Bootstrapping Trust in a “Trusted” Platform

Bryan Parno
Carnegie Mellon University

Abstract

For the last few years, many commodity computers have come equipped with a Trusted Platform Module (TPM). Existing research shows that the TPM can be used to establish trust in the software executing on a computer. However, at present, there is no standard mechanism for establishing trust in the TPM on a particular machine. Indeed, any straightforward approach falls victim to a *cuckoo attack*. In this work, we propose a formal model for establishing trust in a platform. The model reveals the cuckoo attack problem and suggests potential solutions. Unfortunately, no instantiation of these solutions is fully satisfying, and hence, we pose the development of a fully satisfactory solution as an open question to the community.

1 Introduction

Before entrusting a computer with a secret, a user needs some assurance that the computer can be trusted. Without such trust, many tasks are currently impossible. For example, if Alice does not trust her PC, then she cannot login to any websites, read or edit confidential documents, or even assume that her browsing habits will remain private. While the need to bootstrap trust in a machine is most evident when traveling, this problem arises more generally. For example, Alice might wish to check her email on a friend’s computer. Alice may not even be able to say for sure whether she can trust her own personal computer.

One way to bootstrap trust in a computer is to use secure hardware mechanisms to monitor and report on the software state of the platform. Given the software state, the user (or an agent acting on the user’s behalf) can decide whether the platform should be trusted. Due to cost considerations, most commodity computers do not include a full-blown secure coprocessor, such as the IBM 4758 [11]. Instead, the move has been towards cheaper devices called Trusted Platform Modules (TPMs) [13]. The cost reduction is due in part to the decision to make the TPM secure only against software attacks. As a consequence, a TPM in the physical possession of an adversary cannot be trusted.

With appropriate software support, the TPM can be used to measure and record each piece of software loaded for execution, and securely convey this information (via an attestation) to a remote party [9, 13]. Thus, the TPM can be used to establish trust in the software on a machine.

However, the question remains: How do we bootstrap trust in the TPM itself? Surprisingly, neither the TPM specifications and nor the academic literature have considered this problem. Instead, it is assumed that the user magically possesses the TPM’s public key. While this assumption dispenses with the problem, it does not truly solve it, since in real life the user does not typically receive authentic public keys out of the blue. Without the TPM’s public key, the user cannot determine if she is interacting with the desired local TPM or with an adversarially-controlled TPM. For example, malware on the local machine may forward the user’s messages to a remote TPM that the adversary physically controls. Thus, the user cannot safely trust the TPM’s attestation, and hence cannot trust the computer in front of her.

As a result, we need a system to allow a conscientious user to bootstrap trust in the *local* TPM, so that she can leverage that trust to establish trust in the entire platform. In this paper, we make the following contributions:

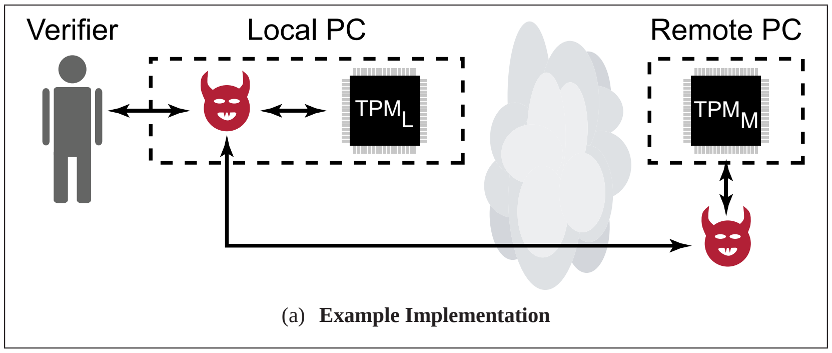
1. We formally define (using predicate logic) the problem of bootstrapping trust in a platform.
2. We show how the model captures the cuckoo attack, as well as how it suggests potential solutions.
3. We give sample instantiations of each type of solution and discuss their advantages and disadvantages.
4. None of the solutions feasible with current hardware are entirely satisfactory, so we recommend improvements for future platforms that aspire to be trusted.

2 Problem Definition

In this section, we provide a summary of the high-level TPM properties relevant to this work. We then give an informal description of the problem, followed by a more rigorous, formal definition.

2.1 TPM Background

From an abstract perspective, a Trusted Platform Module (TPM) can be described as a security chip equipped with a public/private keypair $\{K_{\text{TPM}}, K_{\text{TPM}}^{-1}\}$ and a set of secure memory regions known as Platform Configuration Registers (PCRs). While the TPM specification requires the TPM to preserve the secrecy of the private key K_{TPM}^{-1} and the integrity of the PCRs against software attacks, it makes no guarantees against determined hardware attacks.



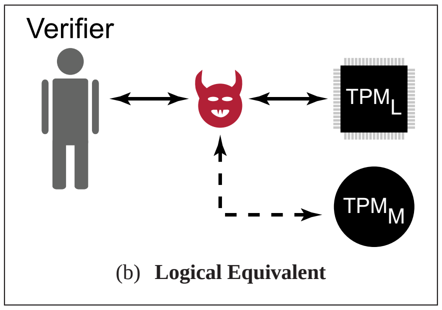


Figure 1: **The Cuckoo Attack.** *In one implementation of the cuckoo attack (a), malware on the user's local machine sends messages intended for the local TPM (TPM$_L$) to a remote attacker who feeds the messages to a TPM (TPM$_M$) inside a machine the attacker physically controls. Given physical control of TPM$_M$, the attacker can violate its security guarantees via hardware attacks. Thus, at a logical level (b), the attacker controls all communication between the verifier and the local TPM, while having access to an oracle that provides all of the answers a normal TPM would, without providing the security properties expected of a TPM.*

The TPM's manufacturer provides the TPM with an Endorsement Certificate (EC). The EC certifies that the TPM is a genuine, hardware TPM and serves to authenticate the TPM's public key K_{TPM}. Note that the EC only guarantees that the public key belongs to *some* TPM, not a *specific* TPM. This weakness can be exploited via a cuckoo attack.

Assuming an external verifier can obtain a TPM's authentic public key K_{TPM}, the TPM can securely describe the software state of the platform to the verifier via an *attestation*. Software on the platform records measurements (hashes) of software executed into the PCRs. Given a cryptographic nonce from the verifier, the TPM can produce a signature over the nonce and the current PCR values. The signature is calculated using[1] the private 2048-bit RSA key K^{-1}_{TPM}. The verifier can use the public key to verify the signature and hence authenticate the PCR values. Using the PCR values, the verifier can decide whether to trust the software on the computer with the user's secrets.

2.2 Informal Problem Description

Our high-level goal is to establish trust in a potentially compromised computer, so that a user can perform security-sensitive tasks. To achieve this goal, we must assume the user already trusts someone or something, and then leverage that trust to establish trust in the computer.

Specifically, we make two initial trust assumptions. First, we assume the user has a mobile, trusted device, such as a cellphone, or a special-purpose USB fob that can compute and communicate with the computer. This device is assumed to be trusted in part due to its limited interface and functionality,[2] so it cannot be used for general security-sensitive tasks. We also assume the user trusts someone (potentially herself) to vouch for the physical integrity of the local machine. Without this assumption (which may not hold for kiosk computers), it is difficult to enable secure, general-purpose computing. Fortunately, humans are relatively good at protecting their physical belongings (as opposed to virtual belongings, such as passwords). Furthermore, the assumption is true relative to Internet-based attackers.

Ideally, from these two trust assumptions (a trustworthy verifier device and a physically secure local computer), we would establish trust in the secure hardware (TPM) in the local computer. Trust in the TPM could then be used to establish trust in the software on the computer. Unfortunately, there is currently no way to connect our trust assumptions to trust in the local TPM. When a user walks up to a computer, she has no reliable way of establishing the identity (public key) of the TPM inside the computer. As a result, she may fall victim to what we call a cuckoo attack.

In a *cuckoo attack*,[3] the adversary convinces the user that a TPM the adversary physically controls in fact resides in the user's own local computer. Figure 1(a) illustrates one possible implementation of the cuckoo attack. Malware on the user's local machine proxies the user's TPM-related messages to a remote, TPM-enabled machine controlled by the attacker. The attacker's TPM$_M$ can produce an EC certifying that the TPM's public key K_{TPM_M} comes from an authentic TPM. The attacker's computer then faithfully participates in the TPM protocol, and it provides an attestation that trusted software has been loaded correctly.

As a result, the user will decide to trust the local PC. Any secrets she enters can be captured by malware and forwarded to the attacker. Even secrets protected by TPM-based guarantees (e.g., encrypted using K_{TPM_M}) will be compromised, since the TPM's specifications offer no guarantees for a TPM in the physical possession of the adversary.

[1]Technically, most TPMs use an Attestation Identity Key (AIK) to sign the attestation. An AIK is a public keypair generated by the TPM. Using the TPM's private key and EC, a TPM owner can convince a Privacy Certificate Authority to issue an anonymous certificate indicating that the public portion of the AIK was generated by a legitimate TPM. Since the AIK's authenticity is based on the EC, it is vulnerable to the cuckoo attack. As an alternative to AIKs, the v1.2 TPM specification [13] includes support for Direct Anonymous Attestation (DAA). However, the goal of DAA is to demonstrate that *a* TPM signed the attestation while preventing the verifier from discovering *which* TPM signed it, thus allowing the cuckoo attack.

[2]Arguably, this assumption does not hold for current smartphones.

[3]The cuckoo bird replaces other birds' eggs with its own. The victim birds are tricked into feeding the cuckoo chick as if it were their own. Similarly, the attacker "replaces" the user's trusted TPM with his own TPM, leading the user to treat the attacker's TPM as her own.

Predicates

Predicate	Meaning
$\mathtt{TrustedPerson}(p)$	User trusts person p.
$\mathtt{PhysSecure}(c)$	Computer c is physically secure.
$\mathtt{SaysSecure}(p, c)$	Person p says computer c is physically secure.
$\mathtt{Trusted}_{\mathtt{C}}(c)$	Computer c is trusted.
$\mathtt{Trusted}_{\mathtt{T}}(t)$	TPM t is trusted.
$\mathtt{On}(t, c)$	TPM t resides on computer c.
$\mathtt{CompSaysOn}(c, t)$	Computer c says TPM t is installed on computer c.

Axioms

1. $\forall p, c$	$\mathtt{TrustedPerson}(p) \land \mathtt{SaysSecure}(p, c) \rightarrow \mathtt{PhysSecure}(c)$
2. $\forall t, c$	$\mathtt{On}(t, c) \land \neg\, \mathtt{PhysSecure}(c) \rightarrow \neg\, \mathtt{Trusted}_{\mathtt{T}}(t)$
3. $\forall t, c$	$\mathtt{On}(t, c) \land \mathtt{PhysSecure}(c) \rightarrow \mathtt{Trusted}_{\mathtt{T}}(t)$
4. $\forall t, c$	$\mathtt{On}(t, c) \land \mathtt{Trusted}_{\mathtt{T}}(t) \rightarrow \mathtt{Trusted}_{\mathtt{C}}(c)$
5. $\forall t, c$	$\mathtt{On}(t, c) \land \neg\, \mathtt{Trusted}_{\mathtt{T}}(t) \rightarrow \neg\, \mathtt{Trusted}_{\mathtt{C}}(c)$
6. $\forall c, t$	$\mathtt{CompSaysOn}(c, t) \rightarrow \mathtt{On}(t, c)$

Figure 2: **Trust Model.** *The predicates describe relevant properties of the system, while the axioms encode facts about the domain.*

Assumption	Encoding
1. Alice trusts herself.	$\mathtt{TrustedPerson}(\text{Alice})$
2. Alice says her computer C is physically secure.	$\mathtt{SaysSecure}(\text{Alice}, C)$
3. The adversary controls machine M containing TPM_M.	$\mathtt{On}(\text{TPM}_M, M)$
4. M is not physically secure.	$\neg\, \mathtt{PhysSecure}(M)$
5. Malware on Alice's machine C causes it to say that TPM_M is installed on C.	$\mathtt{CompSaysOn}(C, \text{TPM}_M)$

Figure 3: **Assumptions.** *We encode our assumptions about the situation in predicates.*

(1)	$\mathtt{TrustedPerson}(\text{Alice})$	Assumption 1
(2)	$\mathtt{SaysSecure}(\text{Alice}, C)$	Assumption 2
(3)	$\mathtt{PhysSecure}(C)$	Axiom 1: (1), (2)
(4)	$\mathtt{CompSaysOn}(C, \text{TPM}_M)$	Assumption 5
(5)	$\mathtt{On}(\text{TPM}_M, C)$	Axiom 6: (4)
(6)	$\mathtt{Trusted}_{\mathtt{T}}(\text{TPM}_M)$	Axiom 3: (5), (3)
(7)	$\mathtt{Trusted}_{\mathtt{C}}(C)$	Axiom 4: (5), (6)
(8)	$\mathtt{On}(\text{TPM}_M, M)$	Assumption 3
(9)	$\neg\, \mathtt{PhysSecure}(M)$	Assumption 4
(10)	$\neg\, \mathtt{Trusted}_{\mathtt{T}}(\text{TPM}_M)$	Axiom 2: (8), (9)
(11)	$\neg\, \mathtt{Trusted}_{\mathtt{C}}(C)$	Axiom 5: (5), (10)
(12)	$\perp$	7, 11

Figure 4: **Proof.** *Applying our axioms to our assumptions leads to a logical contradiction.*

2.3 Formal Model

To analyze the cuckoo attack more formally, we can model the situation using predicate logic. Figure 2 summarizes our proposed model for establishing trust in a computer equipped with secure hardware. The first axiom encodes our assumption that trusted humans can vouch for the physical integrity of a computer. The next two axioms codify the TPM's vulnerability to hardware attacks. The second set of axioms encodes our assumption that trust in the TPM inside a computer suffices (via software attestations) to establish trust in the computer. The final axiom represents the fact that today, without the local TPM's public key, the user must accept the computer's assertion that a particular TPM resides on the computer.

To "initialize" the system, we also encode our assumptions about the concrete setting in a set of predicates (shown in Figure 3). By applying our set of axioms to the initial assumptions, we can reason about the trustworthiness of the local machine. Unfortunately, as shown in Figure 4, such reasoning leads to a logical contradiction, namely that the local machine C is both trusted and untrusted. This contradiction captures the essence of the cuckoo attack, since it shows that the user cannot decide whether she should trust the local machine.

Removing the contradiction requires revisiting our axioms or our assumptions. We explore these options below.

3 Solutions

The cuckoo attack is possible because the attacker can convince the user to accept assurances from an untrustworthy TPM. In this section, we first show that an obvious solution, cutting off network access, addresses one instantiation of the cuckoo attack but does not solve the problem, since malware on the local machine may have enough information to perfectly emulate a TPM in software. To avoid similar missteps, we return to our formal model and consider solutions that remove an assumption, as well as solutions that fix an axiom. For each approach, we provide several concrete instantiations and an analysis of their advantages and disadvantages.

3.1 Removing Network Access Insufficient

From Figure 1(a), it may seem that the cuckoo attack can be prevented by severing the connection between the local malware the adversary's remote PC. The assumption is that without a remote TPM to provide the correct responses, the infected machine must either refuse to respond or allow the true TPM to communicate with the user's device (thus, revealing the presence of the malware).

Below, we suggest how this could be implemented, and show that regardless of the implementation, this solution fundamentally does not work. We demonstrate this both with the formal model from Section 2.3, and with an attack.

There are several ways to remove the local malware's access to the remote TPM. We could instruct the user to sever all network connections. If the user cannot be trusted to reliably accomplish this task,[4] the verifier could jam the network connections. For example, the user's fob might include a small RJ-45 connector to plug the Ethernet jack and jam the wireless network at the logical level (by continuously sending Request-to-Send frames) or at the physical level. Finally, we could use a distance-bounding protocol [3] to prevent the adversary from making use of a remote TPM. Since the speed of light is constant, the verifier can require fast responses from the local platform and be assured that malware on the computer does not have time to receive an answer from a remote party. However, with current TPMs, identification operations take half a second or more, with considerable variance both on a single TPM and across the various TPM brands. A signal traveling at light speed can circle the earth about four times in the time required for an average TPM to compute a signature, making distance-bounding infeasible.

Unfortunately, removing network access is fundamentally insufficient to prevent the replay attack. One way to see this is via the formal model from Figure 2. Neither the predicates nor the axioms assume the local adversary has access to the remote PC. The logical flaw that allows the cuckoo attack to happen arises from Axiom 6, i.e., the local computer's ability to convince the user that a particular TPM resides on it. In other words, as shown in Figure 1(b), the cuckoo attack is possible because the malware on the local machine has access to a "TPM oracle" that provides TPM-like answers without providing TPM security guarantees. If the local malware can access this oracle without network access, then cutting off network access is insufficient to prevent the cuckoo attack.

In particular, since the adversary has physical possession of TPM_M, he can extract its private key. He can then provide the malware on the local computer with the private key, TPM_M's Endorsement Certificate, and a list of trusted PCR values. Thus provisioned, the malware on the local machine can perfectly emulate TPM_M, even without network access.

3.2 Eliminate Malware

An alternate approach is to try to remove the malware on Alice's local computer. In our formal model, this equates to removing Assumption 5, which would remove the contradiction that results in the cuckoo attack. Unfortunately, this approach is both circular and hard to achieve.

First, we arrived at the cuckoo attack based on the goal of ensuring that the local machine could be trusted. In other words, the goal is to detect (and eventually remove), any malware on the machine using the TPM. Removing malware in order to communicate securely with the TPM, in order to detect and remove malware, potentially leaves us stuck in an endless loop.

[4]For example, it may be difficult to tell if an infected laptop has its wireless interface enabled.

In practice, there are two approaches to cutting through this circularity, but neither is satisfactory.

§1 **Trust.** The "null" solution is to simply ask the local machine for its key and trust that no malware is present.
Pros: This is clearly the simplest possible solution.
Cons: The assumption that the machine is not compromised will not hold for many computers. Unprotected Windows PCs are infected in minutes [1]. Even newly purchased devices may not meet this criteria [6, 12].

§2 **Timing Deviations.** Seshadri et al. note [10] that certain computations can be done faster locally than malware can emulate the same computations while hiding its own presence. By repeating these computations, a timing gap appears between a legitimate execution of the protocol, and a malware-simulated execution. Using their system, Pioneer, we can check for malware.
Pros: Since Pioneer does not rely on special hardware, it can be employed immediately on current platforms.
Cons: Using Pioneer requires severing the PC's network access; Section 3.1 shows that this is non-trivial. Also, Pioneer requires specific hardware knowledge that the user is unlikely to possess.

3.3 Establish a Secure Channel

Given the conclusions above, we must keep the assumptions in Figure 3. Thus, to find a solution, we must fix one or more of our axioms. We argue that the correct target is Axiom 6, as the others are fundamental to our problem definition.

We cannot simply remove Axiom 6, since without it, we cannot introduce the notion of a TPM being installed on a computer. Instead, establishing a secure (authentic and integrity-preserving) channel to the TPM on the local machine suffices to fix Axiom 6. Such a secure channel may be established using hardware or cryptographic techniques.

For a hardware-based approach, we would introduce a new predicate $\texttt{HwSaysOn}(t,c)$ indicating that a secure hardwired channel allowed the user to connect to the TPM on the local machine. Axiom 6 would then be written as:

$$\forall t,c \quad \texttt{HwSaysOn}(t,c) \rightarrow \texttt{On}(t,c)$$

A cryptographic approach requires the user to obtain some authentic cryptographic information about the TPM she wishes to communicate with. Based on the user's trust in the source of the information, she could then decide that the TPM was in fact inside the machine. We could encode this using the predicate $\texttt{PersonSaysOn}(p,\ t,\ c)$ indicating that a person p has claimed that TPM t is inside computer c. Axiom 6 would then be written as:

$$\forall p,t,c \quad \texttt{TrustedPerson}(p) \wedge \texttt{PersonSaysOn}(p,t,c) \\ \rightarrow \texttt{On}(t,c)$$

3.3.1 Hardware-Based Secure Channels

Below, we analyze ways to establish a secure channel with the TPM on the local computer.

§3 **Special-Purpose Interface.** Add a new hardware interface to the computer that allows an external device to talk directly to the TPM.
Pros: The use of a special-purpose port reduces the chances for user error (since they cannot plug the external verifier into an incorrect port).
Cons: Introducing an entirely new interface and connector specification would require significant industry collaboration and changes from hardware manufacturers, making it an unlikely solution in the near term.

§4 **Existing Interface.** Use an existing external interface (such as Firewire or USB) to talk directly to the TPM.
Pros: This solution is much simpler to deploy, since it does not require any manufacturer changes.
Cons: Existing interfaces are not designed to support this type of communication. For example, USB devices cannot communicate with the host platform until addressed by the host. Even devices with more freedom, such as Firewire devices, can only read and write to memory addresses. While the TPM is made available via memory-mapped I/O ports, these mappings are established by the software on the machine, and hence can be changed by malware. Thus, there does not appear to be a way to reuse existing interfaces to communicate reliably with the local TPM.

§5 **External Late Launch Data.** Recent CPUs from AMD and Intel can perform a *late launch* of an arbitrary piece of code. During the late launch, the code to be executed is measured and the measurement is sent to the TPM. The code is then executed in a protected environment. If the late launch operation also made the code's measurement code available externally, then the user's verifier could check that the invoked code was trustworthy. The code could then check the integrity of the platform or establish a secure channel from the verifier to the TPM.
Pros: Recent CPUs contain the late launch functionality needed to measure and securely execute code.
Cons: Existing interfaces (such as USB) do not allow the CPU to convey the fact that a late launch occurred nor the measurement of the executed code in an authentic fashion. Malware on the computer could claim to perform a late launch and then send a measurement of a legitimate piece of code. This attack could be prevented by creating a special-purpose interface that talks directly to the CPU, but this brings us back to §3, which is a simpler solution.

§6 **Special-Purpose Button.** Add a new button on the computer for bootstrapping trust. For example, the button can execute an authenticated code module that establishes a secure channel between the verifier (connected via USB, for example) and the TPM.
Pros: A hardware button press cannot be overridden by malware. It also provides the user with a tangible guarantee that secure bootstrapping has been initiated.
Cons: Executing an authenticated code module requires hardware not only for invoking the necessary code, but also for verifying digital signatures (similar to §9), since the code will inevitably need updates. This approach also relies on the user to push the button before connecting the verifier device, since the device cannot detect the button push. If the user plugs in the verifier before pushing the button, on the computer could fool the device with a cuckoo attack.

3.3.2 Cryptographic Secure Channels

Establishing a cryptographically-secure channel requires the user to share a secret with the TPM or to obtain the TPM's public key. Without a prior relationship with the TPM, the user cannot establish a shared secret, so in this section we focus on public-key methods.

§7 **Seeing-is-Believing (SiB).** An approach suggested by McCune et al. [8] (and later used for kiosk computing [4]) is to have the computer's manufacturer encode a hash of the platform's identity in a 2-D barcode and attach the barcode to the platform's case. Using a camera-equipped smartphone, the user can take a picture of the 2-D barcode and use the smartphone to process the computer's attestation.
Pros: This solution is attractive, since it requires relatively little effort from the manufacturer, and most people find picture-taking simple and intuitive.
Cons: Because it requires a vendor change, this solution will not help current platforms. It also requires the user to own a relatively expensive smartphone and install the relevant software. The user must also trust that the smartphone has not been compromised. As these phones grow increasingly complex, this assumption is likely to be violated. In a kiosk setting, the 2-D barcode may be replaced or covered up by an attacker.

§8 **SiB Without a Camera.** Instead of using a 2-D barcode, the manufacturer could encode the hash as an alphanumeric string. The user could then enter this string into a smartphone, or into a dedicated fob.
Pros: Similar to §7, except the user no longer needs a camera-equipped device.
Cons: Similar to those of §7, but it still requires non-trivial input capability on the user's device. Relies on the user to correctly enter a complicated string.

§9 **Trusted BIOS.** If the user trusts the machine's BIOS, she can reboot the machine and have the trusted BIOS output the platform's identity (either visually or via an external interface, such as USB). The trusted BIOS must be protected from malicious updates. For example, some Intel motherboards will only install BIOS updates signed by Intel [5].
Pros: This approach does not require the user to use any custom hardware.
Cons: The user must reboot the machine, which may be disruptive. It relies on the user to only insert the verifier after rebooting, since otherwise the verifier may be deceived by local malware. The larger problem is that many motherboards do not include the protections necessary to guarantee the trustworthiness of the BIOS, and there is no indicator to signal to the user that the BIOS in the local computer is trustworthy.

§10 **Trusted Third Party.** The TPM could be equipped with a certificate provided by a trusted third-party associating the TPM with a particular machine. The verifier can use the trusted third party's public key to verify the certificate and establish trust in the TPM's public key.

Pros: The verifier only needs to hold the public key for the trusted third party and perform basic certificate checks. No hardware changes are needed.

Cons: It is unclear how the verifier could communicate the TPM's location as specified in the certificate to the user in a clear and unambiguous fashion. This solution also simply moves the problem of establishing a TPM's identity to the third party, who would need to employ one of the other solutions suggested here.

4 Preferred Solutions

We argue that §3 (a special-purpose hardware interface) provides the strongest security. It removes almost every opportunity for user error, does not require the preservation of secrets, and does not require software updates. Unfortunately, the cost and industry collaboration required to introduce a new interface make it unlikely to be deployed.

Of the plausibly deployable solutions, we argue in favor of §8 (an alphanumeric hash of the TPM's public key), since it allows for a simpler verification device.

Nonetheless, we recognize that these selections are open to debate, and believe that considerable room remains for additional solutions.

5 Related Work

Measurement. Various TPM-based systems have been proposed, including the Integrity Measurement Architecture by Sailer et al. [9], and the more recent, late-launch-based Flicker [7]. To date, these systems assume that the external verifier has somehow obtained the TPM's authentic public key, thus ducking the bootstrapping problem.

Device Pairing. Considerable work has studied how to establish secure communication between two devices. Proposals use infrared, visual, and audio channels, as well as physical contact, shared acceleration, and even the electrical conductivity of the human body. Unlike this work, in these systems, the two devices are trusted, and the adversary is assumed to be an external entity.

Kiosk Computing. Garriss et al. study the problem of kiosk computing [4], a specific case of the problem considered here. They note the potential for a cuckoo attack (though not by that name) and propose solution §7, which has the advantages and disadvantages described above.

Secure Object Identification. In the realm of access control, researchers have studied a related problem known as the Chess Grandmaster Problem, Mafia Fraud, or Terrorist Fraud [2], in which an adversary acts as a prover to one honest party and a verifier to another party in order to obtain access to a restricted area. Existing solutions rely on distance bounding [3], which, as explained in Section 3.1, is ineffective for a TPM, or employ radio-frequency hopping [2] which is also infeasible for the TPM.

6 Conclusion

Trust in a local computer is necessary for a wide variety of important tasks. Ideally, we should be able to use secure hardware, such as the TPM, to leverage our trust in the physical security of the machine in order to trust the software executing on the platform. Our formal model reveals that current attempts to create this chain of trust are vulnerable to the cuckoo attack. The model is also useful for identifying solutions, though we find that instantiations of these solutions come with multiple disadvantages. We hope that additional research into trust establishment will provide more elegant solutions that can be easily deployed and yet provide strong security guarantees.

7 Acknowledgements

Lujo Bauer and Scott Garriss provided invaluable assistance with the development of the formal logic model. Discussions and editing suggestions from Jonathan McCune and Diana Parno greatly improved the paper, as did discussions with Adrian Perrig. The anonymous reviewers also provided helpful suggestions. The author is supported in part by a National Science Foundation Graduate Research Fellowship.

References

[1] B. Acohido and J. Swartz. Unprotected PCs can be hijacked in minutes. *USA Today*, Nov. 2004.

[2] A. Alkassar, C. Stüble, and A.-R. Sadeghi. Secure object identication or: Solving the chess grandmaster problem. In *Proceedings of the New Security Paradigm Workshow (NSPW)*, 2003.

[3] S. Brands and D. Chaum. Distance-bounding protocols. In *EUROCRYPT*, 1994.

[4] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. *To Appear in Proceedings of MobiSys*, 2008.

[5] P. Lang. Flash the Intel BIOS with confidence. *Intel Developer UPDATE Magazine*, Mar. 2002.

[6] J. LeClaire. Apple ships iPods with Windows virus. *Mac News World*, Oct. 2006.

[7] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of EuroSys*, Apr. 2008.

[8] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing is believing: Using camera phones for human-verifiable authentication. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[9] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of USENIX Security Symposium*, Aug. 2004.

[10] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of SOSP*, Oct. 2005.

[11] S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31, 1999.

[12] Sophos. Best Buy digital photo frames ship with computer virus, Jan. 2008.

[13] Trusted Computing Group. Trusted platform module main specification. Version 1.2, Revision 103, July 2007.