Securing the Distribution and Storage of Secrets with Trusted Platform Modules*

Paul E. Sevinç[1], Mario Strasser[2], and David Basin[1]

[1] Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland
{paul.sevinc, basin}@inf.ethz.ch

[2] Department of Information Technology and Electrical Engineering, ETH Zurich, 8092 Zurich, Switzerland
strasser@tik.ee.ethz.ch

Abstract. We present a protocol that allows servers to securely distribute secrets to trusted platforms. The protocol maintains the confidentiality of secrets in the face of eavesdroppers and careless users. Given an ideal (tamper-proof) trusted platform, the protocol can even withstand attacks by dishonest users. As an example of its use, we present an application to secure document processing.

1 Introduction

Trusted computing is about embedding a trusted computing base (TCB) [1] in a computing platform that allows a third party to determine the trustworthiness of the platform, i.e., whether or not the platform is a *trusted platform* from the point of view of a third party. The Trusted Computing Group (TCG), an industry standards organization, has specified a TCB for trusted computing in the form of three so-called roots of trust [2]: the *root of trust for storage* (RTS), the *root of trust for reporting* (RTR), and the *root of trust for measurement* (RTM). In particular, the TCG has specified a Trusted Platform Module (TPM) [3] that can act as both roots of trust for storage and measurement.[1] These specifications are clearly gaining momentum as witnessed by large-scale R&D projects such as *EMSCB* and *OpenTC*[2], open-source projects such as *TPM Emulator* [6] and *TrouSerS*[3], the inclusion of TPM services in *Windows Vista* [7], and the increasing number of personal computers with a TPM and a basic input/output system (BIOS) that can act as the RTM [8]. In the remainder of this paper, we understand "trusted computing" to mean trusted computing as specified by the TCG and focus on personal computers without limiting the paper's generality.

* This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

[1] There seems to be consensus in the information-security community that TCBs for trusted computing must be hardware-based, but tamper-proof hardware remains an open challenge [4,5].

[2] http://www.emscb.de/ and http://www.opentc.net/.

[3] http://tpm-emulator.berlios.de/ and http://trousers.sourceforge.net/.

D. Sauveron et al. (Eds.): WISTP 2007, LNCS 4462, pp. 53–66, 2007.
© IFIP International Federation for Information Processing 2007

Contribution. Our contribution in this paper is a protocol for securely distributing and storing secrets with TPMs. We specify the protocol in detail at the level of TPM commands and we informally analyze its security. The protocol is general in the sense that it is independent of a specific usage-control application. To illustrate how the protocol can be directly applied to nontrivial problems in usage control, we describe an application from the domain of secure document processing. In our specification, we treat the TPM as a trusted third party that can serve as an oracle for making platform measurements; this not only results in a clear specification, but this analogy for ideal roots of trust could also serve as the basis for a formal model of trusted computing in the future.

Organization. In Section 2, we provide a summary of those aspects of trusted computing that are of relevance to this paper. In Section 3, we discuss related work. In Section 4, we describe the problem that we solve with our protocol. We define the protocol and analyze its security in Sections 5 and 6, respectively. In Section 7, we draw conclusions. A concrete, realistic application scenario is presented in Appendix A.

2 Background

In this section, we summarize the TCG's definitions for root of trust for measurement, reporting, and storage. For a comprehensive description, the reader is referred to the TCG architecture overview [2] and to textbooks on trusted computing [9,10,11].

2.1 Root of Trust for Measurement

When a computer is booted, control passes between different subsystems. First the BIOS is given control of the computer, followed by the boot loader, the operating system loader, and finally the operating system. In an *authenticated boot*, the BIOS measures (i.e., cryptographically hashes) the boot loader prior to handing over control. The boot loader measures the operating system loader, and the operating system loader measures the operating system. These measurements reflect what software stack is in control of the computer at the end of the boot sequence; in other words, they reflect the platform configuration. Hence the name *platform configuration register* (PCR) for the TPM registers where such measurements are stored and which are initialized at startup and extended at every step of the boot sequence.

An attacker who wants to change the platform configuration without being detected has to corrupt the root of trust for measurement (in the BIOS), which we assume to be infeasible without physical access to the computer. Ideally, a tamper-proof piece of hardware will eventually act as the root of trust for measurement and measure the BIOS at the beginning of the boot sequence.

2.2 Root of Trust for Reporting

Each TPM has an *endorsement key* (EK) which is a signing key whose public key is certified by a trusted third party, such as the TPM manufacturer. For privacy reasons, the EK is only used to obtain a key certificate from a certificate authority (CA) for an *attestation identity key* (AIK), which the TPM generates itself. In order to alleviate even the strongest privacy concerns, direct anonymous attestation (DAA) [12,13] is the protocol of choice for certifying AIKs. AIKs are signing keys whose private key is only used for signing data that has originated in the TPM. For example, a remote party interested in learning what software stack is in control of the computer can query the TPM for PCR values. The query contains the set of PCRs to look up and a nonce (in order for the remote party to check for replay attacks). The TPM answers with the respective PCR values and the signature generated by signing the values as well as the nonce with one of its AIKs. Put differently, the TPM *attests* to, or *reports* on, the platform configuration.

2.3 Root of Trust for Storage

The protected storage feature of a TPM allows for the secure storage of sensitive objects such as TPM keys and confidential data. However, storage and cost constraints require that only the necessary (i.e., currently used) objects can reside inside a TPM; the remaining objects must be stored outside in unprotected memory and are revealed to the user or loaded into the TPM on demand. To this end, externally stored objects are encrypted (or *wrapped* in TCG terminology) with an asymmetric *storage key*, which is referred to as the parent key of the object. A parent key can again be stored outside the TPM and (possibly along with other keys) protected by another storage key. The thereby induced storage tree is rooted at the so called *storage root key* (SRK), which is created upon initialization of the TPM and cannot be unloaded. Consequently, a parent key has to be loaded into the TPM before the data it protects can be revealed or a key decrypted (or *unwrapped* in TCG terminology) and loaded into the TPM. Note that protected keys are only used inside the TPM and thus (in contrast to arbitrary data) are never disclosed to the user. Furthermore, each key is either marked as being *migrateable* or *non-migrateable*. In the former case, the key might be replicated and moved to other platforms whereas in the latter case the key is bound to an individual TPM and is never duplicated. Regarding the actual protection of objects, one differentiates between *binding* and *sealing*.

Binding is the operation of encrypting an object with the public key of a *binding key*. Binding keys are encryption keys. If the binding key is non-migratable, only the TPM that created the key can use its private key; hence, the encrypted object is effectively bound to a particular TPM.

Sealing takes binding one step further: the object is not only bound to a particular TPM, but in addition can only be decrypted if the current platform configuration matches the values associated with the protected object at the time of encryption.

It must be assumed that an attacker with physical access to the computer can get access to the private keys stored in the TPM. Current TPMs are designed to protect against software attacks, but not against hardware attacks (they are tamper-resistant at best) [10,11].

3 Related Work

Conceptually, properly measuring the software stack of a computer when the computer is booted (1), remotely attesting to a measurement and securely distributing secrets (2), and performing usage control on a computer once it has been deemed trustworthy and entrusted with the necessary secrets (3) are three straightforward tasks. As is often the case, the real complexity lies in the details.

Unlike the first task (e.g., [14,15,16]) and the third task (e.g., [17,18,19,20]), the second task has, until now, not been addressed in detail. Although protocols have been developed, those published are in the form of programming language-dependent and TPM library-dependent source code, without any security analysis. This may be the case because it is a deceptively simple task, which bears similarity with SSL/TLS. However, while the basic principles of SSL/TLS are fairly easy to understand, its details are quite intricate and the situation is similar here.

So far, the protocols for achieving the second task have been sketched at a very high level (similar to our summary in Figure 5 in Appendix A); for example in the form of the integrity-reporting protocol given in the TCG architecture overview [2, p. 9] and in the form of the two approaches for enhancing the protection of data on remote computers given by Pearson *et al.* [9, pp. 47-48]. This specification gap is filled in this paper.

4 Requirements

Consider the setting depicted in Figure 1: A server has secret data d_s that it is willing to share with certain clients over an open channel (i.e., one that is not encrypted in any way) upon request, but not with the clients' users. In practice (cf. the example given in Appendix A), the secret d_s may be a symmetric key K_D or the private key K_D^{-1} of an asymmetric key pair (K_D, K_D^{-1}). The owner of the secret d_s does not trust the users because they may not understand or respect the owner's security requirements or because they may have an untrustworthy platform, such as one compromised by a Trojan horse. In any case, the server is willing to share the secret d_s with clients who are known to meet the owner's security requirements. Because the main security goal of our protocol is *confidentiality* of the secret d_s, this entails that the client uses the secret d_s without disclosing it to the user or to any other entity. Furthermore, the client must either hinder the user from launching another process or at least force a change in the PCRs. Otherwise, the other (potentially malicious) process could simply request the TPM to disclose the secret d_s. It is in the server's interest to ensure that this security goal is met. Thus, the protocol needs to be resilient

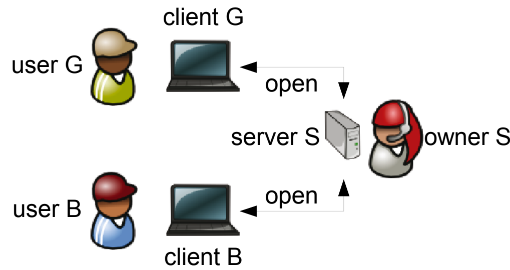

Fig. 1. Setting

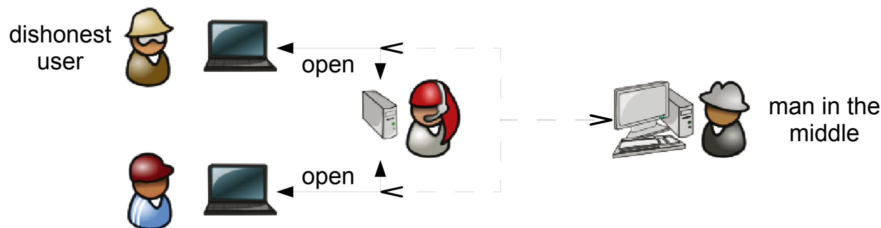

Fig. 2. Attackers

against man-in-the-middle attacks and, given an ideal (tamper-proof) trusted platform, against dishonest users as well (cf. Figure 2).

5 Protocol

In this section, we present a protocol that ensures that the server only distributes given secret data d_s to trusted clients (i.e., clients that meet the data owner's security requirements, as explained in the last section). The protocol involves three parties: the server, a client, and the client's TPM. Considering the TPM to be a participant in its own right may come as a surprise, but the following model should clarify this point.

Ideal roots of trust can be modeled as trusted third parties (cf. Figure 3) with certain oracle properties related to measurement. In particular, for each client there is a third party whom both the server and the client trust. Furthermore, the channel between the client and the trusted third party is secure (i.e., confidential and authentic). Not even the user can intercept or insert messages on this channel. There is no (direct) channel between the server and the trusted third party, though. Nevertheless, the server can encrypt data with the trusted third party's public encryption key along with information about a platform configuration. The trusted third party has the ability to determine the platform configuration of the client and decrypts data for the client only if the client's platform configuration is the one given when the data was encrypted.

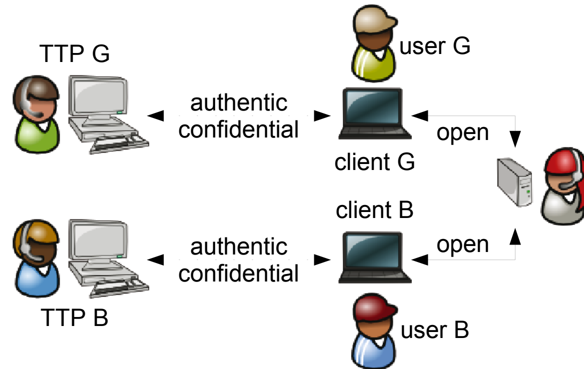

Fig. 3. Trusted Third Parties (TTPs) as a Model for Ideal Roots of Trust

In this model, our protocol basically proceeds as follows:

1. The client requests the secret d_s from the server.
2. The server encrypts d_s with the trusted third party's public key along with information about the platform configuration it trusts and sends the encrypted data to the client.
3. The client sends the encrypted data to the trusted third party and requests the trusted third party to decrypt the data for it.
4. The trusted third party determines the client's current platform configuration and reveals the decrypted data to the client only if the client's platform configuration is the one given when the data was encrypted.

Note that the first two steps only have to be taken once whereas the last two steps may be taken repeatedly.

The real protocol is more complex. In particular, it involves the generation of a platform configuration-dependent binding key and the use of an AIK. Even though privacy is not an issue for the kinds of application we originally had in mind, employing an AIK has the pleasant side effect of extending our protocol's usefulness to settings where privacy matters. For example, the secret data d_s could be the license key for a media player that manages digital rights. By using different AIKs when requesting license keys, the requests cannot be linked to the same client.

5.1 TPM Commands

We briefly introduce the TPM commands required in our protocol. For the sake of simplicity, we omit input values such as command-authorization data and key parameters as well as output values such as error codes. For a comprehensive description of the commands, the reader is referred to the TPM main specification [3] and to Pearson *et al.* [9].

TPM_CreateWrapKey generates an asymmetric key and returns the public key in plain text and the private key encrypted with the key pair's parent key. The input values of interest to us are a key handle that points to the generated key's parent key, the flag which declares the key as a binding or signing key, the flag which declares the key as migratable or non-migratable, and the (potentially empty) set of PCRs to whose values the key is sealed. Note that for the key to be non-migratable, the parent key must be non-migratable as well.

In our protocol, we use this command on the client to generate a non-migratable binding key that is sealed to a (non-empty) set of PCRs.

TPM_LoadKey2 loads an asymmetric key onto the TPM and returns the key handle that points to the loaded key, thus making the key available for use in subsequent TPM commands. The input values of interest to us are the public key, the encrypted private key, and a key handle that points to the loaded key's parent key. A non-migratable key will only be loaded onto the TPM if it was generated by the TPM.

In our protocol, we use this command on the client to load the binding key generated with TPM_CreateWrapKey onto the client TPM.

TPM_CertifyKey returns a key certificate. The input values of interest to us are a key handle that points to the key to certify and a key handle that points to the certifying signing key.

In our protocol, we use this command on the client to certify with an AIK that the binding key generated with TPM_CreateWrapKey and loaded with TPM_LoadKey2 is a non-migratable binding key that is sealed to a set of PCRs.

TSS_Bind encrypts data and returns it in cipher text. The input values of interest to us are the data to encrypt and the public key used for encryption. Note that it is the responsibility of the caller to ensure that the encryption key is a non-migratable binding key. Note further that the TSS_Bind command is not a TPM command, but fully implemented in software.

In our protocol, we use this command on the server to encrypt a secret with the public key of the binding key generated with TPM_CreateWrapKey and certified with TPM_CertifyKey.

TPM_UnBind decrypts data and returns it in plain text. The input values of interest to us are the data to decrypt and a key handle that points to the binding key (whose private key is used for decryption). A sealed binding key will only be used by the TPM if the values in the PCRs match those specified during sealing.

In our protocol, we use this command on the client to decrypt the secret encrypted with TSS_Bind with the private key of the binding key generated with TPM_CreateWrapKey and loaded with TPM_LoadKey2.

5.2 Notation

We employ so-called "Alice & Bob" notation, which leaves implicit many of the checks carried out by principals when executing the protocol and the associated

control flow when these checks fail, e.g., aborting when the verification of a digital signature fails [21]. Nevertheless, we have annotated our protocol specification, making explicit the checks of TPM-specific key properties and platform configuration information, using an assert statement. The semantics of assert is standard: it aborts the protocol execution when the asserted predicate does not hold. Furthermore, we employ the following notation:

- REQ is a constant, requesting the secret data d_s.
- PCR_INFO is a set of PCR indices and their respective values.
- $\mathcal{K}_X$ is a key pair with public key K_X and private key K_X^{-1}.
- H_X is the handle to the key $\mathcal{K}_X$.
- *aik*, *binding*, and *non-migratable* are flags that denote properties of a key, namely that the key is an AIK, a binding key, and non-migratable, respectively.
- $Enc_P(\mathit{binding}, \mathit{non\text{-}migratable}, PCR_INFO, K_C^{-1})$ is the private key K_C^{-1} of the non-migratable binding key $\mathcal{K}_C$ sealed to PCR_INFO, encrypted with the (non-migratable) parent key $\mathcal{K}_P$.
- $Sig_{AIK}(\mathit{binding}, \mathit{non\text{-}migratable}, PCR_INFO, N, K_C)$ is the certificate of the public key K_C of the non-migratable binding key $\mathcal{K}_C$ sealed to PCR_INFO, signed with the private key K_{AIK}^{-1} of the signing key $\mathcal{K}_{AIK}$.
- $Sig_{CA}(aik, K_{AIK})$ is the certificate of the public key K_{AIK} of the attestation identity key $\mathcal{K}_{AIK}$, signed with the private key K_{CA}^{-1} of the signing key $\mathcal{K}_{CA}$.

5.3 Initial Possessions of Parties

The **server** knows

- the secret data d_s,
- the public key K_{CA} of the certificate authority's signing key, and
- the PCRs and their values for the trusted stack.

The **client** knows

- the handle H_P to (and authorization data for) a non-migratable storage key $\mathcal{K}_P$,
- the handle H_{AI} to (and authorization data for) an AIK, and
- the certificate $Sig_{CA}(aik, K_{AIK})$ for verification of the AIK by a third party, in our case the server.

In a protocol run, the AIK allows the client to prove to the server that the latter is indirectly interacting with a TPM.
The **TPM** knows (i.e., has loaded)

- the private key K_P^{-1} of the non-migratable storage key $\mathcal{K}_P$ and
- the private key K_{AI}^{-1} of the AIK.

Table 1. Key Distribution Protocol

1	C ⟶ S	REQ
2	C ⟵ S	PCR_INFO, N
3	TPM ⟵ C	`TPM_CreateWrapKey`$(H_P, \textit{binding}, \textit{non-migratable}, PCR_INFO)$
4	TPM	assert $\mathcal{K}_P$ is *non-migratable*
		generate non-migratable binding key (K_C, K_C^{-1})
5	TPM ⟶ C	$K_C, Enc_P(\textit{binding}, \textit{non-migratable}, PCR_INFO, K_C^{-1})$
6	TPM ⟵ C	`TPM_LoadKey2`$(K_C,$
		$Enc_P(\textit{binding}, \textit{non-migratable}, PCR_INFO, K_C^{-1}), H_P)$
7	TPM ⟶ C	H_C
8	TPM ⟵ C	`TPM_CertifyKey`(H_C, H_{AIK}, N)
9	TPM ⟶ C	$Sig_{AIK}(\textit{binding}, \textit{non-migratable}, PCR_INFO, N, K_C)$
10	C ⟶ S	$Sig_{AIK}(\textit{binding}, \textit{non-migratable}, PCR_INFO, N, K_C)$,
		$Sig_{CA}(aik, K_{AIK})$
11	S	assert K_{AIK} is *aik*
		assert K_C is *binding*
		assert K_C is sealed to PCR_INFO
12	C ⟵ S	$Enc_C(d_s)$
13	TPM ⟵ C	`TPM_UnBind`$(Enc_C(d_s),\ H_C)$
14	TPM	assert C is in state PCR_INFO
15	TPM ⟶ C	d_s

Protocol Run. The protocol is specified in Table 1. The client initiates a protocol run by requesting the secret data d_s from the server (1). The server replies with the set of PCRs that have to be used to represent the (trusted) state of the client and a nonce N which identifies the protocol run (2). Note that the nonce does not provide additional security since replay attacks are not an issue.[4] The client invokes the `TPM_CreateWrapKey` command (3) to have the TPM create a non-migratable asymmetric encryption key $\mathcal{K}_C := (K_C, K_C^{-1})$ (4) that is sealed to the PCRs specified in step 2 (5). The client loads the key into the TPM by invoking `TPM_LoadKey2` (6), receives the key handle from the TPM (7), and has the TPM certify the loaded key with the AIK (8). The TPM returns the certificate (9), which the client forwards to the server together with the certificate of the AIK (10). The server checks that the certificates are valid (11), in particular that $\mathcal{K}_C$ is a non-migratable binding key sealed to the required

[4] The reason is that it is not important when the binding key has been generated and certified, but what its properties are. Because the binding key is non-migratable, these properties (in particular, which TPM it is associated with) never change. So even though the `TPM_CertifyKey` command is specified to take a nonce as argument, the nonce could be replaced with something predictable (and more efficiently implemented) like a counter in our protocol.

PCRs. If the key $\mathcal{K}_C$ has the required properties, the secret d_s is bound to it and the resulting protected object returned to the client (12). Upon receipt of the protected object, the client invokes `TPM_UnBind` to have the TPM decrypt the secret d_s (13). The TPM checks that the client is in the trusted state (14) before using $\mathcal{K}_C$ and returning the secret d_s (14).

Note that there is no need for (and no additional security in) explicitly sealing the secret d_s since the private key K_C^{-1} of the binding key $\mathcal{K}_C$ is sealed to the required PCRs itself. From now on, whenever the client is in the trusted state (i.e., the PCR values match the required ones) it can have the TPM unbind the secret d_s for its intended use.

6 Security Analysis

6.1 Security Against Man-in-the-Middle Attacks

Since the communication channel between the client and the server is open, a man in the middle sees the following four messages exchanged in steps 1, 2, 10, and 12:

1. REQ,
2. $PCR_INFO,\ N$,
3. $Sig_{AIK}(binding, non\text{-}migratable, PCR_INFO, N, K_C)$, $Sig_{CA}(aik, K_{AIK})$, and
4. $Enc_C(d_s)$

Obviously, the first three messages are independent of the secret data d_s, and hence they provide no information about it, even in a strict information-theoretic sense. The fourth message is encrypted with the public key K_C, where the corresponding private key K_C^{-1} is unknown to the man in the middle. Hence, the man in the middle can only decrypt the fourth message by breaking the cryptographic system, which we assume to be infeasible, or by deriving the private key K_C^{-1} from the first three messages. However, the first two messages are also completely independent of the encryption key $\mathcal{K}_C$ and deriving a private key from the corresponding public key in the third message amounts to breaking the cryptographic system. Thus, the protocol is also secure against a passive man-in-the-middle attack.

An active man in the middle could try to replace the third message he sees by $Sig_{MS}[enc,\ nm,\ PCR_INFO,\ N](K_{ME}), Sig_{CA}[sig](K_{MS})$—$\mathcal{K}_{MS}$ is his signing key and $\mathcal{K}_{ME}$ is his encryption key—in order to fool the server into binding the secret d_s to $\mathcal{K}_{ME}$ ($Enc_{ME}(d_s)$). However, this requires forging the CA's signature, which again amounts to breaking the cryptographic system. Thus, the protocol is also secure against an active man-in-the-middle attack.

6.2 Ideal Trusted Platform: Security Against Dishonest Users

Because the server verifies the two certificates exchanged in a protocol run, in particular that $\mathcal{K}_C$ is non-migratable and sealed to the PCRs specified by PCR_INFO,

and because it binds the secret data d_s to $\mathcal{K}_C$, the client must execute step 13 of the protocol as the honest client (i.e., using a trusted software stack). Afterwards, a dishonest user could put the client into a dishonest state (by launching another process if at all permitted or rebooting another stack) which results in different PCR values and in the TPM not unbinding the secret d_s. Alternatively, a dishonest user could mount a hardware attack in order to read the secret d_s out of the TPM, which we assume to be infeasible given an ideal (tamper-proof) trusted platform. Thus, an ideal (tamper-proof) trusted platform not only provides security against man-in-the-middle attacks but also against attacks by dishonest clients.

7 Conclusion

We have presented in detail a protocol at the level of TPM commands that allows servers to securely distribute secrets to trusted platforms. The protocol maintains the confidentiality of secrets in the face of eavesdroppers and careless users. Given an ideal (tamper-proof) trusted platform, the protocol maintains the confidentiality of secrets even in the face of dishonest users.

We have provided an informal analysis of the security of the protocol. A formal analysis of our protocol and other TPM-based protocols would require developing formal models for TPM-specific concepts such as binding and sealing. This is an interesting topic for future work that could be based on groundwork laid by Lin [22].

Acknowledgments

We would like to thank Michael Näf for his feedback on this paper and Thomas Zweifel for his previous collaboration on this topic. We also thank the artists who contribute to the Open Clip Art Library.

References

1. Bishop, M.: Computer Security: Art and Science. Addison Wesley Professional (2003)
2. Trusted Computing Group: TCG architecture overview. (TCG Specification)
3. Trusted Computing Group: TCG TPM specification version 1.2. (TCG Specification)
4. Anderson, R., Bond, M., Clulow, J., Skorobogatov, S.: Cryptographic processors – a survey. Technical Report 641, University of Cambridge (2005)
5. Smith, S.W.: Trusted Computing Platforms: Design and Applications. Springer-Verlag (2005)
6. Strasser, M.: A software-based TPM emulator for Linux. Semesterarbeit, ETH Zurich (2004)
7. Microsoft: Windows Vista beta 2 trusted platform module services step by step guide. (Published on the WWW)
8. Kay, R.L.: This ain't your father's internet: How hardware security will become nearly ubiquitous as a rock solid solution to safeguarding connected computing. (Published on the WWW)

9. Pearson, S., ed.: Trusted Computing Platforms: TCPA Technology in Context. Prentice Hall (2003)
10. Grawrock, D.: The Intel Safer Computing Initiative. Intel Press (2006)
11. Mitchell, C., ed.: Trusted Computing. Volume 6 of IEE Professional Applications of Computing. The Institution of Electrical Engineers (2005)
12. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In Pfitzmann, B., Liu, P., eds.: Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004), ACM Press (2004) 132–145
13. Camenisch, J.: Better privacy for trusted computing platforms. In Samarati, P., Ryan, P., Gollmann, D., Molva, R., eds.: Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS 2004). Volume 3193 of Lecture Notes in Computer Science., Springer-Verlag (2004) 73–88
14. Marchesini, J., Smith, S.W., Wild, O., MacDonald, R.: Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear. Technical Report Dartmouth TR2003-476, Dartmouth College (2003)
15. Marchesini, J., Smith, S.W., Wild, O., Barsamian, A., Stabiner, J.: Open-source applications of TCPA hardware. In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004), IEEE Computer Society (2004) 294–303
16. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th Usenix Security Symposium. (2004) 223–238
17. Sailer, R., Jaeger, T., Zhang, X., van Doorn, L.: Attestation-based policy enforcement for remote access. In: Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004). (2004) 308–307
18. Sandhu, R., Zhang, X.: Peer-to-peer access control architecture using trusted computing technology. In: Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005). (2005) 147–158
19. Zhang, X., Chen, S., Sandhu, R.: Enhancing data authenticity and integrity in p2p systems. IEEE Internet Computing **9**(6) (2005) 42 –49
20. Sandhu, R., Ranganathan, K., Zhang, X.: Secure information sharing enabled by trusted computing and pei models. In: Proceedings of the 2006 ACM Conference on Computer and Communications Security (ASIACCS 2006), ACM Press (2006) 2–12
21. Caleiro, C., Viganò, L., Basin, D.: Deconstructing Alice and Bob. In: Proceedings of the Workshop on Automated Reasoning for Security Protocol Analysis (ARSPA 2005). Volume 135 of Electronic Notes in Theoretical Computer Science. (2005) 3–22
22. Lin, A.H.: Automated analysis of security APIs. Master's thesis, Massachusetts Institute of Technology (2005)
23. Sevinç, P.E., Basin, D., Olderog, E.R.: Controlling access to documents: A formal access control model. In Müller, G., ed.: Proceedings of the 1st International Conference on Emerging Trends in Information and Communication Security (ETRICS 2006). Volume 3995 of Lecture Notes in Computer Science., Springer-Verlag (2006) 352–367
24. Sevinç, P.E., Basin, D.: Controlling access to documents: A formal access control model. Technical Report 517, ETH Zurich (2006)
25. Sevinç, P.E.: Securing Information by Controlling Access to Data in Documents. PhD thesis, ETH Zurich (2007)

A Application: Document Security

We apply our protocol in situations where access to documents of an enterprise must be controlled on computers that are not owned and administrated by the enterprise. For example, members of the enterprise's board of directors may be more inclined to read documents on a computer they already have, such as their home computer, than to be issued a computer from every enterprise on whose board of directors they sit.[5]

Such a situation is depicted in Figure 4. The main entities are a user, the user's computer (which the user owns and administrates), and a document whose content is confidential. Access to the document is governed by its access policy. The document content and the document policy are paired. The document's content is encrypted and the document as a whole signed such that the user can neither directly access the content component (because he does not know the decryption key) nor alter the policy in his favor (because he does not know the signing key). Instead, the user has to access the document via a document processor on an operating system that the document owner trusts to enforce the policy and to maintain confidentiality.

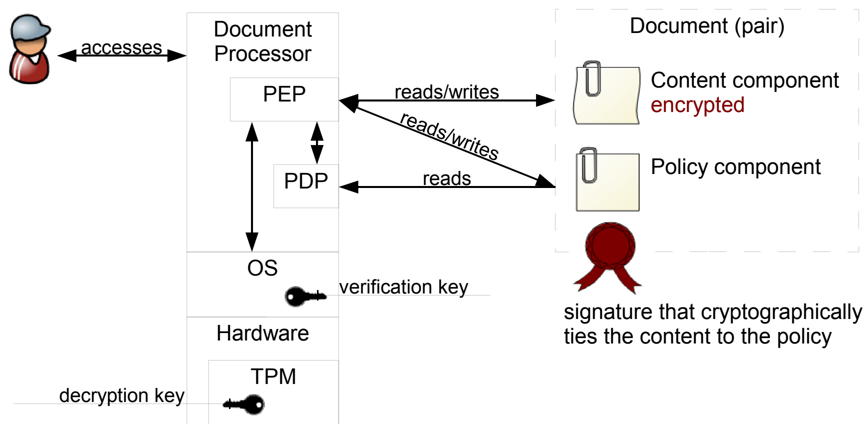

Fig. 4. Key Use

Within the document processor, the policy enforcement point (PEP) is responsible for enforcing the policy based on access decisions made by the policy decision point (PDP). Upon opening a document, the document processor has to verify the document's authenticity (which implies its integrity) and decrypt the

[5] Recall the case of former CIA Director John Deutch who accessed classified material on his unsecured home computer. The problem was not that he was not trusted (as the CIA director, he certainly was), but his software might have been untrustworthy. Had the classified material been encrypted with a key known not even to him, he could have been forced to boot his home computer into a trusted state, and the story would never have made the news.

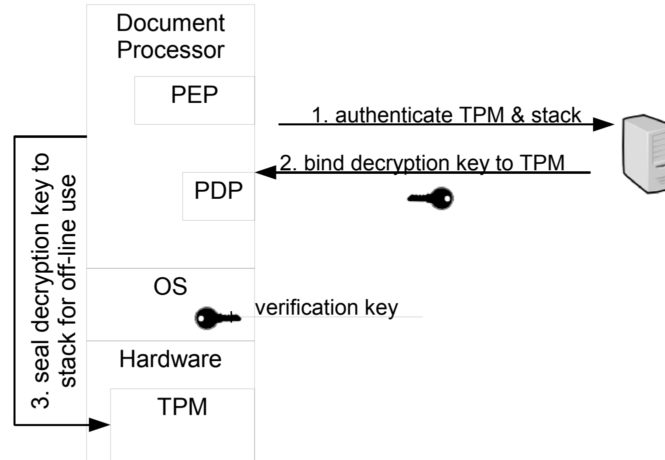

Fig. 5. Key Distribution

content. The decryption key is sealed to the trusted software stack (document processor and operating system) such that it can be used off-line but not accessed with another software stack in control. Note that the user cannot replace the verification key without changing (the hash of) the stack.

We achieve this situation in three main steps (cf. Figure 5):

1. The document owner's server checks whether the trusted stack is in control of the user's computer (steps 1–11 of the key distribution protocol).
2. It binds the decryption key to the user's TPM (step 12).
3. The trusted stack seals the decryption key in the user's TPM (steps 13–15).

In previous work [23,24,25], we developed an access-control system for documents. Standard operating system security mechanisms can be used to ensure that the system is not tampered with within an enterprise. This work is the missing link to ensuring that the system cannot be circumvented outside of the enterprise.