Formal analysis of a TPM-based secrets distribution and storage scheme

Formal Analysis of a TPM-Based Secrets Distribution and Storage Scheme*

Ronald Toegl[1], Georg Hofferek[1], Karin Greimel[1], Adrian Leung[2], Raphael C-W. Phan[3]†
and Roderick Bloem[1]

[1]*IAIK, Graz University of Technology, Austria.*
{*firstname.lastname*}*@iaik.tugraz.at*
[2]*Royal Holloway, University of London, UK.*
a.leung@rhul.ac.uk
[3]*Loughborough University, UK.*
r.phan@lboro.ac.uk

Abstract

Trusted Computing introduces the Trusted Platform Module (TPM) as a root of trust on an otherwise untrusted computer. The TPM can be used to restrict the use of cryptographic keys to trusted states, i.e., to situations in which the computer runs trusted software. This allows for the distribution of intellectual property or secrets to a remote party with a reasonable security that such secrets will not be obtained by a malicious or compromised client. We model a specific protocol for the distribution of secrets proposed by Sevinç et al. A formal analysis using the NuSMV model checker shows that the protocol allows an intruder to give the client an arbitrary secret, without the client noticing. We propose an alternative that prevents this scenario.

Keywords: Trusted Computing, protocol analysis, model checking

1. Introduction

The current computing landscape is plagued by a variety of software-based attacks and threats such as viruses, phishing attacks, and trojan horses. It is becoming increasingly difficult to find suitable countermeasures to all these malicious activities. Trusted Computing promises to improve the security of today's computer systems. With the Trusted Platform Module (TPM), a hardware device is available that allows cryptographically qualified and tamper-proof statements on the software configuration of a machine. The full potential of such platforms becomes apparent on the Internet, as the TPM enables a remote party to decide on the trustworthiness of a host. Recently, the TPM has been integrated in a number of cryptographic network protocols to provide a whole suite of security functionalities [8, 15, 21].

However, the design of cryptographic protocols is a demanding task and indeed many vulnerabilities have been discovered in various protocols [1, 20] . With protocols integrating the TPM, it is tempting to rely on the hardware-based security features. Yet, similar problems as with conventional protocols may still occur, for instance the composition of an insecure protocol from secure components [7]. While it is still common to rely on best-practices and manual cryptanalysis to determine the absence of security risks, progress in the formal and partially automated analysis of security protocols has been made in the last years.

General purpose cryptographic protocols have been the subject of formal analysis for many years [17] using different tools such as Murφ [18] or ProVerif [4]. Basin et al. [3] propose the OFMC symbolic model checker and incorporate it in the AVISPA [2] tool. Zhang et al. [22] showed that it is feasible to model check the basic Needham-Schroeder public-key authentication protocol in SMV. Lomuscio [16] extends NuSMV to support ARCTL and models the dining cryptographers problem with it. In the field of Trusted Computing, Bruschi et al. [6] model the authentication session that is performed between the local application and the TPM module in the Spin model checker [13]. Lin [14] formally analysed parts of the TPM API.

This paper describes a formal analysis of a recent protocol for "securely distributing and storing secrets", "inde-

*This work was supported in part by the European Commission through projects COCONUT (FP7-2007-IST-1-217069) and Open-TC (027635).

†Work done while the author was with the Laboratoire de sécurité et de cryptographie (LASEC), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

978-0-7695-3398-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ICYCS.2008.456

pendent of a specific usage-control application" [19]. The protocol "ensures that the server only distributes given secret data to trusted clients." It does not, however, ensure that the secret received by the client does indeed come from the server. This property is not mentioned in [19], but we argue that it is important in some applications, such as when decisions are based on the contents of the secret. We show the absence of this property using a model checker and suggest an improvement of the protocol which does not suffer from the same drawback.

This paper is organised as follows: In Section 2 we describe Trusted Computing based on the TPM, introduce the protocol we analyse, and outline the fundamentals of Model Checking. Section 3 describes our model of the protocol and the attacker. Section 4 presents the results of the analysis and also suggests enhancements to improve the security of the scheme. Finally, we draw conclusions in Section 5.

2. Preliminaries

2.1. Trusted Computing

The Trusted Computing Group[1] has specified the Trusted Platform Module. Much like a smartcard, the TPM features cryptographic primitives, but it is physically bound to the main device. A tamper-resistant casing contains hardware primitives for public-key cryptography, key generation, cryptographic hashing, and random-number generation. With these components the TPM is able to enforce security policies on hierarchies of secret keys to protect them from any remote attacker.

The TPM implements high-level functionality such as reporting the current system state and providing evidence of the integrity of this measurement, known as *Remote Attestation*. This is done with the help of the Platform Configuration Registers (PCRs), that can only be written to with the *Extend Operation*. A PCR with index i in state t is *extended* with input x by setting

$$PCR_i^{t+1} = \text{SHA-1}(PCR_i^t || x).$$

Before executable code is invoked, the caller computes the code's hash value and extends a PCR with the result. In this way a *chain of trust* is built, starting from the BIOS, covering bootloader, kernel, system libraries, application code, etc. Ultimately, the exact configuration of the platform is mapped to PCR values. If such a system state fulfills the given security or policy requirements, we refer to it as a *trusted state*.

Upon request, the TPM signs the current values of the PCRs. It can also sign the *policy* for a given key pair, thus informing a third party that, for instance, a certain key will not be revealed outside the TPM. To protect the platform owner's privacy, the unique Endorsement Key of the TPM is not used for this signature. Rather, a pseudonym is used: an *Attestation Identity Key (AIK)*. The authenticity of an AIK can either be certified by a trusted third party, named PrivacyCA, or with the group-signature-based DAA scheme [5].

Another high-level feature of the TPM is that it can *bind* data (often a symmetric key) to a platform by encrypting it with a *non-migratable* key. Such a key never leaves the TPM's protected storage unencrypted. An extension to this is *Sealing*. A key may be sealed to a specific (trusted) value of the PCRs. A sealed key will not leave the TPM and is only used by the TPM if the PCRs hold the same values which were specified when the key was sealed. Thus, use of the key can be restricted to a trusted state of the computer.

2.2. Protocol

Sevinç et al. proposed a scheme to securely distribute a secret using trusted computing functionalities [19]. The setting is that the server does not trust the client platform but only the TPM residing at the client's end. It is possible for the server to ascertain with the help of the PCRs that the client platform is in a *trusted state*. Then, the server can expect the client platform to function in the desired manner, rather than (intentionally or unintentionally) running some malicious activity.

Protocol Flow. Table 1 shows the flow of the protocol. To avoid confusion, we stick to the numbering used in [19]. Additionally, we make some abstractions. For instance, we do not explain the local TPM functions in detail (steps 5–7 are left out). The interested reader can find a thorough description in [19].

1			C	$\rightarrow$	S	REQ
2			C	$\leftarrow$	S	V_{PCR}, N
3	TPM	$\leftarrow$	C			V_{PCR}
4	TPM					generate (K, K^{-1})
8	TPM	$\leftarrow$	C			N
9	TPM	$\rightarrow$	C			$Sig_{AIK}(V_{PCR}, N, K)$
10			C	$\rightarrow$	S	$Sig_{AIK}(V_{PCR}, N, K)$
11					S	check certificate
12			C	$\leftarrow$	S	$Enc_K(SECRET)$
13	TPM	$\leftarrow$	C			$Enc_K(SECRET)$
14	TPM					assert *trusted state*
15	TPM	$\rightarrow$	C			$SECRET$

Table 1. Key Distribution Protocol

There are three participants in the protocol, the client's TPM, the client, and the server. First the client sends a request (REQ) to the server. The server replies by sending values for PCRs (V_{PCR}) which define the *trusted state*, and

[1] http://www.trustedcomputinggroup.org

a nonce (N). In the third step, the client asks the TPM to create a key which is sealed to the required PCR values. Next, the client asks the TPM to certify the key. The TPM signs the key, the PCR values to which it is sealed, and the nonce N with an AIK. This certificate is sent to the server. Now the server verifies the certificate and asserts that the key K is non-migratable and sealed to the correct PCR values (step 11). Subsequently it encrypts the secret with K. The encrypted message is then sent to the client which asks the TPM to decrypt it. The TPM checks if the client is in the *trusted state*. If yes, the secret is revealed to the client.

Security Targets. We identify the following three security targets:

1. An intruder never learns the secret.
2. The client learns the secret only if it is in a *trusted state*.
3. A client in the trusted state either learns the (real) secret, or discovers that an attack has taken place.

Sevinç et al. state that the main goal of the protocol is to ensure the confidentiality of the secret, i.e., neither an intruder nor a client in an untrusted state may learn the secret. This corresponds to our first two security targets. The third target, not mentioned in [19], ensures that one of the following cases always occurs: (1) An honest client is capable of successfully completing the protocol and thus learns the secret, or (2) the client is able to detect that some malicious activity occurred, e.g. a signature is invalid. Below, we formalise these security targets and verify them using a model checker.

2.3. Model Checking

Model checking is used to formally prove or disprove that certain properties hold for a given model. For example, we prove that given security properties hold for a protocol. A model checker is a tool that takes a finite model and a specification (set of properties) as input and returns true if and only if the model satisfies the specification. If the model does not satisfy the specification the model checker returns false. Most model checkers are able to give a counterexample showing why the model does not satisfy the specification. Figure 1 illustrates the main idea of model checking and how it is applied in our setting.

In the following, we will shortly describe the models, the specification language, and the model checker we use.

Models. Our models are Finite State Machines (FSMs) or more formally Kripke structures. Let AP be a set of atomic propositions. A *Kripke structure* is a tuple $K = (S, T, S_0, A, L)$, where S is the finite set of states, $T \subseteq S \times S$ is the complete transition relation, $S_0 \subseteq S$ is the set of initial states, $A = 2^{AP}$ is the alphabet and $L : S \to A$ is the labelling function, which associates to every state the set of propositions that hold in that state. An infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ is a *path* of K if $\forall i.\ (s_i, s_{i+1}) \in T$. A path is a *run* of K if additionally, it starts with an initial state. The corresponding *word* $\sigma = L(s_0)L(s_1)L(s_2)\ldots$ is an infinite sequence of letters from the alphabet A, defined by the labelling function. In order to reason about properties of the set of words a Kripke structure defines we next introduce a specification language.

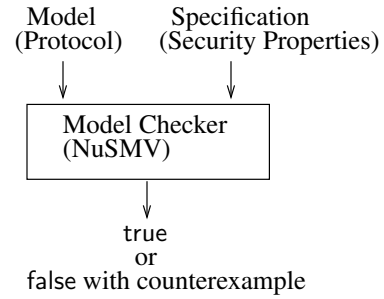

Figure 1. Model Checking

Specifications. The specification language we use is *Computation Tree Logic* (CTL) [10]. The logic is defined over a finite set of propositions, AP, the same set we used to define the labelling of Kripke structures. If we fix a Kripke structure, a CTL formula is satisfied by a set of states. We now define the syntax of CTL inductively and give a brief overview of its semantics. Let s be a state. If p is an atomic proposition, and φ and ψ are CTL formulas, the following are also CTL formulas: p (meaning that $p \in L(s)$), $\mathsf{AX}\,\varphi$ (meaning that φ holds for all successors of s), $\mathsf{AF}\,\varphi$ (meaning that for every path starting in s, φ holds eventually), $\mathsf{AG}\,\varphi$ (meaning that for all paths starting in s, φ holds in all states), and $\mathsf{A}\,\varphi\,\mathsf{U}\,\psi$ (meaning that on all paths starting in s, ψ holds eventually and φ holds in all prior states). We can use Boolean connectives to obtain $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$, with the usual meaning. We say that a formula holds for a Kripke structure if the formula is satisfied by all initial states.

Model Checker. We use the NuSMV model checker [9]. Given a Kripke structure and a CTL specification, it is able to decide if the specification holds for the Kripke structure. The NuSMV modelling language allows for a very easy description of Kripke structures and the CTL model checking functionality is based on a fast symbolic algorithm. For the subset of CTL that interests us, NuSMV produces a coun-

terexample whenever the model violates a formula. A counterexample for an $\mathsf{AG}\,\varphi$ formula consists of a single run of the model where φ does not always hold. A counterexample for an $\mathsf{AF}\,\varphi$ formula consists of a run of the model where φ never holds.

3. Modelling the Protocol

In order to model check the protocol, we need a Kripke structure that represents the protocol flow. It consists of four FSMs, one for each party of the protocol (client, TPM, server, intruder).[2] The states of these FSMs represent the steps of the protocol and the content of the messages, modelled as shared variables. Thus, the state of the overall system determines (1) the current step of the protocol for each party, (2) the content of the messages sent so far, and (3) the current knowledge of each party.

In order to describe the model of the intruder, we will first lay down some assumptions on the capabilities of the intruder.

3.1. Assumptions

Cryptography. Our assumptions on security and intruder capabilities are based on the considerations of Dolev and Yao [12]. We assume that the underlying cryptographic primitives are perfect and that keys and message fields are atomic. Thus, an intruder can read an encrypted message if and only if it knows the correct key. Similarly, an intruder can only create signed messages with signature keys it knows. An intruder is not able to learn partial information of a secret key or message, thus the intruder either knows it completely or not at all. Our model therefore does not cover attacks like statistical analysis or differential cryptanalysis, nor attacks based on mathematical or number-theoretic properties of the underlying cryptosystems.

We further assume that all parties know all public keys.

Model Restrictions. Our model introduces some further restrictions. First, we only model one run of the protocol. Thus, we do not consider replay or interleaved attacks. Furthermore, since the server trusts only the client's TPM but not the client itself, our model includes some malicious client behaviour. We do not model the complex API of the TPM. We thus assume that it is impossible to circumvent the TPM's security policies by abusing the API. We, however, allow the client to pass arbitrary values from its knowledge to the TPM during the protocol run. Due to these restrictions, our model cannot be used to obtain a proof that the protocol under investigation is secure under all circumstances.

Apart from these restrictions, our model of the intruder is quite powerful. We assume the intruder to be a classical man in the middle: the intruder can intercept all messages between the client and the server. Messages between the client and the TPM cannot be intercepted by the intruder.[3] If the intruder knows the correct key, it learns the content of the message and adds it to its knowledge. The intruder can also alter messages or introduce new messages which are composed of items in its knowledge.

3.2. Model Details

The model of the client, the server, and the TPM is a straightforward implementation of the protocol. Whenever a party is supposed to send a message, it fills the fields of the corresponding (global) variable and then sets a flag that the message has been sent. Then the sender remains in an inactive state until the `received` flag of the response message becomes `true`. When the receiver has received the message (possibly after the intruder has changed it), it can perform checks such as whether all expected fields in the message are non-empty, whether the message is signed with the expected key, etc. If all checks pass, the protocol continues until the state `COMPLETED` is reached. If a check fails, the party enters a state `ATTACK_DETECTED` and remains there forever.

We observe that the number of keys and other interesting items such as nonces in the protocol is limited. Thus, we represent them by a finite number of symbolic constants (`NONCE`, `CLIENT_KEY`, `SECRET`, etc.), in addition to the constant `ARBITRARY`, which represents "any other value".

The knowledge of the intruder is stored as an array of Boolean values, where each entry corresponds to one symbolic constant. The entry is `true` if and only if the intruder knows the information represented by the corresponding constant.

An encrypted message simply contains a field which stores the key with which the message should be decrypted. When the intruder sees the message, it checks whether it knows the key. If not, it can not perform any actions that require knowledge of the key. For example, without the key it can not add the content of the message to its knowledge. The same goes for signatures. If an intruder does not know the signature key of a message, it can not change the content of the message. However, note that the intruder is allowed to create a completely new message, either unsigned or signed, with another, known key.

The actions of the intruder are modelled according to the restrictions outlined above. Whenever the `sent` flag

[2] The NuSMV input files with the models we created and the results of running NuSMV on them are available for download at `http://www.iaik.tugraz.at/aboutus/people/hofferek/SSB_protocol_NuSMV_models.zip`.

[3] Eavesdropping on this channel would require physical access to the client's machine or access to the inter-process communication on it.

of a message is true, the intruder can start to operate on the message. It nondeterministically chooses to either leave the message as it is, or to construct a new one based on its current knowledge. When constructing a new message, the actual values of the fields are also chosen nondeterministically from the overall intruder's knowledge. The CTL model checker analyses *all* possible (nondeterministic) combinations. Thus, without explicitly modelling all choices in the state machine, they are all taken into account when checking the specification. After the intruder has dealt with a particular message, it sets its `received` flag to true. This indicates that the receiving party may continue its operation. It also prevents the intruder from making any more changes to the message. Note that the intruder still has read access to the message, which allows it to decrypt and use a message if it learns the corresponding key later on.

4. Results

4.1. Model Checking

When formalising the security targets given in Section 2.2 we obtain the following CTL properties:

1. $\mathsf{AG}(\neg \mathit{IntruderKnowledge}[\mathit{SECRET}])$

   The intruder never knows the secret.

2. $\mathsf{AG}((\mathit{Client.state} \neq V_{PCR}) \rightarrow \neg \mathit{ClientKnowledge}[\mathit{SECRET}])$

   If the client is not in the *trusted state*, which is described by the PCR values V_{PCR} then it does not know the secret.

3. $\mathsf{AG}((\mathit{Client.state} = V_{PCR}) \wedge \neg \mathit{AttackDetected} \rightarrow \mathsf{AF}(\mathit{ClientKnowledge}[\mathit{SECRET}]))$

   If the client is in the *trusted state* and no attack has been detected then the client should eventually know the (real) secret.

We feed our model and these properties to the NuSMV model checker. Within just a few seconds NuSMV finds the first two properties true and thus satisfies the specification given in [19]. The third property, however, is found to be false. An examination of the counterexample reveals a security issue.

Security Issues. The intruder is able to replace the last message from the server to the client. This message only consists of the encrypted secret. Thus the intruder can choose any arbitrary content it knows, encrypt it with the client's public key, and send it to the client. The client has no means of knowing whether the message has been altered or not. It can correctly decrypt the message, but instead of the secret it only learns the arbitrary content chosen by the intruder.

This prevents the two parties from establishing a shared secret. The secret value should not only remain confidential to just the server and the client, but it should also be assured that the client and server share the same secret in the end; yet this is no longer the case due to the aforementioned possibility of the intruder.

In certain scenarios this weakness does not cause a problem. Imagine a scenario in which the SECRET is a key to access some copyrighted intellectual property. If the client receives a faked secret, it will be unable to perform the necessary decryption operation, but otherwise no harm is done. This is no more dangerous than a denial-of-service attack.

Consider, however, another setting. Suppose the SECRET is a symmetric session key, which client and server want to use for confidential communication. After the protocol has been completed, the client uses the session key to encrypt confidential information (i.e. bulk data) and send it to the server. The client would think that (except for itself) only the server knows the session key. However, the session key is actually a faked one, sent by the intruder. Thus, the intruder is able to decrypt the message and learn the confidential content. The server would detect this attack, because the client's message is not encrypted with the expected session key. However, at this point it is too late: confidentiality has already been compromised.

4.2. Enhancements

The underlying problem is that the scheme only considers the security from the server's point of view. Indeed there is no authentication from the server to the client, neither in message 2 nor message 12.

The aforementioned attack can be prevented if the server signs the message in step 12 with its own key S. We also suggest to include the nonce N into message 12, so that it is uniquely linked to a particular protocol session:

12 C $\leftarrow$ S $\mathit{Sig}_S(\mathit{Enc}_K(\mathit{SECRET}), N)$

Now, if the intruder changes the content of the message the client can detect this, because the signature would be invalid. After we made this modification, NuSMV proved all three of our security properties to be true. However, we caution that the model checker cannot prove overall correctness of the protocol. It can only prove the correctness of the model of the protocol (with its limitations) with respect to the specified security properties.

Another possible enhancement is to have message 2 signed by the server. This allows the client to check whether or not V_{PCR} has been altered by the intruder. Without this signature an intruder would be able to obtain a TPM-signed message over whose content it has partial control. This

could help the intruder in cryptanalysis of the AIK in use.

5. Conclusions

Like any other security technology, Trusted Computing is no silver bullet, it does not magically remove or resolve every security issue present. Best practices for protocol design must still be adhered to, otherwise vulnerabilities in protocols may still exist.

In this paper we have formally modeled and analysed a scheme for the distribution of secrets with the NuSMV model checker. The main design goal of the scheme is to securely distribute a secret. We have shown that despite TPM-based encryption of secrets, the lack of authentication leads to a weakness in the cryptographic protocol, allowing an attacker to supply the client with a secret of its choice. We have modified the protocol and shown that the problem does not occur in our enhanced version.

Our experience shows that creating a model forces one to formalize all informal assumptions. Forgetting assumptions usually makes the model checker produce counterexamples, from which it is very easy to determine which additional checks are necessary. This helps the designer to gain confidence in the design and allows her to verify that changes actually lead to the desired improvements. Thus we propose to use model checkers in the spirit of unit-testing, assisting also the early design and specification phase of protocol creation.

In future work we desire to extend our method to more complex Trusted Computing protocols also considering the emerging field of analysis of security APIs [11].

Acknowledgements The authors thank David Basin, Paul Sevinç and Mario Strasser for comments on the paper.

References

[1] Security protocols open repository. Laboratoire Spécification et Vérification, ENS-Cachan, 2003. `http://www.lsv.ens-cachan.fr/spore/index.html`.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P.-C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of CAV'2005*, LNCS 3576, pages 281–285. Springer-Verlag, 2005.

[3] D. Basin, S. Mödersheim, and L. Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.

[4] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[5] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS '04: Proc. 11th ACM conference on computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM.

[6] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proc. 21st Annual Computer Security Applications Conference*, page 11pp., 5–9 Dec. 2005.

[7] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, September 2000.

[8] E. Cesena, G. Ramunno, and D. Vernizzi. Secure storage using a sealing proxy. In *EUROSEC '08: Proc. 1st European workshop on system security*, pages 27–34, New York, NY, USA, 2008. ACM.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, 2002.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[11] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *21st IEEE Computer Security Foundations Symposium*, pages 331–344, 2008.

[12] D. Dolev and A. Yao. On the security of public key protocols. In *IEEE Transactions in Information Theory*, volume 29(2), pages 198–207, March 1983.

[13] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[14] A. H. Lin. Automated analysis of security APIs. Master's thesis, Massachusetts Institute of Technology, May 2005.

[15] H. Löhr, H. V. Ramasamy, A.-R. Sadeghi, S. Schulz, M. Schunter, and C. Stüble. Enhancing grid security using trusted virtualization. In *ATC*, volume 4610 of *LNCS*, pages 372–384. Springer Verlag, 2007.

[16] A. Lomuscio. Automatic verification of knowledge and time with NuSMV. In *Proc. IJCAI 2007*, 2007.

[17] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *Selected Areas in Communications, IEEE Journal on*, 21(1):44–54, Jan 2003.

[18] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. *Security and Privacy, 1997. Proc., 1997 IEEE Symposium on*, pages 141–151, May 1997.

[19] P. E. Sevinç, M. Strasser, and D. Basin. Securing the distribution and storage of secrets with trusted platform modules. In *WISTP 2007*, pages 53–66, 2007.

[20] G. J. Simmons. Cryptanalysis and protocol failures. *Commun. ACM*, 37(11):56–65, 1994.

[21] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert. A robust integrity reporting protocol for remote attestation. In *Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, November 2006.

[22] Y. Zhang, C. Wang, J. Wu, and X. Li. Using SMV for cryptographic protocol analysis: a case study. *SIGOPS Oper. Syst. Rev.*, pages 43–50, 2001.