Pitfalls in formal reasoning about security protocols

Nina Moebius, Kurt Stenzel, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Moebius, Nina, Kurt Stenzel, and Wolfgang Reif. 2010. "Pitfalls in formal reasoning about security protocols." In 2010 International Conference on Availability, Reliability and Security, 15-18 February 2010, Krakow, Poland, edited by Makoto Takizawa, A Min Tjoa, Markus Aleksy, Solange Ghernouti-Hélie, Gerald Quirchmayr, and Edgar Weippl, 248–53. Piscataway, NJ: IEEE. https://doi.org/10.1109/ares.2010.36.

Pitfalls in Formal Reasoning about Security Protocols

Nina Moebius, Kurt Stenzel, Wolfgang Reif
Institute for Software and Systems Engineering
University of Augsburg, Germany
E-Mail: {moebius, stenzel, reif}@informatik.uni-augsburg.de

Abstract

Formal verification can give more confidence in the security of cryptographic protocols. Application specific security properties like "The service provider does not loose money" *can give even more confidence than standard properties like secrecy or authentication. However, it is surprisingly easy to get a meaningful property slightly wrong. The result is that an insecure protocol can be 'proven' secure. We illustrate the problem with a very small application, a copy card, that has only five different messages. The example is taken from a paper where the protocol is secure, but the proved property slightly wrong. We propose to solve the problem by incorporating more of the real-world application into the formal model.*

Keywords: formal verification, security protocols

1 Introduction

There is a large body of work concerning the formal analysis of cryptographic (a.k.a. security) protocols (see [10] for a recent survey). Most approaches deal with standard security properties like secrecy, authentication, integrity, etc. This is especially true for approaches based on fully automatic techniques (model checking, etc., e.g. [11, 1, 6]). For generic protocols (e.g. authentication protocols like the famous Needham-Schroeder protocol [7]) these properties are sufficient – they have no other properties. However, for dedicated protocols that are used as part of a specific application one would like to have more assurance. The formal verification of *application specific* security properties may provide this assurance. Especially e-commerce applications are concerned with the exchange of electronic money, tickets, or other electronic goods. Here the standard properties are used to ensure other goals, e.g. "it is not possible to cheat", or more specific goals like "a ticket on a smart card is genuine, and cannot be copied". However, there is surprisingly few work on this topic. Jürjens [5] models and verifies on paper properties about CEPS, a *Common Electronic Purse Specification*[1]. Bella, Massacci, and Paulson [2] have done extensive work on the very complex SET, the *Secure Electronic Transaction Specification*[1], and proved many properties, e.g. that the bank learns nothing about the goods purchased by the customer. Their proofs were done with Isabelle, using Paulsons inductive approach [9].

The formal verification of application specific security properties has the potential to give high assurance about the security of the application. As it turns out there is a drawback: It is quite easy to inadvertently formulate and prove properties that look good, but actually give no assurance. In this paper we demonstrate that this problem exists, and propose a solution. The current work is part of the *SecureMDD* approach [8], a model-driven development method for secure smart card applications. We take up an earlier work by our group [4] about the verification of a very simple application, a copy card. Actually, the protocol presented in [4] is secure, but this is not guaranteed by the proven property.

In this paper we will present a simple protocol for a copy card (section 2), and describe a security property that holds for the application (section 3). Then we will show an attack on the application (section 4), and describe the proplem and propose a solution to improve the value of application specific security properties (section 5). Section 6 concludes.

2 A Simple Application: Copy Card

We demonstrate the pitfalls when formally reasoning about security protocols with a very simple smart card application, a copy card. The protocols are so small that we can present and discuss them in detail. The idea is to have a smart card that stores *points* that can be used to pay at copy machines in a university, a copy shop, or a library. Points can be loaded at dedicated vending machines by inserting money. The copy card application can be seen as a simple form of electronic cash.

In a scenario like this the application provider (e.g. the

[1] The official website with the specification no longer exists.

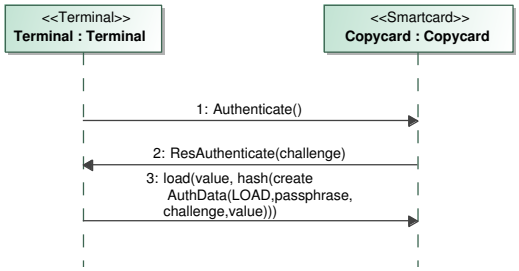

Figure 1. The protocol for loading points.

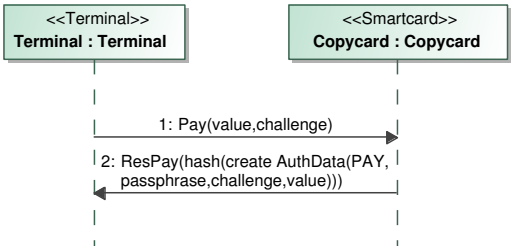

Figure 2. The protocol for paying points.

university that puts up the copying and vending machines and issues the smart cards) does not want to be cheated. On the other hand, there are usually no guarantees for the user of the application. If the smart card is lost or stolen the points and the money is lost. So we will focus only on the security of the application provider.

Usually the owner of a smart card is considered to be a possible cheater. Obviously it would be nice to use the copying machines without paying. This can happen in two ways in our scenario: The smart card owner can try to load points onto a genuine card with his own computer (and without paying for it). Or he can try to use a self-programmed, forged card at the copier. So we have two security requirements: Only a genuine vending machine is able to load points onto a genuine card, and only points from a genuine smart card are accepted at a copier. The load protocol is shown in fig. 1.

For the sake of simplicity we do not distinguish between a vending machine and a copier, but use the term *terminal* for both. The load protocol works as follows:

1. In a smart card scenario the terminal must always begin the communication by sending commands to the card, and the card only answers. So the terminal issues an *Authenticate* command to indicate that it wants to authenticate itself to the card.

2. The card generates a challenge (a nonce) and returns it to the terminal in a *Res(ponse)Authenticate* answer.

3. In the last step the terminal issues a *Load* command. It consists of two parts: the number of points to load (*value*) as clear text, and hash value that contains the tag *LOAD*, a pass phrase *passphrase*, the challenge from the previous message, and the *value* again.

The challenge avoids replay attacks. The pass phrase serves as a shared secret between all terminals and all smart cards, and is used to authenticate the terminal. The hash value ensures message integrity (it is not possible to modify the number of points to load without detection), and makes sure that the pass phrase cannot be recovered even if the message is monitored by an attacker.

The pay protocol (fig. 2) works as follows:

1. The terminal sends a *Pay* command with the number *value* of points to subtract, and a challenge (again a newly generated nonce) to the card.

2. The card answers with a *Res(ponse)Pay* message that contains only a hash value similar to the hash value used in the *Load* command. It consists of the *passphrase* (the shared secret), the *challenge* to avoid replay attacks, and the *value* to ensure that the correct number of points were subtracted. A *PAY* tag distinguishes a load hash from a pay hash.

In the rest of the paper we will focus on the load protocol because the pay protocol is not needed for the purpose of this paper.[2]

3 Formal Models for Security Protocols

Are the protocols secure from the point of view of the application provider? Can we formally verify the security of the copy card application? Actually, no.

The sequence diagrams contain not enough information. It is possible to design a secure copy card with these protocols, but unless more details about the checks and actions to perform are provided it is not possible to reason formally. For example, it must be made explicit that new challenges are generated, and that the hash values are checked.

Therefore we present a much more detailed description of the protocols with UML activity diagrams. The terminal, the smart card, and the messages are modeled as classes with attributes and associations as shown in fig. 3.

The smart card (class *Copycard*) has three attributes, the *balance* containing the currently loaded points, the *passphrase*, and the *challenge*. The types *Secret*, and *Nonce* are predefined types that contain a byte array holding the actual values. The *Load* message has an association with stereotype ≪*hashed*≫ to the class *AuthData* indicating that

[2]All diagrams and proofs can be found at `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/`

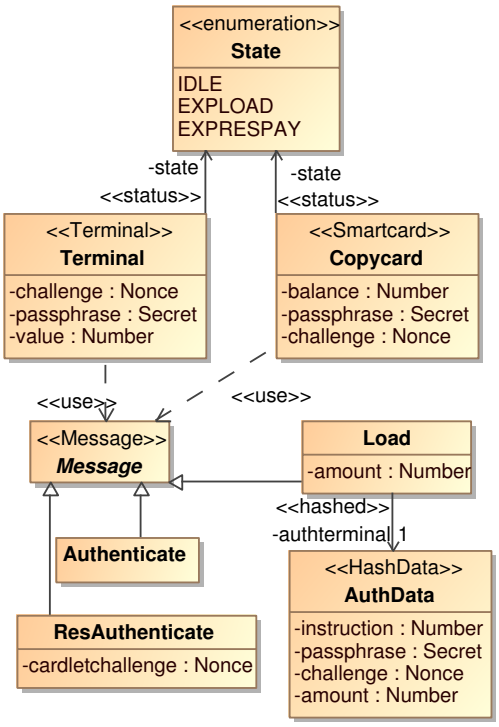

Figure 3. The class diagram describing the copy card application.

it is not sent as clear text but only hashed as in fig. 1. The message classes for the pay protocol are omitted. The activity diagram for the load protocol is shown in fig. 4.

A load protocol run starts in the upper left corner with the initial node. The terminal receives the instruction to run a load protocol with points *val*. The terminal checks that *val* is positive, stores *val* in its attribute *value*, and issues the *Authenticate* command to the card. On the card side the generation of a new challenge is modeled explicitely (`challenge := generateNonce()`). Furthermore, a state is used to ensure that the challenge field indeed contains a fresh nonce (`state := EXPLOAD`). The terminal uses the challenge, the shared secret, the value to load, and a tag to create a fresh authentic *load* message. The card state is checked when this *Load* command is received. Without this check a replay attack is possible by monitoring the *Load* command and sending it a second time to the card without another *Authenticate* command. After checking the hash value and checking for overflows the received points are added to the *balance* attribute of the card.

The activity diagrams and the class diagram are precise enough to generate a formal Abstract State Machine (ASM) specification. Together with a formal attacker specification we obtain a formal model suitable for verification. We

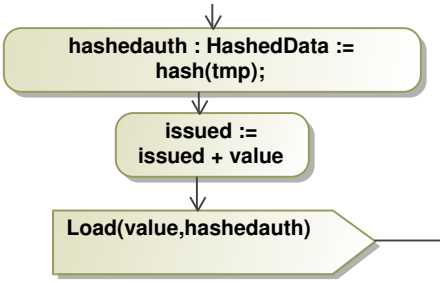

Figure 5. Modified load protocol with *issued*

assume a Dolev-Yao attacker [3], i.e. an attacker that can monitor, manipulate, and suppress all messages, but cannot break the elementary cryptographic operations. For example, the attacker can send the same load message twice, but cannot retrieve the passphrase from the hash value, and cannot construct a correct hash value for a new nonce, because this is assumed to be computationally infeasible for a strong cryptographic hash operation.

The last step is to describe the security of the application formally. While statements like "the user is not able to cheat" sound very intuitively they are again not precise enough: What does "cheating" mean in this context? And is this all? However, for e-commerce applications and particularly for applications using some kind of electronic cash it is usually easy to express what the application provider wants: not to lose money. For the example this means:

> *The sum of all issued points at all vending machines is not smaller than the sum of all collected points at all copiers (at any point in time).*

This includes the security requirements that a user cannot load points onto a genuine card at his own computer, and that a forged card is not accepted at a copier.

The simplest way to express this property is by introducing an *issued* and a *collected* attribute in the terminal that count the issued and collected points, respectively. Then the property is an invariant that can be expressed in OCL as

```
not Terminal.allInstances().issued->sum()
  < Terminal.allInstances().collected->sum()
```

Fig. 5 shows part of the modified *load* protocol. Before sending the actual *load* command the issued points are increased. Similarly we increase the *collected* attribute in the *pay* protocol if the hash value is correct.

These attributes are not necessary for the protocol itself. They are comparable to ghost fields used in the verification of object oriented programs. In state-less, trace-based approaches the security property must be expressed by counting the issued and collected points occurring in the messages of the trace. There is, however, the problem that the

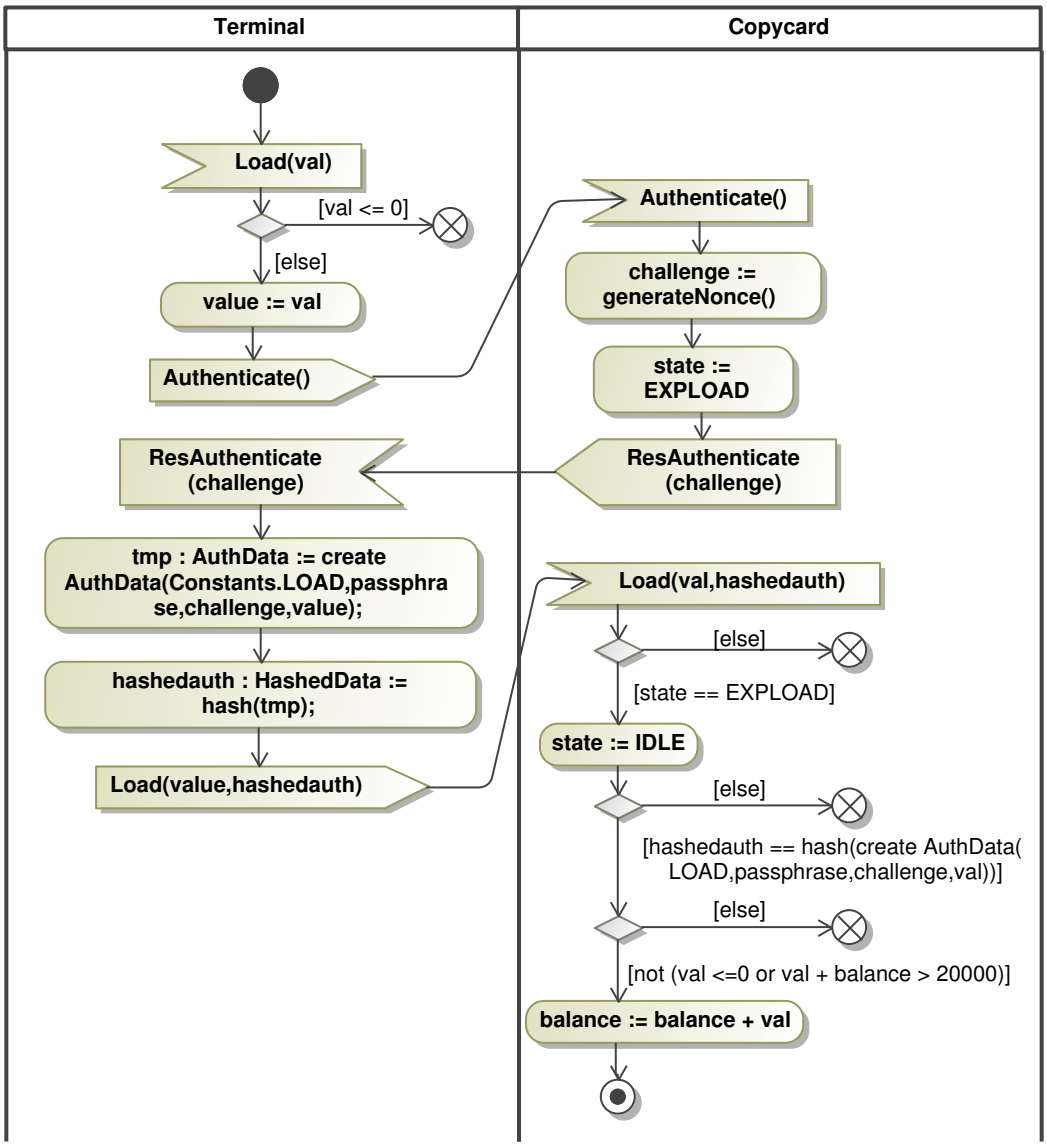

Figure 4. The detailed protocol for loading points.

origin of the message is important (messages from the attacker must not be counted).

Now it is possible to prove the security property for the presented protocol with the KIV[3] theorem prover. The property is an invariant that must be generalized for the proof to succeed. In a simplified form the invariant is

$$\sum \text{issued} \geq \sum \text{collected} + \sum \text{balance} + \sum \textit{value of authentic Load messages in transit} + \sum \textit{value of authentic ResPay messages in transit}$$

$\sum$ iterates over all terminals and cards, resp. The fact that there may be messages in transit is due to the formal model.

[3] http://www.informatik.uni-augsburg.de/swt/kiv

4 An Attack on the Copy Card

In the previous sections we described a security property that is intended to show the security of the protocol for the application provider. However, the protocol has a serious flaw. The attack works as follows (see the load protocol in fig. 4):

1. The attacker approaches a vending machine with a genuine smart card and some additional hardware: a laptop and some wires glued to the contacts of the smart card so that he can intercept and inject messages into the communication between the terminal and the smart card. This is the ability of a Dolev-Yao attacker.

2. The attacker inserts money and loads points (e.g. 10) on the card, and leaves the card in the terminal.

3. The attacker uses the laptop to send an *Authenticate* message to the card. The card answers with a new *ResAuthenticate* message which is passed to the terminal.

4. On reception of the *ResAuthenticate* message the terminal continues as specified in the load protocol: it creates a new *Load* message with the new challenge and the value from the previous protocol run, and sends it to the card.

5. The card accepts the value because the nonce is correct and increases its balance.

In effect the attacker was able to load 20 points onto a genuine card, but paid only for 10.

One can argue that this attack would never work in reality because the terminal would never accept a *ResAuthenticate* message after issuing a *Load* message. Or: in reality the terminal would reset the *value* attribute to zero. However, we are concerned with a formal model of a protocol, and the model allows the attack. There is nothing in the model that enforces a particular order of messages. Indeed, the idea of creating a protocol that is secure against a Dolev-Yao attacker is that it is not necessary to make (implicit) assumptions about how communication works. The order of the messages in the sequence diagram (fig. 1) shows only the intended communication in the absense of an attacker. Furthermore, the problem is not caused by the state-based modeling approach. In a state-less, trace-based approach the same error can occur because it must be specified when issued points are counted and when not.

5 Lessons Learned

As mentioned at the end of section 3 the security property (issued $\geq$ collected) holds for the protocol. This means that the property is not correctly formalized. It holds because the *issued* attribute is increased just before the *load* message is issued. So the attacker is able to increase the *issued* points and the *balance* of a card by the same amount at the same time. For the card, everything is correct, and the attacker is not able to load points himself. However, he can trick the terminal into issuing more points than money was inserted into the vending machine. It may be noted, that there is no attack on the *collected* points.

Ok, so the protocol is faulty. There are many possibilities to fix the protocol and the formal model. And it is not very surprising that even a short protocol like the copy card contains a serious error. But that is not the point of this paper. In this paper we are concerned with application specific security properties. What is disturbing is the fact that an apparently meaningful formal verification does not reveal the error. And the problem is not due to the particular formal framework used.

The problem is that formal protocol verification focuses too much on the exchanged messages and neglects the environment (the application) in which the protocol is embedded. In the copy card application real money is involved, but one tends to care only about the "points". Therefore it is important to tie the security properties to real world events (i.e. inserting money into the vending machine and issuing copies), not just to messages. The central idea behind the security property (the application provider does not want to lose money) is still valid. If we neglect the protocol completely we begin to formalize it as:

> *The sum of (the value of) all issued copies is not larger than the sum of all the money collected.*

This formulation is similar to the one in section 3, but has a completely different notion about what is "issued" and "collected". Now we think like an accountant, which suggests to add an attribute (real) *money* to the terminal instead of *issued* and *collected*.

In reality the user has to insert money into a vending machine before the *load* protocol starts. This means we have to increase the *money* attribute at the beginning of the load protocol, but after the check that the value to load is positive. This mimicks reality: It is not possible to insert a negative amount of money into an automaton. Then the load protocol runs. For the pay protocol the situation is different: The user *requests* to pay (i.e. to copy), then the pay protocol runs, and only after the successful completion of the pay protocol the copy is made. This means the *money* attribute must be decreased after the last step in the pay protocol. The new security property is then that the sum of all *money* attributes is never negative:

```
not Terminal.allInstances()
.money->sum() < 0
```

The modeling framework should make these "real world" actions explicit. In our framework we explicitely model a user and the interaction with the user (see fig. 6). The protocol is still faulty, of course. But now a proof attempt fails.

One possiblity to fix the protocol is by introducing another state EXP(ecting)RESAUTH for the terminal in the load protocol that is set before the *Authenticate* command is issued (fig. 6) and checked when a *ResAuthenticate* message is received. Then the proof of the security property succeeds (though a slightly different generalization of the invariant is needed).[4]

[4]The full model of the protocol and the proofs can be found at `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD`.

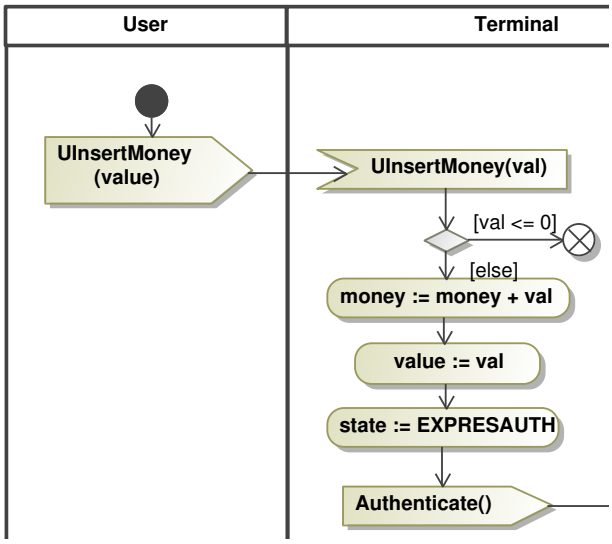

Figure 6. Corrected protocol with explicit user and *money* field.

Another view of the problem is that we use an internal state (the two attributes) as a representation of the external environment (inserting money and issuing copies). In principal there is nothing wrong with this if the values represent the environment faithfully. However, in our scenario the attacker is able to manipulate the values. If we can ensure that the values are *tamper-proof* it is not really important when the *issued* points are increased. So another, more generic solution is to generate proof obligations that guarantee that the attacker cannot manipulate certain values. This is future work. Still it is necessary to look at more than just the protocols.

6 Conclusion

The formal verification of cryptographic protocols has the potential to improve the quality of the protocols and avoid errors. In comparison to standard security properties like secrecy or authentication, application specific properties can give more assurance about the security of the whole application. This is especially true for e-commerce applications that typically involve the exchange of money, tickets, prescriptions etc. However, the example presented shows that it is quite easy to get an application specific security property wrong, even if it looks very convincing. The reason is – in our opinion – that the protocols are modeled too abstractly. Our proposed solution is therefore to include "real world" events, e.g. where real money is involved, into the formal models. This can further improve the quality and significance of formal models and proofs.

References

[1] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In *Proceedings of Esorics'03*, LNCS 2808, pages 253–270. Springer-Verlag, Heidelberg, 2003.

[2] Giampaolo Bella, Fabio Massacci, and Lawrence C. Paulson. Verifying the SET purchase protocols. *Journal of Automated Reasoning*, 36(1-2):5–37, 2006.

[3] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*. IEEE, 1981.

[4] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Smart Card Applications: An ASM Approach. In *International Conference on integrated Formal Methods (iFM) 2007*, volume 4591 of *LNCS*. Springer, 2007.

[5] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[6] Francis Klay and Laurent Vigneron. Automatic methods for analyzing non-repudiation protocols with an active intruder. In *FAST 2008 - 5th International Workshop on Formal Aspects in Security and Trust*. Springer LNCS 5491, 2009.

[7] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*, pages 147–166. Springer LNCS 1055, 1996.

[8] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *Third International Workshop on Secure Software Engineering, SecSE, at ARES 2009*. IEEE Press, 2009.

[9] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6, 1998.

[10] Juan Carlos Lopez Pimental and Raul Monroy. Formal support to security protocol development: A survey. *Computacion y Sistemas*, 12(1), 2008. available at `http://www.ejournal.unam.mx/cys/vol12-01/CYS012000105.pdf`.

[11] P. Y. A. Ryan, S. A. Schneider, M. H. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.