A standard-driven communication protocol for disconnected clinics in an healthcare scenario

Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi

Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
masi@math.unifi.it, rosario.pugliese@unifi.it, tiezzi@dsi.unifi.it

Abstract. When building an Electronic Health Record (EHR) project from scratch, the problem of disconnected, paper-based clinics arises, specially in countries where no governmental network infrastructure exists. These clinics have to exchange sensitive healthcare data that must be protected against disclosure and tampering. On the other hand, interoperability of software components is a key requirement that must be fulfilled when building large scale projects that can affect the safety of patients. Therefore, standard organizations defined a set of specifications that can be used for building EHR-based applications. In this paper, we propose a protocol for obtaining healthcare documents in a setting where no network connection is available. To guarantee interoperability and integrability, as well as security and safety on the treatment of patients healthcare data, our protocol exploits many largely adopted international standards. We also formalise the protocol by means of the process calculus COWS and use model checking techniques for proving its correctness with respect to the properties of interest.

1 Introduction

More and more countries are by now switching from a paper based healthcare management to an Electronic Health Record (EHR) based solution. An EHR is a set of sensitive data written in a machine readable format containing the healthcare history of a patient (e.g. the patient summary, medical exams, prescriptions). However, the adoption of such a technology does not come for free for patients and healthcare professionals, but it affects everyday life. Therefore, software architects are required to design healthcare solutions that are coherent, interoperable, secure and easy to use.

Worldwide standardization initiatives [1, 2] have been hence founded for promoting the coordinated use of established standards (see e.g. [3, 4]). As a result of one of these efforts, the profile Cross Enterprise Document Sharing using Portable Media (XDM) has been proposed. This is an integration profile defined by [1] for the transmission of EHRs using different types of electronic supports like CDs, DVDs or even USB drives. However, simply applying existing standards and technologies makes real world architectures more and more complex and, unfortunately, does not guarantee absence of security flaws, as we shown in [5], where we used formal methods to analyse and tune the standards to real-world healthcare scenarios for HIPAA (Health Insurance Portability and Accountability Act) compliant projects.

In this paper, we continue pursuing that research line and focus on the problem of sharing patients healthcare data among clinics without any connection to the Internet.

This is a frequent problem in developing countries where no network infrastructure is given by the institutions. A clinic located in a desert that can be reached only by means of hundreds of kilometers of tracks sand, or a caravan that travels along townships with first-aid equipment are scenarios in the scope of our work.

As a first contribution of this paper, we propose a protocol, based on XDM and other standards, that guarantees some stringent security and safety properties on the treatment of patients healthcare data. Since we want our protocol to be implementable and integrable with different standard components, all the exchanged messages are based on international standards. Specifically, all messages transporting patients healthcare information are based on SOAP and IHE XDS.b/XCA [1]. The format used for the description of healthcare information is defined by Health Level 7 [4]. We used WS-Security [6] for embedding tokens into the SOAP message header. The W3C standards XML-Signature [7] and XML-Encryption [8] are used for digital signatures and encryption. The protocol used for obtaining a security token is the OASIS WS-Trust [9] and all the security tokens are encoded as SAML Authentication Assertions [10].

As a second contribution, we formally specify the protocol using the process calculus COWS [11] so that the obtained model contains enough details about the involved technologies (this is e.g. reflected by letting the exchanged messages travel along communication channels with different properties). Then, we define a threat model where a theoretical attacker [12, 13] can seriously damage the health of patients or obtain unauthorised resources by carrying out different types of attacks. Finally, we analyze the model obtained by using the model checker CMC [14] and prove that, in our threat model, the attacker cannot perform any action that can compromise the patients' safety.

The rest of the paper is organized as follows. In Section 2, we propose an XDM-based protocol and a threat model for the analysis of security properties. In Section 3, we present the process calculus COWS and then use it for formally modelling the protocol. In Section 4, we check the COWS model against the security properties of interest by using the tool CMC. In Section 5, we conclude by also touching upon related work.

2 An XDM-based communication protocol

The problem of connecting remote and paper-based clinics without Internet connection in a wide geographical area is not novel. To face it, many *ad hoc* solutions have been designed, thus losing the interoperability level required by standard organizations. To recover interoperability, [1] has put forward the use of the profile XDM for exchanging documents by using a common file and directory structure over different media. Here, we want to integrate this specification with other standards in order to be able to prove that an intruder cannot perform actions that can cause damages at the safety of patients.

A typical scenario is illustrated in Figure 1. Some patients are receiving healthcare treatment in clinic *A*, located in a region without neither Internet connection, satellite, nor GSM coverage. A governmental agency, clinic *B*, is providing a storage of patients' healthcare records, based on standard profiles. To provide optimum care for patients, clinic *A* must register all patients' healthcare documents in the central storage. The XDM specification defines a way for clinic *A* to write the documents in a certain format within portable media (e.g. CD, DVD, USB drive) that patients can carry whenever

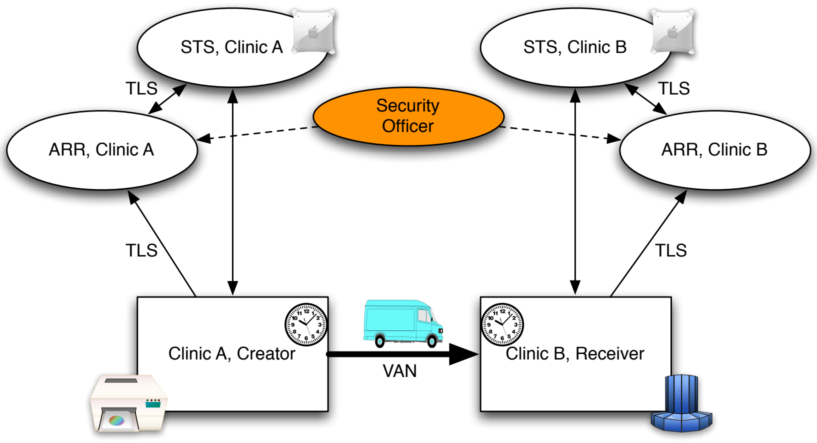

Fig. 1. A typical scenario

they travel over the region of residence. The records produced can also be sent to other clinics (e.g. a regional hospital) creating a connected graph of clinics. Because of lack of network connection, the documents registered in the portable media are sent using a car-transportation system. At clinic *B*, authorised personnel can read the portable media and import the documents in the local hospital information system.

We propose an XDM-based protocol for facing the severe security and safety problems on the treatment of patients healthcare data that the scenario outlined above presents. For example, a patient could forge his own portable media including new prescriptions for drugs or an intruder could easily have access to the data by hijacking the carrier on its way (or the carrier itself could act as an intruder).

Of course, we also want to preserve interoperability. Therefore, we only rely on standards and specifications usually used in EHR settings coming e.g. from [4, 1, 15]. In this way, e.g., we cannot prevent the attacks by the intruder, but we can let another actor, that we call *security officer*[1], to take charge of tracing if one of the attacks mentioned above was performed.

2.1 An abstract model of the protocol

In the abstract model of the scenario illustrated in Figure 1, each clinic has an hospital information system and consists of three actors: the *creator/receiver* that submits/maintains documents stored in a database, an *audit record repository ARR*, i.e. a tamper-proof log storage, that keeps track of all the transactions performed by the creator/receiver, and a *security token service STS* that is responsible to issue and validate SAML tokens. We suppose that each clinic has a local clock, but there is no global clock (i.e. clinics do not synchronise with each other). Generally, clinics receiving patients healthcare documents have an Internet infrastructure; if this is not the case, the protocol can

[1] In the real world, the security officer can be seen as a persona that, on duty of the government, supervises the correct information flow inside the clinics.

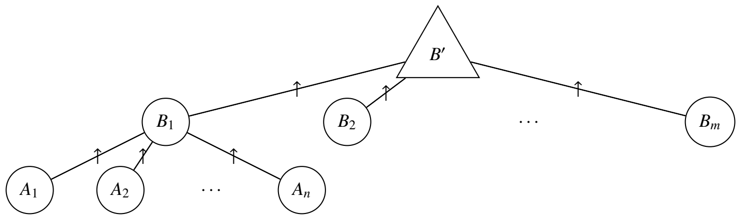

Fig. 2. The tree architecture of the proposed XDM-based protocol

be reused recursively as shown in Figure 2, where each receiving clinic acts as a creator for an upper level receiver.

Our model does not abstract away from the kind of used communication channels in order to take into account the different guarantees they provide. Hence, we use three different kinds of channels. The first one, one, $\rightarrow$, is a plain TCP/IP channel. The second one, $\rightarrow_{TLS}$, is a TCP/IP channel where TLSv1 is available. This means that confidentiality and integrity of TCP packets are guaranteed. The third one, $\rightarrow_{VAN}$, represents the car-transportation system (conveying patients' healthcare documents). Since this channel allows to send only one message per protocol run, it does not satisfy any of the four authentication properties (*aliveness, weak agreement, non-injective agreement* and *agreement*) of [16]. Therefore, differently from the other kinds of channel, in this case the knowledge that "something went wrong" in the communication is not deducible.

Table 1 presents an abstract description of the XDM-based protocol where clinic A wants to send a document (in XDM jargon, a *provide and register document set*) to an upper level clinic B. Notation $\{M\}_{K_B^+}$ stands for the encryption of message M using B's public key, K_B^+, and $\{[M]\}_{K_B^-}$ means that message M is signed using B's private key, K_B^-. Each message M represents a SOAP 1.2 message strictly defined by international standards [15, 1, 4].

Here, we are not interested in the exact structure of messages, but rather in their content, for which it is enough to represent messages as tuples. For instance, notation $\{[K_A^+, user, \text{'}doc\text{'}]\}_{K_A^-}$ stands for the signed tuple encoded as XML-Signature [7] of the structured document '*doc*'[2], done by the user *user* using the certificate K_A^+. The corresponding XML fragment is:

```
<ds:Signature Id="id" >
  <ds:SignedInfo>
    <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <ds:SignatureValue>e76Bd...</ds:SignatureValue>
      <ds:KeyInfo>
        <ds:X509Data><ds:X509Certificate>MIICBz...</ds:X509Certificate></ds:X509Data>
      </ds:KeyInfo>
  </ds:SignedInfo></ds:Signature>
<body><ProvideAndRegisterDocumentSetRequest id="id">
        <Slot name="authorPerson" value="user"/> ...
      </ProvideAndRegisterDocumentSetRequest></body>
```

where the value of the signature is the element `SignatureValue`, the certificate is in the element `KeyInfo` and the user is identified by the slot `authorPerson`.

[2] Notation '*doc*' means that *doc* is not just a value, but an XML structured document.

$$
\begin{array}{l}
0.\ \left\{\begin{array}{ll} A \leftrightarrows STS_A & : \ldots \textit{ obtains a SAML assertion for B at ts1 with context ctx } \ldots \\ STS_A \rightarrow_{TLS} ARR_A & : \mathit{ts1}, \mathit{ctx}, A, B \\ A \leftrightarrows STS_A & : \ldots \textit{ obtains the } \mathit{token}(K_A^+, \mathit{user}, \{\mathit{ctx}\}_{K_B^+}) \ \ldots \end{array}\right. \\
1.\ A \rightarrow_{TLS} ARR_A : \mathit{ts1}, \mathit{ctx}, A, B, \mathit{ts2}, \{[K_A^+, \mathit{user}, \text{‘}\mathit{doc}\text{’}]\}_{K_A^-} \\
2.\ A \rightarrow_{VAN} B : A, B, \mathit{msgId}_1, \mathit{ts2}, \{[K_A^+, \mathit{user}, \text{‘}\mathit{doc}\text{’}]\}_{K_A^-}, \mathit{token}(K_A^+, \mathit{user}, \{\mathit{ctx}\}_{K_B^+}) \\
3.\ B \rightarrow_{TLS} ARR_B : \mathit{ts2}, \mathit{ts3}, A, \mathit{ctx}, \{[K_A^+, \mathit{user}, \text{‘}\mathit{doc}\text{’}]\}_{K_A^-}
\end{array}
$$

Table 1. An abstract description of the proposed XDM-based protocol

Message identifiers and URLs of senders/receivers are encoded using WS-Addressing [17], while timestamps are encoded using WS-Security [6].

According to the XDM specification, the creator starts by obtaining a token (that in our setting is a SAML Assertion [10]) through its local *STS*. Notably, a token permits only one operation via the portable media. The token issuance process is assumed to be performed according to [5] and is not detailed here for the sake of simplicity (for this reason, it is indicated as step 0 in the protocol). It is however worth noticing that there exists a one to one mapping between the token issued during the process and the WS-Trust [9] context *ctx*, that is a unique identifier computed by the *STS*.

Before sending the SAML token to the creator, the *STS* sends a message through channel $\rightarrow_{TLS}$ to its ARR_A containing the timestamp of the original token request message *ts1*, the context *ctx* and the names of the creator *A* and of the destination *B* of the message sent using the portable medium. The knowledge of this message is “user *A* obtained a SAML token using context *ctx* at time *ts1* for sending documents to *B*”. The SAML token obtained, for the sake of readability denoted by $\mathit{token}(K_A^+, \mathit{user}, \{\mathit{ctx}\}_{K_B^+})$, stands for the signed tuple $\{[K_A^+, \mathit{STSName}, \mathit{samlTs}, \mathit{user}, B, \{\mathit{ctx}\}_{K_B^+}]\}_{K_{STS_A}^-}$ representing the XML fragment of the SAML assertion shown in Figure 3. This SAML assertion contains the name *user* of the user that is performing the action and where it comes from, *A*, as `Subject` and `SubjectConfirmation` method elements. *STSName* represents the `Issuer` element, *samlTs* is the timestamp of the assertion and *B* is the value of the `AudienceRestriction` list (where the assertion can be used, the upper level clinic in this case). The token contain a secret as SAML attribute for *B* that is the encrypted context, $\{\mathit{ctx}\}_{K_B^+}$. The encrypted context can be seen as a pointer to the log record in ARR_A that contains the information about the token issuance process.

In message 1, clinic *A* updates the log in ARR_A by adding a new timestamp *ts2*, representing the instant when the portable medium is created (e.g. the CD is burned), and a signed tuple, representing the username of the physical person sitting in front of the workstation, which is the same as the subject of the SAML assertion, and the document, here abstracted as ‘*doc*’. It is worth noticing that we are not deviating from the standards. In fact, the username value is part of the metadata accompanying the document itself; the interested reader is referred to the XDS.b documentation (*XCN values*, [1]) for the other metadata.

In message 2, clinic *A* sends the portable medium through the car-transportation system that travels to *B*. The message contains a unique identifier, msgId_1, the signed document and the SAML token.

When clinic *B* receives the message, it checks if the value *user* equals the subject of the assertion and if the key contained in the signed document matches the holder-of-key

```
<saml:Assertion IssueInstant=''samlTs''><saml:Issuer> STSName </saml:Issuer>
   <ds:Signature> ... </ds:Signature>
   <saml:Subject><saml:NameID> user </saml:NameID>
       <saml:SubjectConfirmation Method="#holder-of-key">
          ... certificate of the machine where the subject is running ...
       </saml:SubjectConfirmation>
   </saml:Subject>
   <saml:Conditions NotBefore="samlTs" NotOnOrAfter="samlTs+n">
       <saml:AudienceRestriction><saml:Audience> B </saml:Audience></saml:AudienceRestriction>
   </saml:Conditions>
   <saml:AttributeStatement> enc_ctx </saml:AttributeStatement>
</saml:Assertion>
```

Fig. 3. Excerpt of a sample SAML token (using the *holder-of-key* method)

data inside the token. If both tests succeed, through message 3, clinic B validates the assertion to its security token service STS_B, as in [5], and writes in its local ARR_B the following knowledge: "someone, probably A, at time *ts2*, sent a message with identifier *ctx* for a document represented by the signed tuple $\{[K_A^+, user, \text{'}doc\text{'}]\}_{K_A^-}$; now is *ts3*". The document contains also the certificate that corresponds to the signer's public key, as the `KeyInfo` element of XML-Signature [7].

The method used by the receiver for confirming the subject deserves special mention. Three methods are defined by the SAML specifications: in the *bearer method* the subject of the assertion is believed to be the "carrier" of the assertion; in the *holder-of-key* the subject of the assertion is identified through a digital identity inserted by the security token service at the moment of the authentication (e.g. an X509 certificate) and can be confirmed by the receiver if the same digital identity is available; in the *sender-vouches* the subject of the assertion is identified trough assertions exchanged along an out-of-bound channel established between the sender and the receiver. We proved in [5] that in untrusted network environments the *holder-of-key* method is preferable because the knowledge given by the STS was available in the communication channel. Therefore, since we have the same authentication assurance level at the clinic, we inherit the properties already proved, by adopting the same subject confirmation method.

2.2 The threat model

By taking the features of the different kinds of communication channels into account, we consider a standard intruder *à la* Dolev-Yao [12] acting only along channel $\rightarrow_{VAN}$. The intruder can see all the messages passing through the channel (i.e. he can read the documents delivered via van), he is a legitimate user of the network (in our setting this means that he is able to sign documents) and has the opportunity to be a receiver for any user. We also assume that encryption and hashing functions are strong enough not to be easily breakable.

Our assumption that channels of type $\rightarrow_{TLS}$ are not subject to the intruder is supported by the fact that, in real world, each clinic' hospital information system usually runs on a single, audited, machine. Therefore the problem of intercepting messages can be solved better by using intrusion detection techniques.

We thus identify 4 different types of attacks the intruder I can perform.

1. I suppresses a message. For instance, I could suppress a submission for a new rare form of allergy or query for crucial healthcare data and, in both cases, the

suppression may lead to the death of the patient if the patient travels along clinics. It is thus crucial for patients safety to consider suppressed messages as attacks, as these messages could bear data necessary for the life of patients. Moreover, since messages travelling along $\rightarrow_{VAN}$ may have unbounded delays, to stay on the safe side, we consider also these delayed messages as attacks. All these attacks can be discovered by the security officer.

2. An healthcare professional sitting on A wants to access restricted resources by reusing an already issued SAML token. For instance, suppose that *nurse* (a valid user at A) wants to obtain a big amount of drugs for the illegal market. He could reuse an already used assertion issued for another purpose for creating a new illicit prescription. *nurse* then steals the assertion from the portable medium, attaches it to his new prescription and sends:

$$A \rightarrow_{VAN} B : A, B, msgId, ts2, \{[K_A^+, nurse, \text{'}prescr\text{'}]\}_{K_A^-}, token(K_A^+, user, \{ctx\}_{K_B^+}) \quad (1)$$

This attack is discovered by the *Receiver* because the user of the token is different from the creating user embedded in the document.

3. I obtains the message by listening on $\rightarrow_{VAN}$, suppresses it and sends

$$I \rightarrow_{VAN} B : I, B, msgId, ts2, \{[K_A^+, user, \text{'}prescr\text{'}]\}_{K_I^-}, token(K_A^+, user, \{ctx\}_{K_B^+}) \quad (2)$$

The difference with attack number 2 is that *nurse* is sitting on clinic A, while here the intruder is intercepting the van. This attack is discovered by the *Receiver* because the public part of the intruder's signing key K_I^- is different from the key associated to the document and inside the token.

4. I sends multiple times the same (intercepted) message, for example for obtaining multiple times the same resource. This attack is similar to attacks 2 and 3, and is a form of replay attack. It is discovered by the *Receiver* because multiple *ctx* are present in the database (only the firstly received message is left).

2.3 The role of the officer

We introduce here another actor in the scenario, the security officer. The officer polls the clinics in a round-robin fashion to detect if attack 1 has been performed, in which case it establishes which actions must be performed by the clinics as countermeasures. To this aim, every quantum of time t the officer checks the logs of every ARR_i by appropriately comparing them with the logs stored in the clinics B at the upper level.

When the officer visits clinic A_i, he obtains the set $\mathcal{A}_i^{log}$ of all the audit trails from his last visit. Let m_i be the maximum among the timestamps of the audits (the officer saves the previous value of m_i as $oldm_i$). When the officer visits clinic B, he obtains the set $\mathcal{B}^{log}$ of all the audits trail. Let's recall how an audit located in ARR_B looks like. It can be defined as a tuple $\mathbf{a} = \langle creation\_ts, arrival\_ts, sender, ctx, Signature\rangle$, where *creation_ts* is the creation time at *sender*, *arrival_ts* is the arrival time at B, *ctx* is the context and *Signature* is the signature of the document. In the sequel, $\mathbf{a}(1)$ will denote *creation_ts*, $\mathbf{a}(3)$ will denote *sender*. The officer can then partition $\mathcal{B}^{log}$ by defining:

$$\mathcal{B}^{log}|_{A_i,m_i} = \left\{ \mathbf{a} \mid \mathbf{a}(3) = A_i \wedge oldm_i < \mathbf{a}(1) \leq m_i \right\}$$

Each element $\mathcal{B}^{log}|_{A_i,m_i}$ of the partition contains the audits coming from clinic A_i with a timestamp greater than the old lecture $oldm_i$ did by the officer and not greater than the last timestamp m_i read at A_i.

The officer can now tell the different situations apart. If $\mathcal{A}_i^{log} = \mathcal{B}^{log}|_{A_i,m_i}$ then all messages produced at clinic A_i arrived safely to clinic B. If this holds for all clinics, then we have $\bigcup_{i=1}^{n} \mathcal{A}_i^{log} = \mathcal{B}^{log}$ that means that every message from any clinic is safely arrived at B. Instead, if $\mathcal{A}_i^{log} \backslash (\mathcal{B}^{log}|_{A_i,m_i}) \neq \emptyset$ then there exist messages in A_i that are not yet arrived at B. The messages are suppressed or are late (attack 1). The opposite situation is when $\mathcal{B}^{log}|_{A_i,m_i} \backslash \mathcal{A}_i^{log} \neq \emptyset$ which means there are more messages at B than those produced by the clinics, i.e. some messages have been introduced in the channels by the intruder (attacks 2, 3 and 4).

By means of the formal analysis carried out in Section 4, we prove that these situations will never occur.

3 A COWS model

In this section, first we report the syntax and the informal semantics of COWS then we present the COWS specification of the protocol proposed in Section 2. For the sake of simplicity, we present here a fragment of COWS without linguistic constructs for dealing with forced termination, since such primitives have not been used in the protocol specification. We refer the interested reader to [11] for the presentation of the full language and for many examples illustrating COWS peculiarities and expressiveness.

3.1 COWS syntax and informal semantics

COWS [11] is a formalism specifically devised for modelling (and analysing) service-oriented applications; in fact, its design has been influenced by the principles underlying the OASIS standard for orchestration of web services WS-BPEL [18]. The syntax of COWS, written in the ‘machine readable’ format accepted by the interpreter and the model checker CMC [14] that we use for the analysis, is presented in Table 2. It is defined using the following notational conventions: *variables* (ranged over by `X`, `Y`, ...) start with capital letters; *names* (ranged over by `n`, `m`, ..., `p`, `p'`, ..., `o`, `o'`, ... and used to represent partners and operations) start with digits or lower case letters; *expressions* (ranged over by e_1, e_2, ...) contain values and variables and are formed using standard arithmetic, boolean and string operators; *name identifiers* (ranged over by u, u', u_1, u_2 ... and used as non-terminal symbols only) are either variables or names; *value identifiers* (ranged over by w_1, w_2, ... and used as non-terminal symbols only) are either variables or values; *service identifiers* (ranged over by `A`, `B`, ...) start with capital letters and have fixed non-negative arities.

Invoke and *receive* are the basic communication activities provided by COWS. Besides the parameters, both activities indicate an *endpoint*, i.e. a pair composed of a partner name `p` and an operation name `o`, through which communication should occur. An endpoint `p.o` can be interpreted as a specific implementation of operation `o` provided by the service identified by the logic name `p`. An invoke `p.o!` $<e_1, \ldots, e_n>$ can proceed as soon as all expressions $e_1, \ldots, e_n$ have been evaluated. A receive `p.o?` $<w_1, \ldots, w_n>$ $.s$

$s ::=$	(services)
$\mathtt{nil} \;\mid\; u\,.\,u'!<e_1,\ldots,e_n> \;\mid\; \mathtt{p}\,.\,\mathtt{o}?<w_1,\ldots,w_m>\,.\,s$	(empty activity, invoke, receive)
$\mid\; s_1 + s_2 \;\mid\; s_1 \mid s_2 \;\mid\; * \, s$	(choice, parallel, replication)
$\mid\; [\mathtt{n}\sharp]\, s \;\mid\; [\mathtt{X}]\, s \;\mid\; \mathtt{A}(w_1,\ldots,w_n)$	(name/var. delimitation, call)
$\mid\; \mathtt{let}\ \mathtt{A}(u_1,\ldots,u_n) = s_1\ \ldots\ \mathtt{B}(u'_1,\ldots,u'_m) = s_2\ \mathtt{in}\ s'\ \mathtt{end}$	(let definition)

Table 2. COWS syntax

offers an invocable operation o along a given partner name p. Partner and operation names can be exchanged in communication (although dynamically received names cannot form the endpoints used to receive further invocations) thus supporting the modelling of many service interaction and reconfiguration patterns.

A *choice* can be used to pick out one receive activity among those enabled for execution.

Execution of *parallel* terms is interleaved, except when a communication can be performed. Inter-service communication takes place when the arguments of a receive and of a parallel invoke along the same endpoint match. If more than one matching receives are ready to process a given invoke, only one of the receives with greater priority (i.e. the receives that generate the substitution with 'smaller' domain, see [11]) is allowed to progress.

The *delimitation* operators are the *only* binders of the calculus: $[\mathtt{n}\sharp]\, s$ and $[\mathtt{X}]\, s$ bind n and X, respectively, in the scope s. Name delimitation can be used to generate 'fresh' private names (like the restriction operator of π-calculus), while variable delimitation can be used to regulate the range of application of the substitution generated by an inter-service communication. In this latter case, each variable argument of the receive is replaced by the corresponding evaluated argument of the invoke within the whole scope of variable's declaration. In fact, to enable parallel terms to share the state (or part of it), receive activities in COWS do *not* bind variables.

The *replication* construct $*\, s$ permits to spawn in parallel as many copies of s as necessary. This, for example, is exploited to model persistent services, i.e. services which can create multiple instances to serve several requests simultaneously.

Finally, the *let* construct permits to re-use the same 'service code', thus allowing to define services in a modular style; $\mathtt{let}\ \mathtt{A}(u_1,\ldots,u_n) = s\ \ldots\ \mathtt{in}\ s'\ \mathtt{end}$ behaves like s', where calls to A can occur. A service *call* $\mathtt{A}(w_1,\ldots,w_n)$ occurring in the body s' of a $\mathtt{let}\ \mathtt{A}(u_1,\ldots,u_n) = s\ \ldots\ \mathtt{in}\ s'\ \mathtt{end}$ behaves like the service obtained from s by replacing the formal parameters $u_1,\ldots,u_n$ with the corresponding actual parameters $w_1,\ldots,w_n$.

3.2 Protocol specification

Due to lack of space we only present here the relevant part of the COWS specification of the protocol in Section 2. We refer the interested reader to [19] for the complete specification.

The COWS term representing the overall scenario follows (to make the reading easier, we have omitted irrelevant details):

```
let ... in
  [hashReq♯] [hashResp♯] ...
```

```
    ( Sha(hashReq,hashResp) | Cipher(...) | Signer(...)
      | ClinicA(hashReq,hashResp,...) | ClinicB(hashReq,hashResp,...) | Officer(...) )
    | ... intruders ...
  end
```

Our protocol has three main participants: `ClinicA`, `ClinicB` and `Officer`. Since COWS does not offer primitives for, e.g., *encryption* and *hashing*, these and other useful security-related features are provided to each participant through a library of functions, implemented as a set of shared services. We have implementations of such algorithms as SHA for hashing (`Sha`) and RSA for public-key cryptography (`Cipher`) and digital signature (`Signer`), that are necessary to properly manage the data to be exchanged during protocol runs. These COWS services play a role similar to that of functions in the applied π-calculus [20]. Name delimitations are used here to restrict access to services `Sha`, `Cipher` and `Signer` by declaring that `hashReq`, `hashResp`, ... are private operation names known only to the three participants (but for, of course, the services that provide them). As an example of the above services, we report here an excerpt of `Signer`:

```
  * [Requester][RequesterName][Hash]
       signer.sign_KaSec_Req?<Requester,RequesterName,Hash>.
       [fresh#] ( Requester.sign_KaSec_Resp!<Hash,fresh>
                 | * [Verifier][X]
                     ( signer.verify_KaPub_Req?<Verifier,RequesterName,X>.
                          ( sys.attack3Detected!<invalidSignature>
                            | sys.attack3Detected?<invalidSignature>.nil )
                       + signer.verify_KaPub_Req?<Verifier,RequesterName,fresh>.
                          Verifier.verify_KaPub_Resp!<fresh,Hash> ) ) | ...
```

When it receives an hash value to be signed with key K_A^-, it generates a fresh name representing the signature value, which is returned to the requester. `Signer` then waits for signature verification. Notably, a request of signature verification for a document signed with a key different from K_A^- raises an invalid signature error (see attack 3).

The creator clinic is rendered in COWS as a service definition within the above `let` construct:

```
ClinicA(hashReq,hashResp,...) = [clockA#][write#][getToken#] ...
                                ( Clock(clockA,...) | ARR(write,...) | STS(getToken,,...)
                                  | Creator(clockA,write,getToken,...) )
```

The term `ClinicB` representing the upper level clinic is similar to `ClinicA`, but for the term `Receiver` in place of `Creator`.

Each clinic has its own local clock (`Clock`), *Audit Record Repository* (`ARR`) and *Security Token Service (*`STS`*)*, with which it shares private partner and operation names (e.g., `clockA`, `write` and `getToken`). These names permit defining private endpoints for simulating internal interaction with `Clock`, secure connection along channels of kind $\rightarrow_{TLS}$ with `ARR`, and the authenticated connection along channels of kind $\rightarrow$ with `STS` as dealt with in [5]. The local clock ticks along a private endpoint and, when prompted, returns the current value to the requester. Instead, when `STS` receives a request from the associated participant, it generates and returns a SAML token containing a unique context. Due to space limitations, we refer the interested reader to [19] for the

COWS terms specifying the behavior of `Clock` and `STS`; instead, we report here the term modelling the behaviour of `ARR`[3]

```
(Lifo(q)
 | -- Functionalities for creator
   * [Ts1][Ctx][A][B] p.write?<Ts1,Ctx,A,B>.
     [Ts2][SignedDoc] p.write?<Ts1,Ctx,A,B,Ts2,SignedDoc>.
     [n#] ( -- Write the data in the Lifo queue
            q.push!<Ctx,A,B,Ts2,SignedDoc,n> | n.op?<>. A.ack!<> )
 | -- Functionalities for receiver
   * [Ts2][Ts3][A][Ctx][SignedDoc] p.write?<Ts2,Ts3,A,Ctx,SignedDoc>.
     [n#] ( -- Write the data in the Lifo queue
            q.push!<Ctx,A,p,Ts2,SignedDoc,n>
            | -- Arrival of another 'write' request with the same context
              [X1][X2][X3][X4] p.write?<X1,X2,X3,Ctx,X4>.
              (sys.attack4Detected!<Ctx> | sys.attack4Detected?<Ctx>.nil ) ) )
```

`ARR` instantiates a *stack*, a data structure used to store audit trails, and provides different functionalities depending on the role of the participant. In case of creator role, `ARR` waits a first message from `STS` stating that an audit trail with a given context will arrive from the clinic. When the audit is received, it is pushed into the stack. In case of receiver role, `ARR` simply pushes the received audit trail into the stack, unless the audit trail contains a context already stored (attack 4). The security officer (that will be shown in the next section) will exploit the stored audit trails for checking if attack 1 has occurred.

The term `Creator` is defined as

```
[ts#] ( clock.get!<ts> | [Ts1] clock.ret?<ts,Ts1>.
          ( stsA.getToken!<a,b,user,Ts1>
           | [Ctx][StsName][SamlTs][EncCtx][Signature]
              a.retToken?<Ctx,a,StsName,SamlTs,user,b,EncCtx,Signature>.
              ( clock.get!<ts> | [Ts2] clock.ret?<ts,Ts2>.
                ( -- Create the hash of doc
                  hFunc.hashReq!<a,user,doc>
                  | [Dochash] a.hashResp?<user,doc,Dochash>.
                  ( -- Sign the hash code: we use "a" also as creator name
                    signer.sign!<a,a,Dochash>
                    | [SignedDoc] a.signResp?<Dochash,SignedDoc>.
                      [msgId1#] ( -- Updates the audit trail of the transaction
                      a.write!<Ts1,Ctx,a,b,Ts2,SignedDoc>
                      | -- Wait an ack from ARR
                        a.ack?<>.( -- Send the document over portable medium
                        b.van!<a,b,msgId1,Ts2,a,user,doc,SignedDoc,a,
                               StsName,SamlTs,user,b,EncCtx,Signature>
                        | -- 2nd message  ... ) ) ) ) ) ) )
```

As expected, `Creator` first obtains the current value of its local clock, then requests a token to its local `STS`, which returns the tuple

$$< \texttt{Ctx}, \texttt{a}, \texttt{StsName}, \texttt{SamlTs}, \texttt{user}, \texttt{b}, \texttt{EncCtx}, \texttt{Signature} >$$

that represents the SAML token $token(K_A^+, user, \{ctx\}_{K_B^+})$ (recall that this token stands for $\{[K_A^+, STSName, samlTs, user, B, \{ctx\}_{K_B^+}]\}_{K_{STS_A}^-}$). The public key of the creator is abstracted by the public name `a`, the $\{ctx\}_{K_B^+}$ by the result `EncCtx` of the application of the encryption function on `Ctx`, and the signature of the token by the outcome `Signature`

[3] The string $--$ indicates that the rest of the line is a comment (and it is ignored by CMC).

of the digital signing process. Before writing to the portable medium, Creator needs to obtain a new clock value, sign the document (hash and encryption of the hash) and inform its local ARR about its intention to write to the portable medium. Creator, whenever receives an acknowledgement from *ARR* about the success of recording, sends a message along the public endpoint b.van (this action abstracts writing the document on the portable medium and sending it through $\rightarrow_{VAN}$). Then, the process is repeated to create and send a second message containing another document.

The term Receiver is as follows

```
* [A][MsgId1][Ts2] ...
  b.van?<A,b,MsgId1,Ts2,AfromDoc,User,Doc,SignedDoc,AfromToken,
         StsNameA,SamlTs,UserFromToken,BfromToken,EncCtx,Signature>.
  [ts#]( clock.get!<ts>
  | [Ts3] clock.ret?<ts,Ts3>.
    [comp#][X][Y]
    ( -- Check if the A's certificate in the document signature equals
      -- the certificate within the token (HoK) and check if the user
      -- value in the document signature equals the user within the token
      b.comp!<AfromDoc,User>
      | b.comp?<X,Y>.
          ( sys.attack2Detected!<User,UserFromToken>
            | sys.attack2Detected?<User,UserFromToken>.nil )
        + b.comp?<AfromToken,UserFromToken>.
          [MsgID]
            ( stsB.validateToken!<MsgId1,Ts2,AfromToken,StsNameA,SamlTs,
                                  UserFromToken,BfromToken,EncCtx,Signature>
            | b.validateResp?<MsgId1,MsgID,invalid>.
                  sys.attackDetected!<invalidToken>
              + b.validateResp?<MsgId1,MsgID,valid>.
                ( -- Check the signature of the document
                  signer.verifyReq!<b,AfromDoc,SignedDoc>
                  | [DecipheredHash]
                     b.verifyResp?<SignedDoc,DecipheredHash>.
                     ( -- Calculate the hash
                       hFunc.hashReq!<b,User,Doc>
                       | [Hash] b.hashResp?<User,Doc,Hash>.
                         [comp#] -- Compare the two hashes
                         ( b.comp!<Hash>
                           | [X][msgId#]
                             ( b.comp?<X>.-- The two hashes do not coincide
                                    ...
                               + b.comp?<DecipheredHash>.-- The two hashes coincide
                                  ( -- Decode ctx with B's secret key
                                    cipher.decode!<b,EncCtx>
                                    | [Ctx] b.decodeResp?<EncCtx,Ctx>.
                                      -- Write to ARR
                                      b.write!<Ts2,Ts3,A,Ctx,SignedDoc> ) ... )
```

Once prompted by a message, representing the portable medium, along the public endpoint b.van, Receiver gets its local clock current time and checks if the user written in the signed document equals the user written in the accompanying assertion and if the key inside the assertion corresponds to the key inside the signed document. These checks are made for discovering attacks of type 2. Receiver then validates the token through its local STS. Now, if the token is not valid (the attacker tampered the token), STS returns an invalid message and Receiver terminates. Otherwise, it checks the signature of the document (for discovering attacks of type 3, see Signer's behaviour), decrypts the WS-Trust context of the assertion issuance process made by `Creator` and records this evidence in its `ARR`.

4 Model Checking based analysis

The analysis of the protocol is carried out by means of the methodology introduced in [14] for model checking COWS specifications. More specifically, the properties we want our protocol to satisfy are expressed as formulae of SocL, an action- and state-based, branching time, temporal logic specifically designed to express properties of service-based systems. The verification of SocL formulae over our COWS specification of the protocol is assisted by CMC, a bounded, on-the-fly model checker. SocL formulae are stated in terms of *abstract* actions, meaning that, e.g., a certain attack has been performed or detected; instead, COWS specifications are expressed in terms of *concrete* actions, i.e. communications of data tuples along endpoints. Thus, to enable the verification, CMC permits also to specify a set of *transformation rules* that map concrete actions to abstract actions. We refer the interested reader to [14] for a complete account of the methodology.

Each of the 4 attacks described in Section 2.2 is modelled as a COWS term running in parallel with the protocol's participants we described in Section 3. For space limitations, we will not introduce all the constructs of SocL, but only explain the formulae used in each attack.

Attack 1 occurs when the intruder suppresses messages. This attack is discovered by the security `Officer`, of which, for space limitations, we only describe a few significant fragments (we refer the interested reader to [19] for its complete specification). Only one instance of `Officer` can run at a given time. Once activated by a signal, `Officer` gets the values stored by `Creator`'s audit record repository, $\mathtt{ARR}_A$.

```
[r#][e#]( qa.pop!<r,e> | [Ctx1][A1][B1][MaxT1][SignedDoc1]
    ( e.op?<>.
            -- ARR_a is empty: Officer does nothing
            nil
      +  r.op?<Ctx1,A1,B1,MaxT1,SignedDoc1>.
            -- The most recent audit trail has been extracted from ARR_a
            ... ) )
```

`Officer` saves in the variable `MaxT1` the maximum timestamp of the audits in $\mathtt{ARR}_A$. Similarly, `Officer` gets the audits, having timestamp not greater than `MaxT1`, stored by `Receiver`'s audit record repository, $\mathtt{ARR}_B$. After collecting the audits from the two repositories, `Officer` compares them

```
-- Get an audit from ARR_a
( -- Check if the audit is in B_log
  off.logB!<Ctx,SignedDoc,n>
  | [X1][X2][X3]
    ( off.n?<>.
           -- Audit found in B_log: continue the loop
           loop.op!<>
      + off.logB?<X1,X2,X3>.
           -- The audit is not in B_log: message suppressed
           ( sys.attack1Detected!<messageSuppressed,SignedDoc>
             | sys.attack1Detected?<messageSuppressed,SignedDoc>.loop.op!<> ) ) )
```

The COWS term modelling the intruder conducting attack 1 is

```
[A][MsgId1][Ts2][SignedDoc] ...
b.van?<A,b,MsgId1,Ts2,SignedDoc,...>.
( sys.attack1!<messageSuppressed,SignedDoc>
  | sys.attack1?<messageSuppressed,SignedDoc>. officier.activate!<> )
```

When the intruder intercepts the message along `b.van`, communication along the end-point `sys.attack1` takes place; this is used during the analysis to signal that the system is under attack. Only at this point `Officer` is activated (this device permits using just one `Officer`'s instance and, hence, reducing the model state space), after which it can go to clinic *A* and *B* at any time. The concrete action corresponding to a communication along `sys.attack1` is mapped to the abstract action `attack1Performed` by means of the following transformation rule:

```
Action sys.attack<messageSuppressed,$signedDoc> -> attack1Performed(message,$signedDoc)
```

A similar rule is used for mapping a communication along `sys.attack1Detected` to the abstract action `attack1DetectedByOfficer`:

```
Action sys.attackDetected<messageSuppressed,$signedDoc>
           -> attack1DetectedByOfficer(message,$signedDoc)
```

The SocL formula we used for discovering the attack is

```
AG [attack1Performed(message,$doc)] AF attack1DetectedByOfficer(message,%doc) true
```

The formula means that it holds *globally* (`AG`), i.e. in any state of the model, that *if* (operator [...][4]) an attack of type 1 is performed by the intruder, then *always* (`AF`) this attack will be detected by `Officer`. Evaluation of the formula returns `TRUE`, which means that if a message containing a certain document is suppressed, then the officer will discover that *B* did not received that specific message.

In the second attack, the intruder tries to reuse a token issued for another user for accessing unauthorized resources, by signing the message with the correct certificate of the clinic. The COWS term modelling the intruder conducting attack 2 is

```
[A][MsgId1][Ts2] ...
b.van?<A,b,MsgId1,Ts2,...>.
( sys.attack2!<> | sys.attack2?<>.
    ( -- Create the hash of the fake document
      hFunc.hashReq!<a,nurse,fakeDoc>
      | [FakeDochash] a.hashResp?<nurse,fakeDoc,FakeDochash>.
        ( -- Sign the hash code: "a" is used also as creator name
          signer.sign!<A,A,FakeDochash>
          | [FakeSignedDoc] a.signResp?<FakeDochash,FakeSignedDoc>.
            -- Send the fake message
            b.van!<A,b,msgIdFake,300,AfromDoc,nurse,fakeDoc,FakeSignedDoc,...> ) ) )
```

To abstract the intruder sitting on clinic *A*, we let the intruder above to intercept the message along the `b.van`. When this happens, communication along the endpoint `sys.attack2` takes place. The intruder then signs the document and sends it along `b.van` again. Notably, the token tuple is not changed.

For discovering this attack we used the SocL formula

[4] This is the modal logic operator *box*: `[a]f` states that, no matter how a process performs action `a`, the state it reaches in doing so will *necessarily* satisfy the property expressed by `f`.

```
AG [attack2Performed(AssertionReused)]
   AF attack2DetectedByB(invalidUser,$user1,differsFrom,$user2) true
```

stating that it holds globally that if an attack of type 2 is performed by the intruder, then this attack is always detected by clinic *B*. Evaluation of this formula returns TRUE. In fact, Receiver, whenever receives a message, compares the two users (UserFromDoc and UserFromToken) and, if they are different, performs a communication action along sys.attack2Detected that corresponds to the abstract action attack2DetectedByB.

In the third attack, the intruder intercepts the original message, e.g. by cracking the van, suppresses it and injects a new forged one, signed by its key (we assumed the intruder is a legitimate user of the network). The COWS specification of the intruder is similar to that for attack 2. The only change is when calling the signature function. In this case, the signature is made by using *I*'s keys pair and is rendered as

```
-- Sign the hash code: we use "I" as creator name
signer.sign!<i,i,FakeDochash>
| [FakeSignedDoc] i.signResp?<FakeDochash,FakeSignedDoc>.
  -- Send the fake message
  b.van!<i,b,msgIdFake,300,AfromDoc,User,fakeDoc,FakeSignedDoc,...>
```

For discovering this attack we used the SocL formula

```
AG [attack3Performed(invalidSignature)] AF attack3DetectedByB(invalidSignature) true
```

stating that it holds globally that if an attack of type 3 is performed by the intruder, then this attack is always detected by clinic *B*. Evaluation of this formula returns TRUE, because Signer, which is invoked by Receiver, detects the attack since the signature has been created by using the intruder's key rather than the key of the token.

Attack 4 takes place when the intruder sends multiple times the same message. We model this attack by letting the intruder to intercept the original message over b.van and send it twice, as in the following COWS fragment

```
[A][MsgId1][Ts2] ...
b.van?<A,b,MsgId1,Ts2,...>.
  ( sys.attack4!<> | sys.attack4?<>.
       ( -- Send multiple copies of the intercepted message
         b.van!<A,b,MsgId1,Ts2,...> | b.van!<A,b,MsgId1,Ts2,...> ) )
```

When the message is intercepted, the system is marked as under attack of type 4.

For discovering this attack we used the SocL formula

```
AG [attack4Performed(bounceAttack)] AF attack4DetectedByB(invalidContext,ctx) true
```

stating that it holds globally that if an attack of type 4 (bounce attack) is performed, then it is always detected by clinic B. Evaluation of this formula returns TRUE: Receiver's ARR discovers this attack since the SAML token is already stored. Notably, the attack is discovered when the second message is received. Indeed, the firstly received message is valid and, hence, the corresponding audit trail is recorded by Receiver in its ARR.

5 Conclusion

We presented a possible solution, based on international standards, for exchanging patients healthcare information amongst remote and disconnected clinics by using a car-transportation system, while preserving security and safety of patients healthcare data.

We defined the protocol and a specific threat model according to the use cases covered by international projects. We then formalised the protocol using the process calculus COWS and analysed it using the model checker CMC, to show that no attack to the protocol can be performed according to our threat model.

Our protocol can be easily extended to cover the *retrieve use case*, i.e. when a clinic *A* wants to use our protocol for retrieving documents from clinic *B*. To this purpose, for the patient with identifier *Susan*, clinic A sends

$$A \rightarrow_{VAN} B : A, B, msgId_1, ts2, token(K_A^+, user, \{ctx\}_{K_B^+})\}, \text{`}Susan\text{'}$$

Clinic *B* performs all the checks stated in our work and answers with the following message

$$B \rightarrow_{VAN} A : B, A, msgId_1, msgId_2, ts3, \{[\{\text{`}docSusan\text{'}\}_{K_B^-}]\}_{K_A^+}$$

This message comes back with the document '*docSusan*' via $\rightarrow_{VAN}$ (i.e. when the van comes back to the clinic). Another attack can be added to our threat model: the intruder intercepts the message travelling back to manipulate it or to be able to watch at unauthorized data. To cope with this attack, *Receiver* enforces *integrity* of the message by signing it. Notably, to preserve confidentiality, the document is also encrypted for the identity found in the key of the token.

It is worth noticing that the message exchanges requested by the protocol can be easily deployed by adding at the implementation presented in [5] the functionality given e.g. by a CD burner for writing messages on portable media.

Related Work. There are by now several national and international projects studying the problem of safely exchanging patients healthcare information [21–24]. These projects are becoming more and more complex and the study of the security of the adopted protocols could greatly benefit from the application of formal methods. On the other hand, formal methods have long been used to study protocols security properties (see, e.g., [25, 12]). More recently, these techniques have been adopted for the analysis of the security of web services. Many tools have been implemented with this aim [26–28] and have contributed finding numerous bugs in web service protocols. However, these tools lack of expressiveness in case of complex structured data to be exchanged along non standard communication channel, like $\rightarrow_{VAN}$. The TulaFale [26] tool aims at discovering possible rewrite attacks in SOAP messages travelling along TCP channels and at proposing changes to the structure of messages to prevent these attacks. In our setting we do not consider such attacks since our signatures are defined by international standards, which we do not intend to change. Actually, for the kind of attacks that we are interested in, it suffices to abstract messages as plain tuples (*à la* [29]). The Casper tool [30] gives the possibility to define properties of the communication channel. We have a similar scenario for the description of our different channels, such as $\rightarrow_{VAN}$. However, Casper's main focus is in proving the hierarchy of Lowe's authentication properties [16] that do not hold in the case of our channel representing the car-transportation system.

References

1. The IHE Initiative: IT Infrastructure Technical Framework (2009) http://www.ihe.net.
2. ANSI: Healthcare Information Technology Standard Panel, HITSP (2010) http://www.hitsp.org.
3. ACR-NEMA: Digital Imaging and Communications in Medicine (DICOM) (1995)
4. Health Level Seven organization: Hl7 standards (2009) http://www.hl7.org.
5. Masi, M., Pugliese, R., Tiezzi, F.: On Secure Implementation of an IHE XUA-Based Protocol for Authenticating Healthcare Professionals. In: ICISS. Volume 5905 of LNCS., Springer (2009) 55–70
6. OASIS Web Services Security TC: Web service security: SOAP message security (2006) http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.
7. Roessler, T., Yiu, K., Solo, D., Hirsch, F., Reagle, J., Eastlake, D.: XML signature syntax and processing version 1.1. W3C working draft, W3C (July 2009) http://www.w3.org/TR/2009/WD-xmldsig-core1-20090730/.
8. Reagle, J., Eastlake, D.: XML encryption syntax and processing version 1.1. W3C working draft, W3C (July 2009) http://www.w3.org/TR/2009/WD-xmlenc-core1-20090730/.
9. OASIS Web Services Security TC: WS-Trust 1.3 (2007) http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.pdf.
10. OASIS Security Services TC: Assertions and protocols for the OASIS security assertion markup language (SAML) v2.02 (2005) http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.
11. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. Technical report, DSI, Università di Firenze (2008) http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf. An extended abstract appeared in *ESOP*, LNCS 4421, Springer (2007) 33-47.
12. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2) (1983) 198–207
13. Broadfoot, P., Lowe, G.: On distributed security transactions that use secure transport protocols. Computer Security Foundations Workshop, IEEE **0** (2003) 141
14. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A model checking approach for verifying COWS specifications. In: FASE. Volume 4961 of LNCS., Springer (2008) 230–245
15. W3C: World Wide Web Consortium http://www.w3.org.
16. Lowe, G.: A Hierarchy of Authentication Specifications. In: CSFW, IEEE Computer Society (1997) 31–44
17. Rogers, T., Hadley, M., Gudgin, M.: Web services addressing 1.0 - core. W3C recommendation, W3C (May 2006) http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.
18. OASIS WSBPEL TC: Web Services Business Process Execution Language Version 2.0. (2007) http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
19. Masi, M., Pugliese, R., Tiezzi, F.: COWS specification of an XDM-based protocol for disconnected clinics in an healthcare scenario. Technical report, DSI, Università di Firenze (2010) `http://rap.dsi.unifi.it/cows/papers/cows-xdm.pdf`.
20. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL, ACM (2001) 104–115
21. The epSOS project: a european ehealth project http://www.epsos.eu.
22. ARGE-ELGA: Die österreich elektronische gesundheitsakte http://www.arge-elga.at.
23. GIP DMP: Dossier Médical Personnel A French Project http://www.d-m-p.org.
24. The South African Department of Health: the EHR project in south africa http://southafrica.usembassy.gov/root/pdfs/pepfar-hmis-docs/ndoh-e-hr-for-south-africa.pdf.

25. Abadi, M., Gordon, A.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. **148**(1) (1999) 1–70
26. Bhargavan, K., Fournet, C., Gordon, A., Pucella, R.: TulaFale: A Security Tool for Web Services. CoRR **abs/cs/0412044** (2004)
27. The AVANTSSAR EU project: Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures http://www.avantssar.eu.
28. The AVISPA project: Automated Validation of Internet Security Protocols and Applications http://www.avispa-project.org/.
29. Roscoe, B., Kleiner, E.: Web Services Security: a preliminary study using Casper and FDR. In: ARSPA. (2004)
30. Lowe, G.: Casper: A Compiler for the Analysis of Security Protocols. Journal of Computer Security **6**(1-2) (1998) 53–84