From: FLAIRS-00 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

Validation of Cryptographic Protocols by Efficient Automated Testing

Sigrid Gürgens
GMD - German National Research Center
for Information Technology
Rheinstrasse 75, 64295 Darmstadt, Germany
guergens@darmstadt.gmd.de

René Peralta *
René Peralta
Computer Science Department
Yale University
New Haven, CT. 06520-8285
peralta-rene@cs.yale.edu

Abstract

We present a method for validating cryptographic protocols. The method can find flaws which are inherent in the design of the protocol as well as flaws arising from the particular implementation of the protocol. The latter possibility arises from the fact that there is no universally accepted standard for describing either the cryptographic protocols themselves or their implementation. Thus, security can (and in practice does) depend on decisions made by the implementer, who may not have the necessary expertise. Our method relies on automatic theorem proving tools. Specifically, we used "Otter", an automatic theorem proving software developed at Argonne National Laboratories.

Keywords: Security analysis, authentication protocol, automatic theorem proving, protocol validation

Introduction

Cryptographic protocols play a key role in communication via open networks such as the Internet. These networks are insecure in the sense that any adversary with the necessary technical background can monitor and even modify the messages. Cryptographic protocols are used to provide confidentiality and authenticity of the communication. Messages are encrypted to ensure that only the intended recipient can understand them. Message authentication codes or digital signatures are used so that modifications can be detected. All of these communication "services" can be provided as long as the cryptographic protocols themselves are correct and secure. However, the literature provides many examples of protocols that were first considered to be secure and were later found to contain flaws. Hence validation of cryptographic protocols is an important issue.

In the last ten to fifteen years, active areas of research have focused on:

1. Developing design methodologies which yield cryptoprotocols for which security properties can be formally proven.
2. Formal specification and verification/validation of cryptoprotocols.

An early paper which addresses the first of these issues is (Berger, Kannan, & Peralta 1985), where 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 work done by people in the "cryptography camp", this research does not adopt the framework of formal language specification. More recent work along these lines includes that of Bellare and Rogaway (Bellare & Rogaway 1995), Shoup and Rubin (Shoup & Rubin 1996) (which extends the model of Bellare and Rogaway to smart card protocols), and Bellare, Canetti and Krawczyk (Bellare, Canetti, & Krawczyk 1998).

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 (Meadows 1995), (Marrero, Clarke, & Jha 1997) and (Paulson 1998). Most of the work in this field can be categorized as either development of logics for reasoning about security (so-called authentication logics) or as development of model checking tools. Both techniques aim at verifying security properties of formally specified cryptoprotocols.

A seminal paper on logics of authentication is (Burrows, Abadi, & Needham 1989). 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.

Model checking involves the definition of a state space (typically modeling 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 (Dolev & Yao 1983).

Much has already been accomplished. A well-known

*Supported in part by NSF Grant CCR-9207204 and by the German National Research Center for Information Technology.
Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

software tool is the NRL Protocol Analyzer (Meadows 1991).[1] Other promising work includes Lowe's use of the Failures Divergences Refinement Checker in CSP (Lowe 1996); Schneider's use of CSP (Schneider 1997); the model checking algorithms of Marrero, Clarke, and Jha (Marrero, Clarke, & Jha 1997); Paulson's use of induction and the theorem prover Isabelle (Paulson 1998); and the work of Heintze and Tygar considering compositions of cryptoprotocols (Heintze & Tygar 1994). As far as we know, the model checking approach has been used almost exclusively for verification of cryptoprotocols. 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 induction or other logical tools to reason about infinitely many states. Both methods have produced valuable insight into ways in which cryptoprotocols can fail. The problem, however, is not only complex but also evolving. The early work centered around proving security of key-distribution protocols. Currently, it is fair to say that other, much more complex protocols, are being deployed in E-commerce applications without the benefit of comprehensive validation.

In this paper we present a ***validation without verification*** approach. That is, we do not attempt to formally prove that a protocol is secure. Rather, our tools are designed to find flaws in cryptoprotocols. We reason that by not attempting the very difficult task of proving security and correctness, we can produce an automated tool that is good at finding flaws when they exist. We use the theorem prover Otter (see (Wos *et al.* 1992)) as our search engine. Some of the results we have obtained to date are presented in the following sections.

Cryptographic protocols

In what follows, we restrict ourselves to the study of key-distribution protocols. 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. In general, the secrecy of cryptographic keys cannot be assumed to last forever. Pairs of agents must periodically replace their keys with new ones. Thus the aim of a key-distribution protocol is to produce and securely distribute what are called "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).

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.

[1]The work in (Meadows & Syverson 1998) expands NRL to include some specification capabilities. The resulting tool was powerful enough to analyze the SET protocol used by Mastercard and Visa (Mastercard and VISA 1996).

To illustrate the need for security analysis of such protocols we describe a protocol introduced in (Needham & Schroeder 1978) and an attack on this protocol first shown in (Denning & Sacco 1982). In what follows, a message is an ordered tuple $(m_1, m_2, \ldots, m_r)$ of concatenated message blocks m_i, viewed at an abstract level where the agents can distinguish between different blocks. The notation $\{m\}_K$ denotes the encryption of the message m using key K. The notation $n.\ A \longrightarrow B :\ m$ denotes that in step n of a protocol execution (which we will call a protocol run throughout this paper) A sends message m to B. The protocol assumes that symmetric encryption and decryption functions are used (where encryption and decryption are performed with the same key).

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

In the first message agent A sends to S its name A, the name of the desired communication partner (B) and a random number R_A which it generates for this protocol run.[2] 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 is indeed a new key. According to the protocol description in (Needham & Schroeder 1978) 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.

[2]In this paper, "random numbers" are abstract objects with the property that they cannot be generated twice and they 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.

However, there is a well-known attack on this protocol, first shown in (Denning & Sacco 1982), which 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' keys holds only for a limited time period, 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. 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 can be due to the underspecification of a cryptoprotocol. 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.

The communication and attack model

The security analysis of protocols does not deal with weaknesses of the encryption and decryption functions used. In what follows we assume that the following security properties hold:

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

While the first two items describe generally accepted properties of encryption functions, the last one which Boyd called the "cohesive property" in (Boyd 1990) does not hold in general. Boyd and also Clark and Jacob (see (Clark & Jacob 1995)) 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.

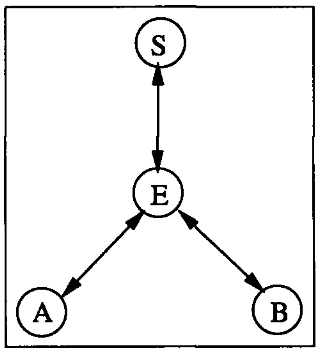

Figure 1: The communication model.

All communication is assumed to occur over insecure communication channels. What an insecure communication channel is can be defined in several ways. In our model we assume that there is a third agent E with the ability to intercept messages. 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. Furthermore, E is considered a valid agent, thus also shares a symmetric key K_{ES} with S and/or owns a key pair (K_E, K_E^{-1}) for an asymmetric algorithm, and is allowed to initiate communication either with other agents or S. Figure 1 depicts our communication model.

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 knows the names of the agents.
2. E knows the keys it owns.
3. E knows every message it has received.
4. E knows every part of a message received (where a ciphertext is considered one message part).
5. E knows everything it can generate by enciphering data it knows using data it knows as a key.
6. E knows every plaintext it can generate by deciphering a ciphertext it knows, provided it knows the inverse of the key used as well.
7. E knows every concatenation of data it knows.
8. E knows the origin and destination of every message.
9. 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, 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, we do not allow E to know ciphertexts of level 3 (e.g. $\{\{\{m\}_K\}_K\}_K$). We do this on an ad-hoc basis. This problem needs further exploration. [3]

Note 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.

Automatic testing 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.

Otter is based on first order logic. Thus for describing the protocol to be analyzed, the environment, and the possible state transitions, we use first order functions, predicates, implications and the usual connectives. At any step of a protocol run, predicates like

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

are used to describe the state of the protocol run given, the list *xevents* of actions already performed and the list *xEknows* of data known by E. These predicates are also used to describe the states of the agents (e.g. data like session keys, nonces, etc. the agents have stored so far for later use, the protocol runs they assume to be taking part in, and the next step they will perform in each of these runs). In general the predicates can include as many agents' states and protocol runs as desired, but so far we have restricted this information to agents A, B, E and S 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 message 1 of the protocol described in the second section can be formalized as follows:

$$\begin{aligned}
& sees(xevents, xEknows, \ldots, xwho, xto, xmess, \ldots) \\
& \land\ xto = S \\
& \land\ is\_agent(el(1, xmess)) \\
& \land\ is\_agent(el(2, xmess)) \\
& \rightarrow \\
& send(update(xevents), xEknows, \ldots, S, el(1, xmess), \\
& [enc(key(el(1, xmess), S), [el(3, xmess), \\
& el(2, xmess), new\_key, enc(key(el(2, xmess), S), \\
& [new\_key, el(1, xmess)])])], \ldots)
\end{aligned}$$

where *update(xevents)* includes the current send action and *el(k, xmess)* returns the kth block of the message *xmess* just received.

Whenever E receives a message (indicated by a formula $sees\_E(\ldots)$), an internal procedure adds all that E can learn from the message to the knowledge list *xEknows*. Implications $sees\_E(\ldots) \rightarrow send\_E(\ldots, message1, \ldots) \land send\_E(\ldots, message2, \ldots) \land \ldots$ formalize that after having received a particular message, E has several possibilities to proceed with the protocol run, i.e. each of the predicates $send\_E(\ldots, message, \ldots)$ 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.

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

$$dec(x, enc(x, y)) = y$$

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, the initial knowledge of the enemy E, etc.) and a formula describing a security goal. An example of such a formula is "if agents A and B share a session key, then this key is not known by E".

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.

Otter has a number of facilities for directing the search, for a detailed description see (Wos *et al.* 1992).

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

[3] We note, however, 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.

obvious checks. For example, consider step 3 of the Needham-Schroeder protocol explained in the second section.

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.

To our great surprise, 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 as follows: 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. $S \longrightarrow B: \ \{R_E, A, K_{AB}, \{K_{AB}, B\}_{K_{AS}}\}_{K_{BS}}$

This is passed on by E, and B interprets it as being the third message of a new protocol run. Since this message passes the checks described above, B takes R_E to be a new session key to be shared with A and the last two prototol 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 (Gürgens & Peralta 1999) 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 Denning-Sacco attack.

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 (Gürgens & Peralta 1999) for a detailed description of the protocol and its analysis) and on the Yahalom protocol, as described in (Burrows, Abadi, & Needham 1989).

Furthermore, we analyzed an authentication protocol for smartcards with a digital signature application being standardized by the German standardization organisation DIN ((DIN NI-17.4 1998)) and found the proposed standard to be subject to some interpretations which would make it insecure (see (Gürgens, Lopez, & Peralta 1999)).

Among the known attacks which were also found by our methods are the Lowe ((Lowe 1996)) and the Meadows (Meadows 1996) attacks on the Needham-Schroeder public key protocol and the Simmons attack on the TMN protocol (see (Tatebayashi, Matsuzaki, & Newman 1991)). The latter is noteworthy since this attack uses the homomorphism property of some asymmetric algorithms.

Conclusions

As is well known, there does not exist an algorithm which can infallibly decide whether a theorem is true or not. Thus the use of Otter, and per force of any theorem proving tool, is an art as well as a science. 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. In particular, our methods found protocol weaknesses not previously found by other formal methods.

We do not yet have a tool which can be used by applications programmers. Analyzing protocols with our methods requires some expertise in the use of Otter as well as some expertise in cryptology. Although it is clear that much of what we now do can be automated (e.g. a compiler could be written which maps protocol descriptions to Otter input); it is not clear how much of the search strategy can be automated.

Our work with Otter has led us to the conclusion that it is not sufficient to ask protocol designers to follow basic design principles in order to avoid security flaws. One reason for this is that those principles might not be appropriate for particular applications (e.g. security considerations in smart card applications are quite different from, say, security considerations in network applications). Adherence to some basic design principles, although a good idea, will not guarantee security in the real world. The new attack we found on the Needham-Schroeder protocol, for example, would not be prevented by using additional message fields to indicate the direction of a message. Our methodology can be useful for deciding what implementation features are necessary in a particular environment for a given protocol to be secure.

References

Bellare, M., and Rogaway, P. 1995. Provably secure session key distribution - the three party case. In *Annual Symposium on the Theory of Computing*, 57–66. ACM.

Bellare, M.; Canetti, R.; and Krawczyk, H. 1998. A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols. In *Annual Symposium on the Theory of Computing*. ACM.

Berger, R.; Kannan, S.; and Peralta, R. 1985. A framework for the study of cryptographic protocols. In *Advances in Cryptology - CRYPTO '95*, Lecture Notes in Computer Science, 87–103. Springer-Verlag.

Boyd. C. 1990. Hidden assumptions in cryptographic protocols. In *IEEE Proceedings*, volume 137, 433–436.

Burrows, M.; Abadi, M.; and Needham, R. 1989. A Logic of Authentication. Report 39, Digital Systems Research Center, Palo Alto, California.

Clark, J., and Jacob, J. 1995. On the Security of Recent Protocols. *Information Processing Letters* 56:151–155.

Denning, D., and Sacco, G. 1982. Timestamps in key distribution protocols. *Communications of the ACM* 24:533–536.

DIN NI-17.4. 1998. *Spezifikation der Schnittstelle zu Chipkarten mit Digitaler Signatur-Anwendung / Funktion nach SigG und SigV, Version 1.0 (Draft).*

Dolev, D., and Yao, A. 1983. On the security of public-key protocols. *IEEE Transactions on Information Theory* 29:198–208.

Gürgens, S., and Peralta, R. 1999. Efficient Automated Testing of Cryptographic Protocols. Technical Report 45-1998, GMD German National Research Center for Information Technology.

Gürgens, S.; Lopez, J.; and Peralta, R. 1999. Efficient Detection of Failure Modes in Electronic Commerce Protocols. In *DEXA '99 10th International Workshop on Database and Expert Systems Applications*, 850–857. IEEE Computer Society.

Heintze, N., and Tygar, J. 1994. A Model for Secure Protocols and their Compositions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy*, 2–13. IEEE Computer Society Press.

Lowe, G. 1996. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Second International Workshop, TACAS '96*, volume 1055 of *LNCS*, 147–166. SV.

Marrero, W.; Clarke, E. M.; and Jha, S. 1997. A Model Checker for Authentication Protocols. In *DIMACS Workshop on Cryptographic Protocol Design and Verification.*

1996. Mastercard and VISA Corporations, Secure Electronic Transactions (SET) Specification. http://www.setco.org.

Meadows, C., and Syverson, P. 1998. A formal specification of requirements for payment transactions in the SET protocol. In *Proceedings of Financial Cryptography.*

Meadows, C. 1991. A system for the specification and verification of key management protocols. In *IEEE Symposium on Security and Privacy*, 182–195. IEEE Computer Society Press, New York.

Meadows, C. 1995. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology - Asiacrypt '94*, volume 917 of *LNCS*, 133 – 150. SV.

Meadows, C. 1996. Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of Two Approaches. In *Proceedings of ESORICS.* Naval Research Laboratory: Springer.

Needham, R., and Schroeder, M. 1978. Using encryption for authentication in large networks of computers. *Communications of the ACM* 993–999.

Paulson, L. C. 1998. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security* 6:85–128.

Schneider, S. 1997. Verifying authentication protocols with CSP. In *IEEE Computer Security Foundations Workshop.* IEEE.

Shoup, V., and Rubin, A. 1996. Session key distribution using smart card. In *Advances in Cryptology - EUROCRYPT '96*, volume 1070 of *LNCS*, 321–331. SV.

Tatebayashi, M.; Matsuzaki, N.; and Newman. 1991. Key Distribution Protocol for Digital Mobile Communication Systems. In Brassard, G., ed., *Advances in Cryptology - CRYPTO '89*, volume 435 of *LNCS*, 324–333. SV.

Wos, L.; Overbeek, R.; Lusk, E.; and Boyle, J. 1992. *Automated Reasoning - Introduction and Applications.* McGraw-Hill, Inc.