/IEC 9798 use time variant parameters such as time stamps, sequence numbers, or random numbers, to prevent valid authentication information from being accepted at a later time or more than once. If a time stamp or sequence number is used, one pass is needed for unilateral authentication, while two passes are needed to achieve mutual authentication. If a challenge and response method employing random numbers is used, two passes are needed for unilateral authentication, while three passes are required to achieve mutual authentication.

In this section we consider only the two unilateral mechanism (the first and the second protocols described in the document[3]) denoted ISO9798-4.1 and ISO9798-4.2. In these authentication mechanisms the entities A and B shall share a common secret authentication key or two unidirectional secret keys prior to the commencement of any particular run of the authentication mechanisms.

The use of the text fields specified in the following mechanisms is outside the scope of this part of ISO/IEC 9798 (they may be even empty), and will depend upon the specific application. A

[3]Since the protocols do not have a name, we identify them by the order of appearance in the document

text field may only be included in the input to the cryptographic check function if the verifier can determine it independently, e.g., if it is known in advance, sent in clear or can be derived from one or both of those sources.

Here we present the first protocol of the ISO 9797-4 specification in APS based on [1]

```
Protocol: ISOCCFOnePassUnilateralAuth
# ISO9798-4.1

Types:
    Agent A,B;
    Function hash;
Formats:
    tokenAB979841 (Number,Number,Msg);
    # Agent is optional in the standard
    fkab(Number,Agent,Number);
    farg(SymmetricKey,Msg);
    fm1(Msg,Agent,Number);

Knowledge:
    A: A,B,shk(A,B);
    B: B,A,shk(A,B);
Actions:
    A: Number NA, Text1, Text2
    A->B: fm1(tokenAB979841(NA,Text2,hash(farg(shk(A,B),fkab(NA,B,Text1)))),B,Text1)

Goals:
    B authenticates A on Text1

Private:
    shk(A,B)
```

Listing 14: 9798-4.1 in APS

Now we explain the protocol specification given above by section.

Protocol In this section we give the name of the protocol, ISOCCFOnePassUnilateralAuth.

Types Here we declare the protocol identifiers and annotate them with types. Those identifiers include the agents of the protocol: a claimant `A` and a verifier `B`. We also declare a function `hash` which is used as a cryptographic check function. As defined in ISO/IEC 9798-1, `hash`(`K`, `X`) denotes the cryptographic check value computed by applying the cryptographic check function `hash` to the data `X` using the key `K`.

Formats Formats are detailed in 9.2.

Knowledge In this section we give the initial knowledge for each of the protocol *participants*; a participants is an agent involved in the message exchanging in a protocol. The participants use the terms/messages in their initial knowledge to compose the message they send or to decompose the messages they receive. Now we give the initial for each participants:

- `A` `A` knows initially his name, the name of `B`, and `shk`(`A`, `B`), a secret symmetric key pre-shared between the two agents.

`B` `B` knows initially his name, the name of `A`, and shk(A,B), a secret symmetric key pre-shared between the two agents.

Actions This section specifies what messages are exchanged among protocol participants as well as what data is created freshly during a protocol run.

1. `A` generates a nonce `NA`[4] and optionally two text fields `Text1` and `Text2`. Then `A` computes and sends `TokenAB979841` to `B`. On receipt of the message containing `TokenAB979841`, `B` verifies `TokenAB979841` by checking the sequence number, comparing it with the cryptographic check value of the token, thereby verifying the correctness of the distinguishing identifier `B`, if present.

Goals here we specify the goals of the protocol. ISO 9797-4.1 aims at achieving unilateral authentication between the two participants, the claimant `A` and the verifier `B`.

Here we present the second protocol of the ISO 9797-4 specification in APS based on [1].

```
Protocol: ISOCCFTwoPassUnilateralAuth
# ISO9897-4.2

Types:
    Agent A,B;
    Function hash;
Formats:
    tokenAB979842 (Number,Msg);
    # Agent is optional in the standard
    fkab(Number,Agent,Number);
    farg(SymmetricKey,Msg);
    fab(Number,Number);
    fm2(Msg,Number);

Knowledge:
    A: A,B,shk(A,B);
    B: B,A,shk(A,B);

Actions:
    B: Number NB,Text1
    B->A: fab(NB,Text1)
    A: Number Text3,Text2
    A->B: fm2(tokenAB979842(Text3,hash(farg(shk(A,B),fkab(NB,B,Text2)))),Text2)

Goals:
    B authenticates A on Text2

Private:
    shk(A,B)
```

Listing 15: 9798-4.2 in APS

The main difference of the two-step protocol, with respect to the one-step version, is the use of a random number instead of a sequence number (or a time stamp). Hence, there is the need of an extra step in the protocol run, to implement the challenge-response mechanism.

[4] The standard also considers the possibility of using time stamps

9.2 Formats

As defined in ISO/IEC 9798-1, when the result of concatenating two or more data items is an input to a cryptographic check function as part of one of the mechanisms specified in this part of ISO/IEC 9798, this result shall be composed so that it can be uniquely resolved into its constituent data strings, i.e. so that there is no possibility of ambiguity in interpretation. Since the actual implementation of the formats is application specific, here we do not give a precise description of the formats as in the previous examples but we simply assume that the following formats, and the associated constructor and destructor, satisfy the aforementioned property. The formats for ISO9798-4.1 are:

- `tokenAB979841`: this format models the authentication token. It is a concatenation of three fields; namely a nonce, a text, and a hashed-value. According to the standard, the parsing of these fields of this format(and all other formats) must unambiguous. Therefore, a possible way to achieve that is using fixed-length fields. Following is `tokenAB979841` represented in the notation for formats that we used earlier in PACE and TLS:

$$\texttt{tokenAB979841}(\texttt{nonce}, \texttt{text}, \texttt{hashed}) = \texttt{nonce} \cdot \texttt{text} \cdot \texttt{hashed}$$

- `fkab`: this format models a concatenation of three fields. Based on the standards requirement on formats (must be unambiguous), we can present this format as follows:

$$\texttt{fkab}(\texttt{nonce}, \texttt{agent}, \texttt{text}) = \texttt{nonce} \cdot \texttt{agent} \cdot \texttt{text}$$

- `farg`: this format models the argument of the hash function. It structures two fields: a symmetric key, and a message formatted with `fkab`. Following is its structure:

$$\texttt{farg}(\texttt{key}, \texttt{msg}) = \texttt{key} \cdot \texttt{msg}$$

- `fm1`: this format models the data packet sent over the network, it has three fields: a message formatted with `tokenAB979841`, an agent name, and a text, as follows:

$$\texttt{fm1}(\texttt{msg}, \texttt{agent}, \texttt{text}) = \texttt{msg} \cdot \texttt{agent} \cdot \texttt{text}$$

The formats for ISO9798-4.2 may be implemented using the same style of encoding.

9.3 Analysis Results

Here we present the formal verification aspects and results of these protocols. i.e., Proverif proofs or/and OFMC attacks or bounded verification.

9.3.1 Proverif

Using the APS compiler, we translated the ISO9798-4.1 and ISO9798-4.2 specification into an applied-π code. The summary of Proverif analysis of these protocols is shown in Table 7.

Table 7: Proverif Analysis Summary for ISO9798-4

Goal of ISO9798-4.1	Proverif result
Authentication Goals `B authenticates A on Text1`	Proof found

Goal of ISO9798-4.2	Proverif result
Authentication Goals `B authenticates A on Text2`	Proof found

9.3.2 OFMC

APS compiler also generates AVISPA IF specifications that can be checked with the OFMC. The summary of the OFMC results for ISO9798-4 (1,2) are shown in Table 8.

Table 8: Proverif Analysis Summary for ISO9798-4

Goal of ISO9798-4.1	OFMC result
Authentication Goals `B authenticates A on Text1`	No attack found

Goal of ISO9798-4.2	OFMC result
Authentication Goals `B authenticates A on Text2`	No attack found

10 Conclusion

In this deliverable we have presented the APS files for some selected protocols that are highly relevant to the electronic identity systems (eID) in general and FutureID in particular. The selected protocols are: (1) the Extended Access Control protocol (EAC), (2) The Password Authenticated Connection Establishment protocol (PACE), (3) The Transport Security Layer protocol (TLS) and (4) a selection of ISO 9798-4 authentication protocols.

By means of these examples, we have shown the effectiveness of our approach. In fact, we were able to encode in APS a significant class of real-world protocols relevant to eID systems, to precisely define the message formats employed in these protocols, and to formally verify them by means of two state of the art verification tools (Proverif and OFMC).

Future work may involve encoding new protocols along with integrating and consolidating the results of this deliverables with work carried out in WP24 (compositional reasoning) and WP12 (analysis of security and privacy properties).

List of References

[1] ISO/IEC 9798-4:1999. Information technology – Security techniques – Entity authentication – Part 4: Mechanisms using a cryptographic check function, 1999.

[2] ISO/IEC 9798-2:2008. Information technology – Security techniques – Entity authentication – Part 2: Mechanisms using symmetric encipherment algorithms, 2008.

[3] ISO/IEC 9798-5:2009. Information technology – Security techniques – Entity authentication – Part 5: Mechanisms using zero-knowledge techniques, 2009.

[4] ISO/IEC 9798-1:2010. Information technology – Security techniques – Entity authentication – Part 1: General, 2010.

[5] ISO/IEC 9798-3:1998/Amd 1:2010. Information technology – Security techniques – Entity authentication – Part 3: Mechanisms using digital signature techniques, 2010.

[6] ISO/IEC 9798-6:2010. Information technology – Security techniques – Entity authentication – Part 6: Mechanisms using manual data transfer, 2010.

[7] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.

[8] AVISPA Project. Deliverable 2.3: The Intermediate Format. Available at `www.avispa-project.org/publications.html`, 2003.

[9] B. Blanchet et al. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 82, 2001.

[10] B. Blanchet and B. Smyth. ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2011.

[11] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol, Version 1.2, 2008.

[12] D. Eastlake. RFC6066: Transport Layer Security (TLS) Extensions: Extension Definitions, 2011.

[13] Federal Office for Information Security. Advanced Security Mechanism for Machine Readable Travel Documents - Extended Access Control (EAC), Password Authenticated Connection Establishment (PACE), and Restricted Identification (RI). Technical Guideline BSI-TR-03110, Version 2.10, Part 1 - 3, 2012. `https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html`.

[14] FutureID Project. Deliverable D32.1: Survey and Analysis of Existing eID and Credential Systems , 2013.

[15] FutureID Project. Deliverable D42.3: Future AnB: The projected APS Language of FutureID, 2013.

[16] German Federal Office for Information Security (BSI). Advanced Security Mechanism for Machine Readable Travel Documents- Extended Access Control (EAC), Password Authenticated Connection Establishment (PACE), and Restricted Identification (RI) , 2008. Available at `www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110`.

[17] International Civil Aviation Organization(ICAO). Machine Readable Travel Documents, 2015.

[18] International Organization for Standardization). Identification cards - Integrated circuit cards - Part 1-15. International Standard.

[19] S. Mödersheim and G. Katsoris. A sound abstraction of the parsing problem. In *CSF*, pages 259–273. IEEE, 2014.

[20] S. Mödersheim and L. Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. *Foundations of Security Analysis and Design V*, pages 166–194, 2009.

[21] S. Mödersheim and L. Viganò. Secure pseudonymous channels. *ESORICS 2009*, pages 337–354, 2009.

A Formal Verification Summary

Here we summarize the results of running the formal verification tools (Proverif and OFMC) on the auto-generated code from our APS complier. The protocol test suite (that we used to test our compiler) consists of over 30 protocols. The suite includes not only many protocols from the Clark/Jacob library but also many other widely used like Kerberos, h530, and protocols defined by the ISO 9798 standard. As a result for the test, the tools were able to find all known attacks in the suite. Tables 9 and 10 summarize these results (a protocol with a star means that it is one of our case studies).

Table 9: Formal Verification Summary for Protocols Specified in APS - Part 1

Protocol	**Proverif**	**OFMC**
Basic-Kerberos	No Attack	No Attack
EPMO	Attack found	Attack found
⋆ EAC-PCDAuth	No Attack	No Attack
⋆ EAC-Secrecy	No Attack	No Attack
⋆ EAC-PICCAuth	Attack found	Attack found
⋆ PACE	No Attack	No Attack
IKEv2-DS-PayloadSecrecy	No Attack	No Attack
IKEv2-DS-PayloadAuthB	No Attack	No Attack
IKEv2-DS-KeySecrecy	No Attack	No Attack
IKEv2-DS-KeyAuthB	No Attack	No Attack
AndrewSecureRPCSecrecy	No Attack	No Attack
ISOsymKeyOnePassUnilateralAuthProt	No Attack	No Attack
ISOsymKeyThreePassMutual	No Attack	No Attack
ISOsymKeyTwoPassMutualAuthProt-Corr	No Attack	No Attack
ISOsymKeyTwoPassUnilateralAuthProt	No Attack	No Attack
NonReversible	No Attack	No Attack
⋆ISO9798-4.2	No Attack	No Attack
ISOCCFThreePassMutual	No Attack	No Attack
⋆ISO9798-4.2	No Attack	No Attack

Table 10: Formal Verification Summary for Protocols Specified in APS - Part 2

Protocol	Proverif	OFMC
h530	Attack found	Attack found
h530-fix	No Attack	No Attack
SSO	Attack found	Attack found
⋆ tls-noClientAuth	No Attack	No Attack
⋆ tls	No Attack	No Attack
AndrewSecureRPC	Attack found	Attack found
Needham-Schroeder	Attack found	Attack found
Needham-Schroeder-Lowe	No Attack	No Attack
Amended-NSCK	No Attack	No Attack
Denning-Sacco-TimeStamp	No Attack	No Attack
Denning-Sacco-Corr	No Attack	No Attack
NSCK	No Attack	No Attack
ISOpubKeyOnePassUnilateralAuthProt	No Attack	No Attack
ISOpubKeyTwoPassMutualAuthProt-CORR	No Attack	No Attack
ISOpubKeyTwoPassUnilateralAuthProt	No Attack	No Attack

B Auto-generated Applied-π code for the selected protocols

B.1 EAC

```
(*Auto-Generated by APS2Java*)
free ch: channel.
free i:bitstring.
fun inv(bitstring):bitstring [private].
fun sign(bitstring,bitstring):bitstring.
    reduc forall k: bitstring, m: bitstring; open(k,sign(inv(k),m))=m.
fun pk(bitstring):bitstring[private].
fun certform(bitstring,bitstring):bitstring.
    reduc forall a:bitstring,b:bitstring;certformget1(certform(a,b))=a.
    reduc forall a:bitstring,b:bitstring;certformget2(certform(a,b))=b.
const g: bitstring.
fun exp(bitstring,bitstring):bitstring.
    equation forall  x: bitstring, y: bitstring; exp(exp(g,x),y)=exp(exp(g,y),x).
fun sk(bitstring):bitstring [private].
fun eac1input(bitstring,bitstring):bitstring.
    reduc forall a:bitstring,b:bitstring;eac1inputget1(eac1input(a,b))=a.
    reduc forall a:bitstring,b:bitstring;eac1inputget2(eac1input(a,b))=b.
fun eac2input(bitstring,bitstring):bitstring.
    reduc forall a:bitstring,b:bitstring;eac2inputget1(eac2input(a,b))=a.
    reduc forall a:bitstring,b:bitstring;eac2inputget2(eac2input(a,b))=b.
fun eac2additionalinput(bitstring):bitstring.
    reduc forall a:bitstring;eac2additionalinputget1(eac2additionalinput(a))=a.
fun x59d(bitstring,bitstring,bitstring):bitstring.
```

```
24      reduc forall a:bitstring,b:bitstring,c:bitstring;x59dget1(x59d(a,b,c))=a.
25      reduc forall a:bitstring,b:bitstring,c:bitstring;x59dget2(x59d(a,b,c))=b.
26      reduc forall a:bitstring,b:bitstring,c:bitstring;x59dget3(x59d(a,b,c))=c.
27  fun eac1output(bitstring,bitstring,bitstring):bitstring.
28      reduc forall a:bitstring,b:bitstring,c:bitstring;eac1outputget1(eac1output(a,b,c))=a
            .
29      reduc forall a:bitstring,b:bitstring,c:bitstring;eac1outputget2(eac1output(a,b,c))=b
            .
30      reduc forall a:bitstring,b:bitstring,c:bitstring;eac1outputget3(eac1output(a,b,c))=c
            .
31  fun eac2output(bitstring):bitstring.
32      reduc forall a:bitstring;eac2outputget1(eac2output(a))=a.
33  fun eac22output(bitstring,bitstring,bitstring):bitstring.
34      reduc forall a:bitstring,b:bitstring,c:bitstring;eac22outputget1(eac22output(a,b,c))
            =a.
35      reduc forall a:bitstring,b:bitstring,c:bitstring;eac22outputget2(eac22output(a,b,c))
            =b.
36      reduc forall a:bitstring,b:bitstring,c:bitstring;eac22outputget3(eac22output(a,b,c))
            =c.
37  fun mac(bitstring,bitstring):bitstring.
38  fun hash(bitstring):bitstring.
39  fun scrypt(bitstring,bitstring):bitstring.
40      reduc forall k: bitstring, m: bitstring;  sdecrypt(k,scrypt(k,m))=m.
41
42  (* END OF FUNCSansPROPS*)
43
44  fun secCh(bitstring,bitstring): channel[private].
45  fun confCh(bitstring): channel[private].
46  fun authCh(bitstring): channel.
47  free hashskpiccexpgxrmacPICCPCD:bitstring[private].
48
49  query attacker(hashskpiccexpgxrmacPICCPCD).
50
51  (*Agent PCD process in protocol:EAC  is *)
52  let processPCD(x1:bitstring,x2:bitstring,x3:bitstring,x4:bitstring,x5:bitstring,x6:
        bitstring,x7:bitstring) =
53  new x8:bitstring;
54
55  out(secCh(x2,x1), eac1input(x3, x8));
56  in(secCh(x1,x2), x9:bitstring);
57  let x10:bitstring = eac1outputget1(x9) in
58  let x11:bitstring = eac1outputget2(x9) in
59  let x12:bitstring = eac1outputget3(x9) in
60  if(x11 = x1) then
61  new x13:bitstring;
62
63  new x14:bitstring;
64
65  out(secCh(x2,x1), eac2input(x14, exp(x7, x13)));
66  in(secCh(x1,x2), x15:bitstring);
67  let x16:bitstring = eac2outputget1(x15) in
68  out(secCh(x2,x11), eac2additionalinput(sign(x6, x59d(x11, x16, exp(x7, x13)))));
69  in(secCh(x11,x2), x17:bitstring);
70  let x18:bitstring = eac22outputget1(x17) in
71  let x19:bitstring = eac22outputget2(x17) in
72  let x20:bitstring = eac22outputget3(x17) in
73  let x21:bitstring = open(x5, x18) in
74  let x22:bitstring = certformget1(x21) in
```

```
let x23:bitstring = certformget2(x21) in
if(x22 = x11) then
if(x23 = exp(x7, sk(x11))) then
if(x19 = mac(hash((exp(exp(x7, x13), sk(x11)), x20)),(exp(x7, x13)))) then
if(x1<> i) then
out(ch, scrypt(hash((sk(x1), exp(x7, x13), x20)),hashskpiccexpgxrmacPICCPCD));out(ch,
    scrypt(hash((sk(x1), exp(x7, x13), x20)),hashskpiccexpgxrmacPICCPCD));0.
(* -------------------------- *)
        (*EOP*)
(*Agent PICC process in protocol:EAC  is *)
let processPICC(x1:bitstring,x2:bitstring,x3:bitstring,x4:bitstring,x5:bitstring,x6:
    bitstring) =
in(secCh(x1,x2), x7:bitstring);
let x8:bitstring = eac1inputget1(x7) in
let x9:bitstring = eac1inputget2(x7) in
let x10:bitstring = open(x4, x8) in
let x11:bitstring = certformget1(x10) in
let x12:bitstring = certformget2(x10) in
if(x11 = x1) then
new x13:bitstring;
new x14:bitstring;

out(secCh(x2,x1), eac1output(x14, x2, x13));
in(secCh(x1,x2), x15:bitstring);
let x16:bitstring = eac2inputget1(x15) in
let x17:bitstring = eac2inputget2(x15) in
new x18:bitstring;

out(secCh(x2,x11), eac2output(x18));
in(secCh(x11,x2), x19:bitstring);
let x20:bitstring = eac2additionalinputget1(x19) in
let x21:bitstring = open(x12, x20) in
let x22:bitstring = x59dget1(x21) in
let x23:bitstring = x59dget2(x21) in
let x24:bitstring = x59dget3(x21) in
if(x22 = x2) then
if(x23 = x18) then
if(x24 = x17) then
new x25:bitstring;

out(secCh(x2,x1), eac22output(x3, mac(hash((exp(x17, x5), x25)),(x17)), x25));
if(x1<> i) then
out(ch, scrypt(hash((x5, x17, x25)),hashskpiccexpgxrmacPICCPCD));out(ch, scrypt(hash((x5
    , x17, x25)),hashskpiccexpgxrmacPICCPCD));0.
(* -------------------------- *)
        (*EOP*)

process
new ca:bitstring;

!new x:bitstring;
out(ch, x)|
!in(ch,(picc:bitstring));
processPCD(picc, x, sign(inv(pk(ca)), certform(x, pk(x))), pk(x), pk(ca), inv(pk(x)), g)
    |
out(ch,(picc, i, sign(inv(pk(ca)), certform(i, pk(i))), pk(i), pk(ca), inv(pk(i)), g))|
!in(ch,(pcd:bitstring));
processPICC(pcd, x, sign(inv(pk(ca)), certform(x, exp(g, sk(x)))), pk(ca), sk(x), g) |
```

```
129 out(ch,(pcd, i, sign(inv(pk(ca)), certform(i, exp(g, sk(i)))), pk(ca), sk(i), g))
```

Listing 16: Applied-π code for EAC