S. Gurgens, J. Lopez, and R. Peralta, "Analysis of E-commerce Protocols: Adapting a Traditional Technique", *International Journal of Information Security*, vol. 2, pp. 21-36, 2003.
NICS Lab. Publications: https://www.nics.uma.es/publications

International Journal of Information Security manuscript No.
(will be inserted by the editor)

Analysis of e-commerce protocols: Adapting a traditional technique

Sigrid Gürgens[1] **Javier Lopez**[2] **René Peralta**[3]

[1] Fraunhofer - Institute for Secure Telecooperation SIT
Darmstadt, Germany
guergens@sit.fraunhofer.de

[2] Computer Science Department
University of Malaga, Spain
jlm@lcc.uma.es

[3] Department of Computer Science
Yale University, U.S.
peralta-rene@cs.yale.edu

The date of receipt and acceptance will be inserted by the editor

Abstract We present the adaptation of our model for the validation of key distribution and authentication protocols to address some of the specific needs of protocols for electronic commerce. The two models defer in both the threat scenario and in the protocol formalization. We demonstrate the suitability of our adaptation by analyzing a specific version of the Internet Billing Server protocol introduced by Carnegie Mellon University. Our analysis shows that, while the security properties a key distribution or authentication protocol shall provide are well understood, it is often not clear which properties an electronic commerce protocol can or shall provide. We use the automatic theorem proving software "Otter" developed at Argonne National Laboratories for state space exploration.

Keywords: Security analysis, cryptographic protocol, automatic theorem proving, protocol validation, electronic commerce

1 Introduction

Experience shows that the correct design of communication protocols is hard because even the most carefully developed protocol specifications contain subtle errors. Design of cryptographic protocols, which play a key role in most applications over Internet, is an even more difficult task because networks are insecure in the sense that an adversary can monitor and alter messages exchanged by application users.

A protocol must not allow an attack to be performed unnoticed. However, it is not always clear what can be considered an attack. A protocol may be secure (i.e. achieve its goals) in one environment and insecure in a different environment. However, the environments a protocol is designed for are not always sufficiently specified (for example the question of type confusion - e.g. can a random number be taken for a key - is often not addressed). Moreover, protocols might be used in an environment they were not designed for (see [30] for the importance of clearly specifying the environment).

Thus, many of the largely used cryptoprotocols have been shown to fail under certain assumptions. In consequence, the use of formal methods that allow for the verification and validation of such protocols in a systematic and formal way has received increasing attention. In the last twenty years, active areas of research have developed around the problems of:

- developing design methodologies which yield cryptographic protocols for which security properties can be formally proven,
- formal specification and verification / validation of cryptographic protocols.

An early paper which addresses the first of these issues is [4]. Here, it is argued that protocols should not be defined simply as communicating programs but rather as sequences of messages with verifiable properties; i.e. security proofs can not be based on unverifiable assumptions about how an opponent constructs its messages. As with much of the research work done in the area of cryptography, this research does not adopt the framework of formal language specifications. Some other work along the lines of specifying provably secure protocols includes that of Bellare and Rogaway [3], Shoup and Rubin [37] (which extends the model of Bellare and Rogaway to smartcard protocols), Bellare, Canetti and Krawczyk

[2], and the work of Heintze and Tygar considering compositions of cryptoprotocols [17].

An approach based on the theory of formal languages is introduced in [34]. Here agents are modelled by using communicating automata and state components, and security mechanisms are formalized in terms of properties of abstract communication channels. A system is represented by all possible sequences of state transitions, and security properties of the system are defined in terms of properties of the possible sequences.

The second issue, formal specification and automatic verification or validation methods for cryptographic protocols, has developed into a field of its own. Partial overviews of this area can be found in [24], [22] and [31].

Most of the work in this field can be categorized as development of either logics for reasoning about security (so-called authentication logics) or of model checking tools or of theorem proving tools. These techniques aim at verifying security properties of formally specified protocols.

A seminal paper on logics of authentication is [7]. Work in this area has produced significant results in finding protocol flaws, but also appears to have limitations which will be hard to overcome within the paradigm. Confidentiality, for example, can not be expressed in terms of an authentication logic.

Model checking, on the other hand, involves the definition of a state space (typically modelling the "knowledge" of different participants in a protocol) and transition rules which define both the protocol being analyzed, the network properties, and the capabilities of an enemy. Initial work in this area can be traced to [12].

A prominent approach in the category theorem proving is [31,32,1]. Here protocols are specified as traces of events (an event being the sending or storing of a message) and an adversary is included by using operators *parts*, *analz* and *synth* on the messages. By induction over the possible traces it is then proven (under certain assumptions) that a protocol provides certain security properties.

Much has already been accomplished. A well-known software tool is the NRL Protocol Analyzer [23], which has been expanded to include some specification capabilities, thus being able to specify the security requirements of the SET protocol [26]. Other notable work includes Lowe's use of the Failures Divergences Refinement Checker in CSP [21] and Schneider's use of CSP [35]. Also to mention are the model checking algorithms of Marrero, Clarke, and Jha [22]. To the best of our knowledge, the model checking approach has been used almost exclusively for verification of cryptographic protocols.

Verification can be achieved efficiently if simplifying assumptions are made in order to obtain a sufficiently small state space. Verification tools which can handle infinite state spaces must simplify the notions of security and correctness to the point that proofs can be obtained using either induction or other logical means to reason about infinitely many states. Both methods have produced valuable insight into ways in which cryptographic protocols can fail.

The problem, however, is not only complicated but it is also evolving. The "classical" work has centered around proving security of entities authentication and key-distribution protocols, focusing on message properties such as key freshness, message confidentiality, and message origin authentication. Currently, with the advent of electronic commerce applications, cryptographic protocols are being adapted to implementing commercial transactions.

This new generation of protocols imposes higher requirements on security issues. Secure electronic commerce functions such as transaction certification, notarization, operational performance, commerce disposal, anonymity, auditing, etc., are necessary for a successful deployment of electronic commerce transactions. These functions in turn produce requirements like authentication of sender and/or receiver of a message, non repudiation, certification of submission and delivery, timestamping, detection of tampering, electronic storage of originality-guaranteed information, approval functions, access control, etc.

In addition, electronic commerce transactions involve an increasing number of agents that participate in the same protocol: issuers, cardholders, merchants, acquirers, payment gateways, Certification Authorities (CAs), Attribute Authorities, etc. Consequently, protocols for electronic commerce scenarios are becoming much more complicated, and are more likely to contain errors. Moreover, traditionally cryptoprotocols were designed with the assumption that the entities being involved in a protocol act honestly, thus only taking into account an outside attacker that tries to impersonate honest agents. In the scope of electronic commerce, this assumption is no longer realistic, as protocol participants, while acknowledging their proper names, may gain advantage from acting dishonestly. Thus the complexity of security analysis of electronic commerce protocols is increasing exponentially.

After Kailar's first analysis of an e-commerce protocol [19], there have been more attempts to model e-commerce protocols, some of which are being used in real applications, such as Kerberos [1], TLS/SSL [32] and Cybercash coin-exchange [6]. New analysis techniques were developed and existing ones were extended.

An example for the latter is the extension of the CSP model to cover non-repudiation [36], a security requirement of electronic commerce protocols. The outside intruder was removed from the model and the protocol participants were given the ability to fake messages. However, it is not clear whether this model can be extended to malicious agents additionally intercepting all messages.

We believe that already existing analysis techniques can be successfully adapted, and hence applied to elec-

tronic commerce protocols. In this paper we show how the method we developed for the analysis of classical key distribution and authentication protocols can be adapted to cover specific needs of e-commerce protocols. As an example protocol we chose the IBS protocol the various versions of which were first introduced in [29]. Here, quite a number of security requirements are explained which were useful for our work. On the other hand, any e-commerce protocol would have served as example to explain the extension of our methods.

The structure of the paper is as follows. Section 2 reviews some of the flaws of classical cryptographic protocols. Section 3 describes the respective communication and attack model. Section 4 explains how the formalization of protocol instantiations is performed. Section 5 shows the analysis of an standardized authentication protocol for smartcards with a digital signature application. Section 6 explains how our model is extended when considering new features that are typical in e-commerce scenarios. Sections 7 reviews the IBS protocol and describes our formalization, which is used for the analysis of the protocol in section 8. Finally, section 9 concludes the paper.

2 Classical Cryptographic protocols

In this section we explain what we call classical cryptographic protocols such as key distribution protocols and give an example. The participants of such a protocol will be called "agents". Agents are not necessarily people. They can be, for example, computers acting autonomously or processes in an operating system. Secrecy of cryptographic keys cannot be assumed to last forever; hence, pairs of agents must periodically replace their keys with new ones. That is the aim of a key-distribution protocol, which produces and securely distributes "session" keys. This is often achieved with the aid of a trusted key distribution server S (this is the mechanism of the widely used Kerberos System [20]).

The general format of these protocols is the following:

- Agent A wants to obtain a session key for communicating with agent B.
- It then initiates a protocol which involves S, A, and B.
- The protocol involves a sequence of messages which, in theory, culminate in A and B sharing a key K_{AB}.
- The secrecy of K_{AB} is supposed to be guaranteed despite the fact that all communication occurs over insecure channels.

To illustrate how easily such a protocol can fail we describe the first attack [9] on the well-known Needham-Schroeder protocol introduced in [27]. In what follows, a message is an ordered tuple $(m_1, m_2, \ldots, m_r)$ of concatenated messages m_i, viewed as an abstract object at an abstraction level where the agents can distinguish between different message parts. $\{m\}_K$ denotes the encryption and digital signature, respectively, of the message m using key K. Messages that are neither a concatenation of messages nor a ciphertext are called atomic messages. Commonly a protocol step s in which A sends to B message m is denoted by $s.\ A \longrightarrow B :\ m$. We call one execution of a protocol a protocol run. The symmetric Needham-Schroeder protocol uses symmetric encryption and decryption functions (where encryption and decryption are performed with the same key). The protocol steps are as follows:

1. $A \longrightarrow S :\ A, B, R_A$
2. $S \longrightarrow A :\ \{R_A, B, K_{AB}, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
3. $A \longrightarrow B : \{K_{AB}, A\}_{K_{BS}}$
4. $B \longrightarrow A : \{R_B\}_{K_{AB}}$
5. $A \longrightarrow B : \{R_B - 1\}_{K_{AB}}$

In the first message agent A sends to S its name A, the name B of the desired communication partner and a random number R_A which it generates for this protocol run[1]. S then generates a ciphertext for A, using the key K_{AS} that it shares with A. This ciphertext includes A's random number, B's name, the new key K_{AB}, and a ciphertext intended for B. The usage of the key K_{AS} shall prove to agent A that the message was generated by S. The inclusion of R_A ensures A that this ciphertext and in particular the key K_{AB} is generated after the generation of R_A, i.e. during the current protocol run. Agent A also checks that the ciphertext includes B's name, making sure that S sends the new key to B and not to someone else, and then forwards $\{K_{AB}, A\}_{K_{BS}}$ to B.

For B the situation is slightly different, as it learns from A's name who it is going to share the new key with, but nothing in this ciphertext can prove to B that this key is indeed new. According to the protocol description in [27] this shall be achieved with the last two messages. The fact that B's random number $R_B - 1$ is enciphered using the key K_{AB} shall convince B that this is a newly generated key. However, the protocol attack introduced by Denning and Sacco makes it clear that this conclusion can not be drawn. In fact, all that B can deduce from message 5 is that the key K_{AB} is used by someone other than B in the current protocol run.

The attack assumes that an eavesdropper E monitors one run of the protocol and stores $\{K_{AB},\ A\}_{K_{BS}}$. Since we assume the secrecy of agents' session keys holds only for a limited time period (after all, it takes only hours to break a simple 64 bit DES key which still serves well in many applications if used only for a short period of

[1] In this paper, "random numbers" are abstract objects with the property that they cannot be generated twice and cannot be predicted. It is not trivial to implement these objects, and protocol implementations may very well fail because the properties are not met. However, tackling this problem is beyond the scope of this paper.

time), we consider the point at which E gets to know the key K_{AB}. Then it can start a new protocol run with B by sending it the old ciphertext $\{K_{AB}, A\}_{K_{BS}}$ as message 3 of a new protocol run. Since B has no means to notice that this is an old ciphertext, it proceeds according to the protocol description above. After the protocol run has finished, it believes that it shares K_{AB} as a new session key with A, when in fact it shares this key with E.

There is no way to repair this flaw without changing the messages of the protocol if we do not want to make the (unrealistic) assumption that B stores all keys it ever used. So, this is an example of a flaw inherent in the protocol design.

Another type of flaw arises, as we will explain in later sections, from the particular protocol implementation (see [16]). Many protocol descriptions are vague about what checks are to be performed by the recipient of a message. Thus, a protocol implementation may be secure or insecure depending exclusively on whether or not a particular check is performed.

2.1 Attack models

The agents A, B, etc. participating in a classical cryptographic protocol are usually assumed to act honestly, the only threat being that of a hostile environment. A common way to model such an environment was introduced by Dolev and Yao in [12]. The intruder is modelled with maximal power, as he/she:

- can obtain any message passing through the network,
- is a legitimate agent, thus in particular can initiate a protocol run,
- can be addressed by any agent as the responder of a protocol run,
- can send any message it can generate to any agent of the system.

This model is widely used for the analysis of classical cryptographic protocols and is the basis for our communication and attack model, being explained in the next section, for this type of protocols. Figure 1 depicts a system with agents A, B, S and the intruder E.

In [40] the maximal power of the Dolev-Yao intruder is reduced to a so-called "Macchiavellian" intruder composed of self-interested collaborators that might be unwilling to share signature keys and other long-term secrets. This attack model is more adequate for e-commerce protocols than that of Dolev and Yao. It is shown that under certain assumptions (e.g. an agent will only accept messages having the format specified in the protocol, and agents do not test for inequality) the Macchiavellian model is attack-equivalent to the Dolev-Yao model, i.e. every attack mounted in the latter can be mounted in the former and vice versa.

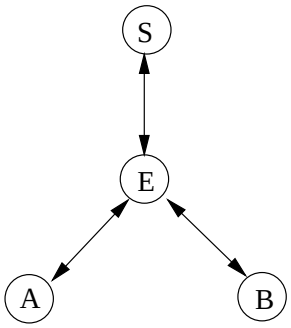

Fig. 1 The Dolev-Yao attack model

Other approaches to model the intruder are concerned with a situation where k out of n nodes of a network are compromised (see for example [39],[28]).

3 Our communication and attack model

The security analysis of protocols does not deal with weaknesses of the encryption and decryption functions from a cryptanalytic point of view. In what follows we assume "perfect encryption", i.e. we assume in particular that the following security properties hold:

1. Messages encrypted using a function f and a secret key K can only be decrypted with the inverse function f^{-1} and key K^{-1}.
2. A key can not be guessed (during the period of its validity).
3. Given m, it is not possible to find the corresponding ciphertext for any message containing m without knowledge of the key.

The first two items describe properties of encryption functions and keys that are generally accepted for protocol analysis. In particular, every protocol will be found insecure if an attacker can simply guess a key being used. However, we do assume that already discarded keys are guessable. The third property which Boyd called the "cohesive property" in [5] does not hold in general. Boyd and also Clark and Jacob (see [8]) show that under certain conditions, particular modes of some cryptographic algorithms allow the generation of a ciphertext without knowledge of the key.

These papers were important in that they drew attention to hidden cryptographic assumptions in "proofs" of security of cryptoprotocols. In fact, it is clear now that the number (and types) of hidden assumptions usually present in security proofs is much broader than what Boyd and Clark and Jacob point out.

All communication is assumed to occur over insecure communication channels. We model these in accordance with the Dolev-Yao attack model and assume that there is a further agent E that intercepts all messages sent by others. After intercepting, E can change the message

to anything it can compute. This includes changing the destination of the message and the supposed identity of the sender. Depending on the decisions taken for a specific analysis, E may or may not act as a valid agent of the system, i.e. may or may not share a symmetric key K_{ES} with S and/or own a key pair (K_E, K_E^{-1}) for an asymmetric algorithm. As a valid agent, E can be allowed to initiate communication either with other agents or S. Our model leaves these decisions open to be fixed for a particular analysis run.

E intercepts all messages and the agents and S only receive messages sent by E. What E can send depends on what it is able to generate, and this in turn depends on what E knows.

The knowledge of E can be recursively defined as follows:

1. E may or may not know the names of the agents.
2. E knows the keys it owns.
3. E knows its own random numbers.
4. E knows every message it has received.
5. E knows every part of a message received (where a ciphertext is considered one message part).
6. E knows everything it can generate by enciphering data it knows using data it knows as a key.
7. E knows every plaintext it can generate by deciphering a ciphertext it knows, provided it knows the inverse of the key and the encryption function used as well.
8. E knows every concatenation of data it knows.
9. E knows the origin and destination of every message.
10. At every instance of the protocol run, E knows the "state" of every agent, i.e. E knows the next protocol step they will perform, the structure of a message necessary to be accepted, E knows in principle what they will send after having received (and accepted) a message, etc.

At every state in a possible run, after having received an agent's message, E has the possibility to forward this message or to send whatever message it can generate to one of the agents or S. Obviously, the state space will grow enormously. Thus, we must limit the concept of knowledge if the goal is to produce a validation tool which can be used in practice. Therefore we restrict the definition in various ways in order to make E's knowledge set finite. For example, if the protocol to be analyzed uses a public key algorithm, we do not allow E to generate ciphertexts using data as a key which is neither the private nor the public part of a key pair. We do this on an ad-hoc basis. This problem needs further exploration. It is important to note that leaving this part of our model open allows for a very general tool. This is in part responsible for the success of the tool, as documented in later sections. Also to note is that the above knowledge rules imply that E never forgets. In contrast, the other agents forget everything which they are not specifically instructed to remember by the protocol description.

For the analysis of classical cryptographic protocols we assume that agents can not distinguish between different message types, that is, will for example accept a nonce as a key. This may or may not be relevant for the particular environment the protocol will be used in. Smartcard applications for example allow to use message structures like ASN.1 but often use implicitly defined structures. This is the case for the authentication protocol of the German DIN standard for digital signature smartcards [10] that we will discuss in section 5. Here the structure of the messages is fixed and the specification of the algorithm includes the specification of the message structure. Other smartcard applications use so-called *headers* to define the message structure. The header specifies the type and length of each message part and the message is interpreted accordingly. In these cases the messages itself do not carry any type information. However, as we will later see, it is easy to include types of messages into the model.

4 Formalization of protocol instantiations

As the title of this section suggests, we make a distinction between a protocol and a "protocol instantiation". We deem this a necessary distinction because protocols in the literature are often under-specified. A protocol may be implemented in any of several ways by the applications programmer.

For the security analysis of such protocols we used Otter (Organized Techniques for Theorem-proving and Effective Research) developed by Argonne National Laboratories. Otter is a resolution-style theorem prover applying to statements written in first-order logic with equality. However, we have not used Otter for proving any security properties on protocols, but for exploring the state space of a given protocol. For an extensive description of how Otter works and what kind of theories it can be applied to we refer the reader to [42]. In the following, we will explain our use of Otter as an automatic state exploration machine.

For formalizing a protocol, the environment properties and possible state transitions, we use first order logic functions, predicates, implications and the usual connectives. At any step of a protocol run predicates like

$$state(xevents, xEknows, \ldots, xwho, xto, xmess, \ldots)$$

describe a possible state of the system. Each predicate contains the list *xevents* of actions already performed, the list *xEknows* of data known by the intruder E, and additionally, for each agent and each run it assumes to be taking part in, components *xagent, xrun, xstep, xrandom, xkey,* etc., that describe the state of the agent in the particular run (e.g. data like session keys, nonces, etc., the agent has stored so far for later use, and the next step it will perform). In general the predicates can

include as many agents' states and protocol runs as desired, but so far we have restricted this information to at most two agents, a server and the attacker in the context of two interleaving protocol runs. Furthermore, the predicates contain, among other data used to direct the search, the agent *xwho* which sent the current message, the intended recipient *xto* and the message *xmess* sent.

Using these predicates we write formulas which describe possible state transitions. The action of S when receiving the first message of the Needham-Schroeder symmetric key protocol described in section 2 can be formalized as follows:

$$
\begin{aligned}
&send(xevents, xEknows, \ldots, xwho, xto, xmess, \ldots)\\
&\wedge\\
&xto = S \wedge xwho = E\\
&\wedge\\
&is\_agent(el(1, xmess)) \wedge is\_agent(el(2, xmess))\\
&\rightarrow\\
&send([\ldots|xevents], xEknows, \ldots, S, el(1, xmess),\\
&\quad [scrypt(key(el(1, xmess), S),\\
&\quad [el(3, xmess), el(2, xmess), new\_key,\\
&\quad scrypt(key(el(2, xmess), S),\\
&\quad [new\_key, el(1, xmess)])])], \ldots)
\end{aligned}
$$

where $[\ldots|xevents]$ includes the current send action, $el(k, xmess)$ returns the kth part of the message *xmess* just received, and *new_key* stands for a number uniquely generated for each protocol run. By checking that the message was sent by E ($xwho = E$) we model the fact that agents only receive messages sent by E.

Whenever a message is intercepted by E (indicated by a formula $send(\ldots) \wedge xwho \neq E$), an internal procedure adds all that E can learn from the message to the knowledge list *xEknows*. The result of this procedure is a predicate $added\_all\_knows(\ldots)$. Implications $added\_all\_knows(\ldots) \rightarrow send(\ldots, xwho, \ldots, message1, \ldots) \wedge send(\ldots, xwho, \ldots, message2, \ldots) \wedge \ldots$ considering $xwho = E$ formalize that after having intercepted a particular message, E has several possibilities to proceed, i.e. each of the predicates $send(\ldots, E, \ldots, message, \ldots)$ constitutes a possible continuation of the protocol run. The messages E sends to the agents are constructed using the basic data in *xEknows* and the specification of E's knowledge as well as some restrictions as discussed in the previous section.

Symmetric keys shared by agents X and Y are denoted by *key(X,Y)*, *key(X,pub)* denotes the public key and *key(X,priv)* the private key of agent X. We use different symbolic functions (i.e. functions that are not evaluated) to distinguish between different cryptographic algorithms. In this paper, *scrypt(k,m)* denotes the symmetric encryption of message m with key k. Similarly, *sdecrypt(k,m)* formalizes the decryption of message m with the same key. An asymmetric algorithm being used is indicated by *acrypt(k,m)* and a digital signature by *sig(k,m)*.

We use equations to describe the properties of the encryption and decryption functions, keys, etc., used in the protocol. For example, the equation

$$sdecrypt(x, scrypt(x, y)) = y$$

formalizes symmetric encryption and decryption functions where every key is its own inverse. These equations are used as demodulators in the deduction process.

The above formulas constitute the set of axioms describing the particular protocol instantiation to be analyzed. An additional set of formulas, the so-called "set of support", includes formulas that are specific for a particular analysis. In particular, we use a predicate describing the initial state of a protocol run (the agent that starts the run, with whom it wants to communicate, the initial knowledge of the intruder E, etc.) and a formula describing a security goal. An example of such a security goal is "if agent A believes it shares a session key with agent B, then this key is not known by E", formalized by $-(state(\ldots) \wedge xAstep = finished \wedge xAagent = B \wedge in\_list(xAkey, xEknows))$.

Using these two formula sets and logical inference rules (specifically, we make extensive use of hyper resolution), Otter derives new valid formulas which correspond to possible states of a protocol run. The process stops if either no more formulas can be deduced or Otter finds a contradiction which involves the formula describing the security goal. When having proved a contradiction, Otter lists all formulas involved in the proof. An attack on the protocol can be easily deduced from this list.

4.1 Known and new attacks

We used Otter to analyze a number of protocols. Whenever we model a protocol we are confronted with the problem that usually the checks performed by the parties upon receiving the messages are not specified completely. This means that we must place ourselves in the situation of the implementer and simply encode the obvious checks. For example, consider step 3 of the Needham-Schroeder protocol explained in section 2:

3. $A \longrightarrow B : \{K_{AB}, A\}_{K_{BS}}$

Here is a case where we have to decide what B's checks are. We did the obvious:

- B decrypts using K_{BS}.
- B checks that the second component of the decrypted message is some valid agent's name.
- B assumes, temporarily, that the first component of the decrypted message is the key to be shared with this agent.

However, Otter determined that this was a fatal flaw in the implementation. We call the type of attack found the "arity" attack, since it depends on an agent not checking the number of message blocks embedded in a

cyphertext. The arity attack found by Otter works in the following way. In the first message E impersonates B (denoted by E_B) and sends to S:

1. $E_B \longrightarrow S : B, A, R_E$

Then, S answers according to the protocol description and sends the following message to B:

2. $S \longrightarrow B : \{R_E, A, K_{AB}, \{K_{AB}, B\}_{K_{AS}}\}_{K_{BS}}$

The message is intercepted by E and then passed on to B, and B interprets it as being the third message of a new protocol run. Since this message passes the checks described above (B does not notice that the ciphertext contains more than 2 items), B takes R_E to be a new session key to be shared with A and the last two protocol steps are as follows:

4. $B \longrightarrow E_A : \{R_B\}_{R_E}$
5. $E_A \longrightarrow B : \{R_B - 1\}_{R_E}$

where the last message can easily be generated by E since it knows its random number sent in the first protocol step (see the appendix for a formalization of B's actions).

It is interesting to note that an applications programmer is quite likely to not include an "arity-check" in the code. This is because modular programming naturally leads to the coding of routines of the type

$$get(cyphertext, key, i)$$

which decodes *cyphertext* with *key* and returns the i^{th} element of the plaintext.

By including the arity check in B's actions when receiving message 3, we analyzed a different instantiation of the same protocol. This time Otter found the well-known Denning-Sacco attack [9], where the third message of an old protocol run is replayed to B and the attack run ends with B and the intruder E sharing an old session key that possibly was broken by E.

In order for this attack to work, Denning and Sacco removed the assumption made in [27] by Needham and Schroeder that K_{AB} is unpredictable, fresh and only known to A and B. The arity attack found by Otter, however, works in an environment that fulfills all assumptions stated in [27]. All the intruder needs to be able to is to read messages from and add messages to the network. In particular, he does not share a key with the server, thus is not a regular agent of the system.

It seems that many protocols may be subject to the "arity" attack: our methods found that this attack is also possible on the Otway-Rees protocol (see [15] for a detailed description of the protocol and its analysis) and on the Yahalom protocol, as described in [7].

Among the known attacks which were also found by our methods are the Lowe [21] and the Meadows [25] attacks on the Needham-Schroeder public key protocol and the Simmons attack on the TMN protocol [41] described in [38]. For the latter we relaxed the assumption of perfect encryption and modeled the homomorphic property of cryptoalgorithms like RSA [33] that allows to generate a ciphertext by multiplication. For this, we introduced symbolic functions *mult* and *div* and an equation *crypt(k,mult(m,n)) = mult(crypt(k,m),crypt(k,n))*. By adding the intruder's respective capabilities we found the attack described in [41].

5 A different scenario

All the above mentioned protocols were analyzed using a scenario where we have several honest agents and the attacker. In this section we will describe the analysis of a first draft of an authentication protocol for smartcards with a digital signature application that was standardized by the German standardization organization DIN in [11] (an earlier paper on this work is [14]). For this reason we chose a somewhat restricted scenario where the smartcard (Integrated Circuit Card, ICC) is the only honest agent. The intruder plays the role of the smartcard reader (Interface Device, IFD).

Generally it is assumed that the digital signature application being for example located in a PC controls the smartcard by using the smartcard reader. By sending the respective commands to the IFD the application causes the IFD to communicate with the smartcard accordingly. The actual computations (e.g. of signatures) are accomplished by either the smartcard or the IFD (depending on the capabilities of the smartcard). For simplicity we will use IFD and the digital signature application located in the PC synonymously. We will also assume that all computation is performed in the smartcard. Note that the smartcard always acts as the responding entity, i.e. never sends commands on its own.

The protocol uses a signature algorithm with the possibility of message recovery (which allows to retrieve the original message from the signature) for authentication purposes. After the successful run of the protocol the smartcard and the smartcard reader shall share a session key that they can then use for authentic and confidential communication. The first protocol phase proceeds as follows:

1. $IFD \longrightarrow ICC : SELECT\_FILE < name >$
2. $ICC \longrightarrow IFD : OK$
3. $IFD \longrightarrow ICC : READ\_BINARY$
4. $ICC \longrightarrow IFD : OK$
5. $IFD \longrightarrow ICC : MSE($
   $SET <$ *key-reference*$(PK_{CA}) >)$
6. $ICC \longrightarrow IFD : OK$
7. $IFD \longrightarrow ICC : VERIFY\_CERTIFICATE$
   $(Cert_{IFD})$

In this first phase of the authentication process the smartcard reader IFD selects the file that contains the certificate of the smartcard ICC (step 1) and reads it (step 3). After having checked that the certificate is valid (for simplicity we assume here that both IFD and ICC have been issued a certificate by the same authentication authority CA), IFD determines the public key PK_{CA}

that the smartcard shall use for the subsequent certificate check with the command MSE (*Manage Security Environment*, step 5) and then sends its own certificate $Cert_{IFD}$ (step 7). When ICC finds IFD's certificate valid it stores the certificate's public key (i.e. the public key PK_{IFD} of IFD) together with the certificate holder's distinguished name (ID_{IFD}) in a temporary file for later use. It also stores the access right to a file with the message to be displayed after successful authentication. The certificates of ICC and IFD differ in that the smartcard is not allowed to access this file. Then the authentication phase of the protocol starts. In a first draft of the standard [11] this consisted of the following steps:

9. $IFD \longrightarrow ICC : MSE$ $(SET(< \textit{key-reference} >))$
10. $ICC \longrightarrow IFD : OK$
11. $IFD \longrightarrow ICC : INTERNAL\_AUTH(R_{IFD},$ $ID_{IFD})$
12. $ICC \longrightarrow IFD : acrypt(PK_{IFD}, sig(SK_{ICC},$ $(K1, hash(K1, R_{IFD},$ $ID_{IFD}))))$
13. $IFD \longrightarrow ICC : GET\_CHALLENGE$
14. $ICC \longrightarrow IFD : R_{ICC}$
15. $IFD \longrightarrow ICC : MSE$ $(SET(< \textit{key-reference} >))$
16. $ICC \longrightarrow IFD : OK$
17. $IFD \longrightarrow ICC : EXTERNAL\_AUTH$ $(acrypt(PK_{ICC}, sig(SK_{IFD},$ $(K2, hash(K2, R_{ICC},$ $ID_{ICC}),' BC')))))$

In step 9 IFD determines the signature key SK_{ICC} to be used by the smartcard for signature generation. Then it requests the ICC to authenticate itself by sending the command $INTERNAL\_AUTH$ in step 11 with its challenge R_{IFD} and its distinguished name ID_{IFD}, where ID_{IFD} determines the public key to be used by the smartcard for the encryption of the signature. (The smartcard searches for ID_{IFD} in the temporary file in which it has stored the certificate data received in step 7, and uses the respective key.) The smartcard now generates a secret $K1$ to be used later as part of the new session key and then generates a signature using the key determined in step 9. Essentially the following data are signed: the key part $K1$ concatenated with some hash value containing $K1$, IFD's random number R_{IFD}, and IFD's distinguished name ID_{IFD}, both received in the previous step. The exact message format is not of interest for what follows. This signature is then encrypted using the key determined by ID_{IFD} and the resulting ciphertext is sent to IFD. IFD decrypts the ciphertext using its own private key and checks whether the signature is valid. Then IFD requests ICC's challenge R_{ICC} (step 13) and again determines a key with the command MSE which the smartcard shall use subsequently to verify IFD's signature (step 15). Finally, in step 17, IFD generates its own authentication token by signing the equivalent data using its signature key SK_{IFD} and encrypting the signature with ICC's public key PK_{ICC}. For this it generates the second secret part $K2$ of the session key to be used later on. The smartcard uses the key determined in step 9 for decrypting the ciphertext and the key determined in step 15 for checking the signature.

After successful mutual authentication the IFD is allowed to read the display message of the smartcard which shall inform the smartcard owner of the positive result of the authentication. This part of the communication is already processed in secure messaging mode, which means that each data field carries a cryptographic checksum, and additionally the display message is encrypted using a symmetric algorithm. The key K used for this is a combination of the secret parts $K1$ and $K2$ exchanged in the authentication phase.

5.1 Our analysis

Since in the above explained first draft of the standard the possible use of the keys was not explicitly restricted in any way, we allowed the smartcard to use any key for any purpose, except that it only uses the CA's public key for checking certificates. As was pointed out to us later, this is not in accordance with [18], a DIN standard describing security related smartcard commands such as MSE. However, we believe that a designer of a smartcard operating system may not be aware of the key restrictions if not explicitly mentioned, which justifies our formalization.

For verifying a certificate we let the smartcard only check the certificate's signature (the verification of certificates is out of the scope of [11]), which is the way this will probably be implemented. (Note that the smartcard is not able to check a validity date as it does not own an internal clock.)

We modelled the smartcard reader as being a malicious interface device that does not own a valid certificate. The first fact that our analysis showed is that the smartcard will accept its own certificate as the IFD's certificate, thus storing its own public key to be used in the subsequent authentication phase. This is possible since the smartcard does not check that the certificate owner is someone other than itself.

Second, we found that during the phase of authentication the following can happen:

9. $IFD \longrightarrow ICC : MSE(SET$ $(< \textit{key-reference}(SK_{ICC}) >))$
11. $IFD \longrightarrow ICC : INTERNAL\_AUTH$ (R_{IFD}, ID_{ICC})

IFD correctly determines SK_{ICC} as the signature key to be used by the smartcard in step 9, but with ID_{ICC} as the second item of the command *INTERNAL_AUTH* in step 11 it requests the smartcard to use its own public key as encryption key (the key that was

stored in step 7 with the certificate data). Since the protocol is designed for an algorithm with message recovery, where $acrypt(PK_X, sig(SK_X, m)) = m$ for any agent X, as a result the smartcard does not send its encrypted signature in the next step. Instead, it sends plaintext, in particular it sends the first secret part $K1$ of the session key in plaintext:

12. $ICC \longrightarrow IFD : K1, hash(K1, R_{IFD}, ID_{IFD})$

Now IFD requests the smartcard's challenge as described in the standard, but then, instead of determining the correct public key for signature verification, it demands the smartcard to use its own signature key for this purpose:

13. $IFD \longrightarrow ICC : GET\_CHALLENGE$
14. $ICC \longrightarrow IFD : R_{ICC}$
15. $IFD \longrightarrow ICC : MSE(SET$ $(key\text{-}reference(SK_{ICC}))$

Then it generates a "signature" by using the smartcard's public key received in step 4 of the protocol and encrypts this data, again using the same key. The resulting ciphertext is then sent to the smartcard:

17. $IFD \longrightarrow ICC : EXTERNAL\_AUTH$ $(acrypt(PK_{ICC}, sig(PK_{ICC},$ $(K2, hash(K2, R_{ICC},$ $ID_{ICC})))))$

The smartcard uses its private key SK_{ICC} (determined in step 9) to decrypt the ciphertext and then, according to the key having been specified in step 15, uses this key again for checking the "signature". Since the keys PK_{ICC} and SK_{ICC} correspond to each other the smartcard finds the signature valid and accepts IFD's authentication data.

Now IFD can try to read the display message of the smartcard but will not succeed, since the certificate data stored by the smartcard in the first phase of the protocol is the smartcard's own data, and the smartcard is not allowed to access its display message. However, we may assume that a malicious interface device will be able to present the necessary confirmation of the successful authentication on its own.

At the end of this protocol run the malicious IFD owns the two secrets that form the session key to be used by the smartcard subsequently for secure messaging purposes. If the smartcard accepts the above protocol run as a successful authentication, i.e. if it does not insist on the display message being read successfully, IFD is now able to perform whatever action the application allows. In any case the fact whether or not the display message was read successfully should have no impact on the security of the two earlier protocol phases.

As we pointed out already at the beginning of this section, the above described use of the keys is not in accordance with [18]. In general, signature keys may not be used for encryption and vice versa. In particular, the smartcard is not allowed to use its own public key for the encryption of its signature in step 12. However, after the presentation of our analysis the standard was changed to explicitly not allowing the above described attack. Notes were added that explain which keys the smartcard is not allowed to use for which purpose. Additionally, key determination by IFD was changed: in the new version (of November 30th, 1998) one command $MSE(SET(< key\text{-}reference1, key\text{-}reference2 >))$ is used in step 9 to determine both keys to be used by the smartcard later on (steps 15 and 16 are now obsolete), which prevents the use of the smartcard's signature key for the signature verification during the $EXTERNAL\_AUTH$ command. This new version of the standard has not yet been analyzed.

Nevertheless, in the new version of the standard it is still possible to present the smartcard its own certificate in the first protocol phase. In this case the authentication process will break off with the $INTERNAL\_AUTH$ command, but it might be hard in practice to find the reason for this, as the certificate was verified as being valid. We believe that it is advisable to have the smartcard only accept those certificates that are reasonable in an authentication process.

6 Extending the model

As already pointed out in previous sections, e-commerce protocols are much more complex than traditional cryptographic protocols with respect to security requirements and threats. In traditional protocols there is at most one agent assumed to act dishonestly, namely the intruder which in many cases is even not considered a valid agent of the system. However, in e-commerce scenarios we have a situation where agents do not necessarily trust each other. A customer in a payment system might want to gain advantage of the system by not paying the agreed amount of money, a service provider may try to get away with not sending the service. So it is not so much the abilities of an outside intruder we are interested in but the abilities of the agents themselves that are taking part in a protocol.

As to security properties the protocols shall provide, we still have to deal with confidentiality and authenticity, but furthermore there are properties such as non-repudiation, anonymity, accountability, notarization, etc. that pose different questions. So besides the fact that protocol flaws may arise because of type confusion, we are interested in questions like "Is it possible for the client to receive the service without ever delivering the payment?".

It is clear that the difference in security threats between classical and e-commerce protocols induces differences in the formalization of the attack scenario. For the rest of the paper, we will refer to the analysis model for classical cryptoprotocols as the "classical model" and to

that for e-commerce protocols as to the "e-commerce model".

Our communication and attack model for the analysis of e-commerce protocols (the e-commerce model) is in some sense similar to the "Macchiavelli" model mentioned in section 2.1. However, we do not compose dishonest agents to one Macchiavellian intruder, and we do not restrict dishonest agents' actions.

We assume that additionally to an outside intruder, any number of agents participating in the system can be dishonest. The capability of agents to generate faked messages is based on their knowledge, which is derived along the "rules" given in section 3. Furthermore, we also allow a situation where dishonest agents are capable of intercepting messages as well. This may or may not be relevant, depending on the environment in which to use the protocol.

In general our model allows an unrestricted number of dishonest agents, but so far we have restricted analysis runs to at most two. In the case of two dishonest agents, we assume that they cooperate, i.e. exchange whatever knowledge they have, except that they do not make available to each other their private keys. Modelling two dishonest agents not cooperating only leads to more results than modelling just one if messages maliciously generated by one agent (i.e. messages not being specified in the protocol) can be used by the other for some attack, an event we considered unlikely.

As in the classical model, we are not concerned with the security of the algorithms that are being used in the protocols, i.e. we again assume perfect encryption. But in contrast to the analysis of classical protocols where we assume that agents are not able to distinguish between different types, in the e-commerce model we add types to the messages. Thus the agents can be modelled as being able or not being able to distinguish between different types of messages. This allows the analysis to concentrate on issues specific to e-commerce protocols rather than searching for protocol flaws caused by type confusion, but it keeps the search for type confusion based flaws possible.

Consequently, in the analysis runs performed so far we allowed dishonest agents only to send messages in the correct format. Thus a message identification number will not be sent as a price, a signature will not be used as a random number, etc.

For the formalization of e-commerce protocols we have made a few changes to predicates and formulas. The most important ones are listed below:

- The predicates additionally contain a list of agents that act dishonestly and a list of agents that cooperate (which allows for future extension of the number of cooperating agents). Furthermore, agents own a list of keys and a list for keeping information they extracted from messages not intended for them.
- In the classical cryptoprotocols we analysed so far there was no need to model an agent sending two messages subsequently. However, in e-commerce protocols the situation may be different, in particular when allowing for cooperating malicious agents whose actions are not fixed by the protocol description. Thus we introduced a "dummy message": If agent X sends first a message to agent Y and then a message to agent Z, we model this by having Y answer to X with *dummy*, after which X can proceed with sending the second message to Z.
- We model the interception of messages by dishonest agents principally in the same way as in the classical model: Whenever a message is sent by an honest agent ($send(\ldots) \wedge -in\_list(xwho, xbadguys)$), an internal procedure adds all that a dishonest agent P can learn from the message to the respective knowledge list *xPknows*. If there are cooperating agents, their knowledge lists are exchanged. Finally for each of the dishonest agents the messages to be sent are constructed, on the basis of the agent's knowledge and keys he owns. The messages are then sent to those agents that will accept the message with respect to the format (e.g. an order will not be send to a client, but only to a service provider). If we want to analyse a protocol in an environment where agents can not intercept messages, we skip the internal procedure of knowledge extraction and connect the reception of messages directly with the generation of faked messages by way of demodulators such as $in\_list(xto, xbadguys) \longrightarrow send(\ldots) = construct\_messages(\ldots)$.

7 The Internet Billing Server Protocol

Using the above described model, we analyzed the Internet Billing Server Protocol (IBS protocol) developed by Carnegie Mellon University and described in [29]. Actually, [29] includes several different versions of protocol templates where the precise format of messages is left open. We chose the version that uses only asymmetric algorithms and made those changes we deemed necessary for a reasonable analysis.

The protocols were designed to enable sales of goods to be delivered over the network. There are three different types of agents: service providers, users (those agents which buy the goods) and the billing server, trusted by all agents, which handles the money transfer. Both service provider SP and user *User* register with the billing server BS, which means in particular that accounts are opened and key pairs are issued by BS.

The protocol assumes that a signature algorithm with message recovery is used (an algorithm that allows to decipher a signature in order to get the plaintext again). $\{message\}_{SK_X}$ denotes the message signed with the private key of agent X. In the following, we describe our

version of the IBS protocol. It consists of two phases, the first of which is price delivery, where the user asks some service provider for a price and the service provider answers with a price.

1. $User \longrightarrow SP \quad : \{ID, \text{request}, servicename\}_{SK_{User}}$
2. $SP \longrightarrow User : \{ID, price\}_{SK_{SP}}$

The second phase is the service provision and payment phase:

3. $User \longrightarrow SP : \{\{ID, price\}_{SK_{SP}}, ID, price\}_{SK_{User}}$
4. $SP \longrightarrow BS : \{\{\{ID, price\}_{SK_{SP}}, ID, price\}_{SK_{User}}\}_{SK_{SP}}$
5. $BS \longrightarrow SP \quad : \{ID, \text{authorization}\}_{SK_{BS}}$
6. $SP \longrightarrow User : \{ID, service\}_{SK_{SP}}$
7. $User \longrightarrow SP \quad : \{ID, \text{ackn}\}_{SK_{User}}$
8. $SP \longrightarrow log \quad : \{\{ID, \text{ackn}\}_{SK_{User}}\}_{SK_{SP}}$
9. $SP \longrightarrow BS \quad : log$

First, in step 3, the user orders the service. In step 4 SP asks BS for authorization of this service, which means in particular that BS checks that the user's account provides enough money. If so, BS transfers the price for the service from the user's to the service provider's account and keeps a record with all data relevant for this transaction. Then it authorizes the service (step 5) and the service is delivered (step 6). The user's last action is to send a message acknowledging that he received the service. The service provider collects all acknowledgement messages in a log file (step 8) which is sent to the billing server in off-peak hours (step 9). When receiving the log file, the billing server checks that the acknowledgement messages in the log file match the previously performed money transfers.

We added a message ID to all messages of the original version, and the constant "*request*" and the variable *servicename* to the first message, taking into account that a price will depend on the time it was given, the user it was given to, the service, and the service provider, that an authorization refers to a specific amount of money to be paid by a specific user to a specific service provider for a specific service, etc. The identification number both serves as a time stamp (it is issued by the user and unique for each run) and connects the messages of one run.

Some general assumptions where made in [29]:

1. In case of dispute, the user is always right. In consequence, if the service provider tries to cheat and the user complains about the charge, the service provider will always loose. On the other hand, the service provider can protect himself against malicious users by refusing access to the system.
2. All agents have access to a verifiable source of public keys, none of the keys gets compromised.
3. To secure the protocol against replay attacks etc., "time stamps, nonces, and other well documented means" can be used.

A number of security requirements the protocol imposes are also explained in [29], some of which we list below.

1. The service provider can only be held responsible for those prices he indeed offered.
2. The user must be sure he will only be charged for services he received for the price mutually agreed on.
3. The service provider must be sure that the user is not able to receive a service and deny the receipt. Clearly, the protocol does allow the user to deny having ever received the service: he can just refuse to send the acknowledgement message. According to [29], the protocol is designed for services with small value only, thus a service provider can always deny a user that has refused to send acknowledgement messages further access to the system.

7.1 Our assumptions and formalization

To model this particular version of the IBS protocol, we used the following assumptions:

1. The system consists of two users *User1*, *User2*, two service providers $SP1$, $SP2$, the billing server BS and an outside intruder E. The billing server is the only agent that always behaves according to the protocol description. Furthermore, we have two services $service1$, $service2$ with respective service names and different prices to start with for the service providers. The price is incremented after having been used in order to model different prices being given to the users. (We can use, however, a initial state where both service providers use the same price as a starting point.)
2. All agents own valid key pairs and the public keys of all other agents.
3. All predicates contain, for each user (service provider) P , an additional component *xPmemory*. For each protocol run an agent is engaged in we have an entry in this component that contains the ID, the user / service provider he is communicating with, the service that is being negotiated, the price given and the state the agent is in. Equivalently, the billing server holds a list of authorization requests ($xauthrequ$), each of which contains ID, $price$, *User* and SP being engaged in the particular negotiation. Additionally, BS holds a list with the accounts of users and service providers.
4. Although in [29] it is not explicitly said so, we assume that all signatures are checked. Since the protocol does not provide information on who signed the messages, we added a further component *xdetmessages* to the predicates that includes the ID, the message

type (i.e. $message1$, $message2$, etc.) and the signers of the message, starting with the outermost signature, the next inner one, then the innermost signature (if there are that many). This allows to identify the signer of the first message (a service provider can not know who has sent a particular price request), and to identify which run the message belongs to by use of the ID.
Whenever in the recipient P's $xPmemory$ there exists already an entry with the ID given in $xdetmessages$, the public keys to be used for signature verification are determined by using the public keys of the user and service provider, respectively, given in this particular entry. This models the fact that a user will only accept a service provided by the service provider he ordered the service from, that a service provider will only accept an order that was sent by the user he offered the specific price to, etc. However, there may be situations where only the integrity of the message needs to be ensured and the actual signer is not of interest. For example, a user might not be interested in knowing who actually sent the mpeg file he ordered. To model this we can determine the public keys to be used for signature verification by using the agent names given in $xdetmessages$.

8 Our analysis of the IBS protocol

As already pointed out in the previous section, there are a number of security requirements to be met. In the following we will list a few of these requirements and the formula that models the respective property (thus the formula to be found by Otter for finding a contradiction and hence an attack).

1. The service provider can only be held responsible for those prices he indeed offered. This can be formalized with the formula $-(state(\ldots) \wedge give\_price(xSP, xID) > give\_price(xauthrequ, xID))$.
2. The user must be sure he will only be charged for services he received at the price mutually agreed on. These are actually two requirements that can be captured by $-(state(\ldots) \wedge -service\_delivery(xauthrequ, xevents))$ and $-(state(\ldots) \wedge give\_price(xUser, xID) > give\_price(xauthrequ, xID))$.
3. The service provider must be sure that the user is not able to receive a service and deny the receipt: $-(state(\ldots) \wedge denial\_ackn(xUsermemory, xSPmemory))$.

The values of the predicates $give\_price, denial\_ackn$, etc., are determined by means of conditional demodulation.

For the analysis, in general we assumed that all possible checks are performed. This includes that the service provider checks, receiving message 3, that the price given in his own signature (which is part of the message signed by the user) is the same as the one signed by the user; that the billing server checks that the two signatures of the service provider in the request for authorization (inner and outer) are performed by the same service provider, etc. This also includes that the billing server checks, when receiving an authorization request, that he has not yet received a request with the ID given.

The first protocol flaw we found is the one already stated in [29]: By modelling a malicious user that can stop the protocol run at any given point, we found (which is not surprising) a state in which the user has stopped the run after having received the service and the service provider has not received an acknowledgement message [13].

We then disabled the user's ability to stop runs and let the analysis run with a malicious user cooperating with a malicious service provider. Now Otter found a protocol run in which the user asks the service provider for a price (they may also agree on a price beforehand, we only modelled protocol runs that start with step 1), the service provider sends the price and the user then orders the service. The service provider then correctly asks the billing server for authorization, which is given. But now, instead of sending the service, this protocol run proceeds with the user sending the message that acknowledges having received the service (which in fact he did not) to the service provider. Finally the billing server accepts the log file containing the user's acknowledgement message. Thus the protocol does allow a situation where the user is charged without having received a service.

One may argue that this can not be considered an attack, since it is only possible with the active cooperation of the user, and thus is no violation of the security requirement that a user may only be charged for services actually received. In many cases this will be right. However, our analysis reveals the fact that the messages received by the billing server do not constitute a proof that a service was delivered but a proof that the user agrees on being charged a specific amount of money. There may be environments where it matters whether or not the billing server can be used for transferring money from one account to another without service delivery being involved, environments where any kind of "money laundery" has to be avoided.

Our analysis shows that care must be taken when specifying the security requirements. One way to formalize security requirements precisely is by way of system states that describe a violation of the desired property. In an environment where the above described protocol run is unacceptable, the formula $-(state(\ldots) \wedge -service\_delivery(xauthrequ, xevents))$ captures the demand that **no** user shall be charged without having received a service. If "money laundery" is of no importance, the requirement can be relaxed by adding $\wedge -in\_list(xuser, xbadguys)$.

9 Conclusions

In this paper we have presented our approach for the security analysis of cryptographic protocols. We used the theorem proving tool Otter, not in order to verify security properties of cryptographic protocols, but as a means to state space exploration. The learning curve for using Otter is quite long, which is a disadvantage. Furthermore, one can easily write Otter input which would simply take too long to find a protocol failure even if one exists and is in the search path of Otter's heuristic. Thus, using this kind of tool involves the usual decisions regarding the search path to be followed.

Nevertheless, our approach has shown itself to be a powerful and flexible way of analyzing protocols. By making all of the agents' actions explicit, our methods reveal implicit assumptions that are not met by the protocol being analyzed. In particular, our methods found protocol weaknesses not previously found by other formal methods. Furthermore, our method is not restricted to assuming perfect encryption. Properties of cryptographic algorithms (such as the homomorphic property of RSA) that may be misused by a dishonest agent can easily be modelled by including symbolic functions and adequate equations.

On the other hand, as we are concerned with protocol validation rather than verification, if our analysis does not find an attack, we can not conclude that the protocol is secure in general. All we know is that under certain assumptions concerning the security properties of the cryptoalgorithms used and the abilities of malicious agents, a certain state is not reached.

Furthermore, we have successfully adapted our methods to address the specific needs of e-commerce protocol analysis. The new model for the analysis of e-commerce protocols is flexible enough to be used for different threat scenarios with respect to possibilities of malicious agents. Our analysis of a specific version of the IBS protocol both shows our methods to be applicable to e-commerce protocols and emphasizes that care must be taken when identifying the security requirements of such a protocol. A way to formalize the desired properties is to use formulas that describe a system state where the property in question is violated. Future work will include the formalization of more security properties relevant for e-commerce protocols and the application of our methods to other e-commerce protocols.

References

1. G. Bella and L.C. Paulson. Kerberos version iv: Inductive analysis of the secrecy goals. In *5th European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, pages 361–375. Springer-Verlag, 1998.
2. M. Bellare, R. Canetti, and H. Krawczyk. A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols. In *Annual Symposium on the Theory of Computing*. ACM, 1998.
3. M. Bellare and P. Rogaway. Provably secure session key distribution - the three party case. In *Annual Symposium on the Theory of Computing*, pages 57–66. ACM, 1995.
4. R. Berger, S. Kannan, and R. Peralta. A framework for the study of cryptographic protocols. In *Advances in Cryptology – CRYPTO '85*, Lecture Notes in Computer Science, pages 87–103. Springer-Verlag, 1985.
5. C. Boyd. Hidden assumptions in cryptographic protocols. In *IEEE Proceedings*, volume 137, pages 433–436, 1990.
6. S. Brackin. Automatically detecting authentication limitations in commercial security protocols. In *Proc. of the 22nd National Conference on Information Systems Security*, October 1999.
7. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Report 39, Digital Systems Research Center, Palo Alto, California, Feb 1989.
8. J. Clark and J. Jacob. On the Security of Recent Protocols. *Information Processing Letters*, 56:151–155, 1995.
9. D. Denning and G. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24:533–536, 1982.
10. DIN NI-17. *Chipkarten mit Digitaler Signatur - Anwendung/Funktion nach SigG und SigV - Teil 1: Anwendungsschnittstelle*, April 2000.
11. DIN NI-17.4. *Spezifikation der Schnittstelle zu Chipkarten mit Digitaler Signatur-Anwendung / Funktion nach SigG und SigV, Version 1.0 (Draft)*, November 1998.
12. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
13. S. Gürgens and J. Lopez. Suitability of a classical analysis method for e-commerce protocols. In Yair Frankel George I. Davida, editor, *Information Security, 4th International Conference, ISC 2001*, volume 2200 of *lncs*, pages 46–62. Springer Verlag, 2001.
14. S. Gürgens, J. Lopez, and R. Peralta. Efficient Detection of Failure Modes in Electronic Commerce Protocols. In *DEXA '99 10th International Workshop on Database and Expert Systems Applications*, pages 850–857. IEEE Computer Society, 1999.
15. S. Gürgens and R. Peralta. Efficient Automated Testing of Cryptographic Protocols. report 45, GMD German National Research Center for Information Technology, Darmstadt, Germany, December 1998.
16. S. Gürgens and R. Peralta. Validation of Cryptographic Protocols by Efficient Automated Testing. In *FLAIRS2000*, pages 7–12. AAAI Press, May 2000.
17. N. Heintze and J.D. Tygar. A Model for Secure Protocols and their Compositions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–13. IEEE Computer Society Press, May 1994.
18. ISO/IEC. *ISO/IEC CD 7816-8.2: "Identification cards - Integrated circuit(s) cards with contacts - Part 8: Security related interindustry commands"*, June 1997.
19. R. Kailar. Accountability in Electronic Commerce Protocols. *IEEE Transactions on Software Engineering*, 22(5):313–328, 1996.
20. J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). Network Working Group, Request for Comments 1510, September 1993.

21. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Second International Workshop, TACAS '96*, volume 1055 of *LNCS*, pages 147–166. SV, 1996.
22. W. Marrero, E. M. Clarke, and S. Jha. A Model Checker for Authentication Protocols. In *DIMACS Workshop on Cryptographic Protocol Design and Verification*, http://dimacs.rutgers.edu/Workshops/Security/, 1997.
23. C. Meadows. A system for the specification and verification of key management protocols. In *IEEE Symposium on Security and Privacy*, pages 182–195. IEEE Computer Society Press, New York, 1991.
24. C. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology - Asiacrypt '94*, volume 917 of *LNCS*, pages 133 – 150. SV, 1995.
25. C. Meadows. Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of Two Approaches. In *Proceedings of ESORICS*, Naval Research Laboratory, 1996. Springer.
26. C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In *Proceedings of Financial Cryptography*, 1998.
27. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, pages 993–999, 1978.
28. R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of PODC*, pages 51–59, 1991.
29. K. O'Toole. The Internet Billing Server - Transaction Protocol Alternatives. Technical Report INI TR 1994-1, Carnegie Mellon University, Information Networking Institute, 1994.
30. S. Pancho. Paradigm shifts in protocol analysis. In *New Security Paradigms Workshop*, 1999.
31. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
32. L. C. Paulson. Inductive Analysis of the Internet Protocol TLS. *ACM Trans. on Information and System Security*, 2(3):332–351, 1999.
33. R. L. Rivest, A. Shamir, and L. A. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
34. C. Rudolph. *A Model for Secure Protocols and its Application to Systematic Design of Cryptographic Protocols.* PhD thesis, Queensland University of Technology, 2001.
35. S. Schneider. Verifying authentication protocols with CSP. In *IEEE Computer Security Foundations Workshop*. IEEE, 1997.
36. S. Schneider. Formal Analysis of a non-repudiation Protocol. In *IEEE Computer Security Foundations Workshop*. IEEE, 1998.
37. V. Shoup and A. Rubin. Session key distribution using smart card. In *Advances in Cryptology - EUROCRYPT '96*, volume 1070 of *LNCS*, pages 321–331. SV, 1996.
38. G. J. Simmons. Proof of Soundness (Integrity) of Cryptographic Protocols. *Journal of Cryptology*, 7(2):69–77, 1994.
39. P. Syverson. A Different Look at Secure Distributed Computation. In *10th Computer Security Foundations Workshop*, pages 109–115. IEEE, 1997.
40. P. Syverson, C. Meadows, and I. Cervesato. Dolev-Yao is no better than Machiavelli. In *Proceedings of WITS 2000, Workshop on Issues in the Theory of Security*, pages 87–92, 2000.
41. M. Tatebayashi, N. Matsuzaki, and D. Newman. Key Distribution Protocol for Digital Mobile Communication Systems. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89*, volume 435 of *LNCS*, pages 324–333. SV, 1991.
42. L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning - Introduction and Applications.* McGraw-Hill, Inc., 1992.

Appendix

In the following we show the formalization of the agents' actions in the Needham-Schroeder protocol for symmetric algorithms. In these formulas, we use the Otter function $NEXT_CL_NUM to generate fresh numbers (the function returns the number of the next clause generated by Otter).

All formulas are of the form $(p_1 \wedge \ldots \wedge p_i) \longrightarrow f_1 = f_2$ which is interpreted by Otter in the following way: If predicates $p_1, \ldots, p_i$ hold on formula f_1, then f_1 is replaced (demodulated) by f_2. This models an agent that accepts the message being part of f_1 after having performed the checks modelled by the predicates, and then sends the message of f_2. Note that some of the predicates $p_1, \ldots, p_i$ model the agent's checks, while others are used to direct the search.

```
% Agent's protocol start:

(
  $ID(xastep,init)
     % xa is in state init
)
->
state(
xevents,xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,xbelief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                    xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                    xbbelief2,
xrunnumber,no_send,no_receive,no_message,
                                 xscount,xdepth)
=
state(
[[xa,S,xrunnumber+1,message1,
[xa,xb,$NEXT_CL_NUM]] | xevents],xeknows,
xa,xrunnumber+1,expects_message2,$NEXT_CL_NUM,
                                    no_key,xb,
xb,xbrun,init,no_nonce,no_key,no_agent,
                                    no_belief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
```

```
                                    xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                    xbbelief2,
xrunnumber+1,xa,S,[xa,xb,$NEXT_CL_NUM],
                              xscount,xdepth+1).


% S's program step2 (receive message 1,
%                    send message 2)

(
  is_agent(x1)  % 1. message element is
                % agent's name
  &
  is_agent(x2)  % 2. message element is
                % agent's name
)
->
send_E(
xevents,xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,xbelief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                    xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                    xbbelief2,
xrunnumber,E,S,[x1,x2,x3],xscount,xdepth)
=
state(
[[S,x1,elem(3,elem(1,xevents)),message2,
[enc(key(x1,S),[x3,x2,$NEXT_CL_NUM,
                                 enc(key(x2,S),
[$NEXT_CL_NUM,x1])])]] | xevents],xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,xbelief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                    xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                    xbbelief2,
xrunnumber,S,x1,[enc(key(x1,S),[x3,x2,
                                  $NEXT_CL_NUM,
enc(key(x2,S),[$NEXT_CL_NUM,x1])])],
                              xscount,xdepth+1).

% Agent's program step3 (receive message 2,
%                        send message 3)

(
  $ID(give_message_type(xevents),message2)
     % message shall be interpreted as message2
  &
  $ID(elem(1,dec(key(xa,S),xmessage)),xanonce)
     % After decryption with key(xa,S), the
     % first element of result is xa's nonce
  &
  $ID(elem(2,dec(key(xa,S),xmessage)),xaagent)
     % After decryption with key(xa,S), the
     % second element of result is xa's desired
     % communication partner
  &
  $ID(xastep,expects_message2)
     % xa expects message 2
)
->
send_E(
xevents,xeknows,xa,xarun,xastep,xanonce,no_key,
                                       xaagent,
 xb,xbrun,xbstep,xbnonce,xbkey,xbagent,xbelief,
 xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                    xabelief2,
 xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                    xbbelief2,
 xrunnumber,E,xa,[xmessage],xscount,xdepth)
=
state(
[[xa,xaagent,xarun,message3,
[elem(4,dec(key(xa,S),xmessage))]] | xevents],
                                     xeknows,
xa,xarun,expects_message4,xanonce,
elem(3,dec(key(xa,S),xmessage)),xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,a_believes,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                    xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                    xbbelief2,
xrunnumber,xa,xaagent,
[elem(4,dec(key(xa,S),xmessage))],xscount,
                                    xdepth+1).


% Agent's program
step 4 (receive message 3,
%                          send message 4)

(
%  $ID(length(dec(key(xb,S),xmessage)),2)
       %checking this avoids arity attack
%  &
  $ID(xbstep,init)
     % xb is waiting for message 3
  &
  $ID(give_message_type(xevents),message3)
     % message shall be interpreted as
     % message 3
  &
  different(elem(2,dec(key(xb,S),xmessage)),xb)
     % element 2 is not agent's name
)
->
send_E(
xevents,xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,xbelief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
```

```
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber,E,xb,[xmessage],xscount,xdepth)
=
state(
[[xb,elem(2,dec(key(xb,S),xmessage)),
                          xrunnumber+1,message4,
[enc(elem(1,dec(key(xb,S),xmessage)),
[$NEXT_CL_NUM])]]| xevents],xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xrunnumber+1,expects_message5,$NEXT_CL_NUM,
elem(1,dec(key(xb,S),xmessage)),
elem(2,dec(key(xb,S),xmessage)),xbelief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber+1,xb,elem(2,dec(key(xb,S),xmessage)),
[enc(elem(1,dec(key(xb,S),xmessage)),
[$NEXT_CL_NUM])],xscount,xdepth+1).

% Agent's program step 5a (receive message 4,
%                          send message 5)

(
  $ID(xastep,expects_message4)
    % xa expects message 4
  &
  $ID(give_message_type(xevents),message4)
    % message shall be interpreted as message 4
)
->
send_E
(xevents,xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,xbelief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber,E,xa,[xmessage],xscount,xdepth)
=
state(
[[xa,xaagent,xarun,message5,
[enc(xakey,[elem(1,dec(xakey,xmessage))-1])]]
| xevents],xeknows,
xa,xarun,a_finished,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,a_believes,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber,xa,xaagent,
[enc(xakey,[elem(1,dec(xakey,xmessage))-1])],
xscount,xdepth+1).

% Agent's program step 5b (receive message 5)

(
  $ID(elem(1,dec(xbkey,xmessage)),xbnonce-1)
     % After decryption with session key, result
     % is xb's nonce minus 1
  &
  $ID(xbstep,expects_message5)
     % xb expects message 5
  &
  $ID(give_message_type(xevents),message5)
     % message shall be interpreted as message 5
)
->
send_E(
xevents,xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,a_believes,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber,E,xb,[xmessage],xscount,xdepth)
=
state(
[[both_believe]|xevents],xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,b_finished,xbnonce,xbkey,xbagent,
                                     both_believe,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber,no_send,no_receive,no_message,xscount,
                                        xdepth+1).


(
  $ID(elem(1,dec(xbkey,xmessage)),xbnonce-1)
  &
  $ID(xbstep,expects_message5)
  &
  $ID(give_message_type(xevents),message5)
)
->
send_E(
xevents,xeknows,
xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,xbstep,xbnonce,xbkey,xbagent,no_belief,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                        xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                        xbbelief2,
xrunnumber,xwho,xb,[xmessage],xscount,xdepth)
=
state(
```

```
[[b_believes]|xevents],xeknows,
Xa,xarun,xastep,xanonce,xakey,xaagent,
xb,xbrun,b_finished,xbnonce,xbkey,xbagent,
                                    b_believes,
xa2,xarun2,xastep2,xanonce2,xakey2,xaagent2,
                                     xabelief2,
xb2,xbrun2,xbstep2,xbnonce2,xbkey2,xbagent2,
                                     xbbelief2,
xrunnumber,no_send,no_receive,no_message,xscount,
                                      xdepth+1)
```