Cloaking Malware with the Trusted Platform Module

Alan M. Dunn Owen S. Hofmann Brent Waters Emmett Witchel
The University of Texas at Austin
{*adunn,osh,bwaters,witchel*}*@cs.utexas.edu*

Abstract

The Trusted Platform Module (TPM) is commonly thought of as hardware that can increase platform security. However, it can also be used for malicious purposes. The TPM, along with other hardware, can implement a *cloaked computation*, whose memory state cannot be observed by any other software, including the operating system and hypervisor. We show that malware can use cloaked computations to hide essential secrets (like the target of an attack) from a malware analyst.

We describe and implement a protocol that establishes an encryption key under control of the TPM that can only be used by a specific infection program. An infected host then proves the legitimacy of this key to a remote malware distribution platform, and receives and executes an encrypted payload in a way that prevents software visibility of the decrypted payload. We detail how malware can benefit from cloaked computations and discuss defenses against our protocol. Hardening legitimate uses of the TPM against attack improves the resilience of our malware, creating a Catch-22 for secure computing technology.

1 Introduction

The Trusted Platform Module (TPM) has become a common hardware feature, with 350 million deployed computers that have TPM hardware [14]. The purpose of TPM hardware, and the software that supports it, is to increase the security of computer systems. However, this paper examines the question of how a malware author can use the TPM to build better malware, specifically malware that cannot be analyzed by white hat researchers.

Trusted computing technology [42] adds computer hardware to provide security primitives independent from other system functionality. The hardware provides certain low-level security guarantees directly. For example, it guarantees that only it can read and write certain data. Trusted software uses these low-level, hardware-enforced properties to build powerful guarantees for programmers.

The TPM, as developed by the Trusted Computing Group (TCG), is one of the more popular implementations of trusted computing technology. The TPM has seen significant use in industry and government; the TPM is used in Microsoft's popular BitLocker drive encryption software [7] and the United States Department of Defense has required the TPM as a solution for securing data on laptops [4]. TPMs are regularly included on desktop, laptop, and server-class computers from a number of manufacturers. The wide dissemination of TPM functionality is potentially a boon for computer security, but this paper examines the potential of the TPM for malware authors (a first to our knowledge).

A malware writer can use the TPM for implementing **cloaked computations** which, combined with a protocol described in this paper, impede malware analysis. The TPM is used with "late launch" processor mechanisms (Intel's Trusted Execution Technology [12, 8], abbreviated TXT, and AMD's Secure Startup mechanism [10]) that ensure uninterrupted execution of secure binaries. Late launch is a hardware-enforced secure environment where code runs without any other concurrently executing software, including the operating system. We demonstrate a protocol where a malware author uses cloaked computations to completely prevent certain malware functions from being analyzed and understood by any currently available methods. TPM functionality ensures that a cloaked program will remain encrypted until it is running directly on hardware. Assuming certificates for hardware TPMs identify these TPMs as hardware and cannot be forged, our malware will refuse to execute in a virtualized environment.

Timely and accurate analysis is critical to the ability to stop widespread effects of malware. Honeypots are constantly collecting malware and researchers use creative combinations of static analysis, dynamic emulation and virtualization to reverse engineer malware behavior [47, 30, 19, 24, 35, 36]. This reverse engineering is often crucial to defeating the malware. For example, once the domain name generation algorithm used for propagating the Conficker worm was determined, the Conficker cabal blocked the registration of those DNS names [45, 43], thereby defeating the worm.

While the idea of using the TPM to cloak malware computation is conceptually straightforward, existing TPM protocols do not suffice and must be adapted to the task of malware distribution. We clarify the capabilities of and countermeasures for this threat. Cloaking does not make malware all-powerful, and engineering malware to take advantage of a cloaked environment is a design challenge. A cloaked computation runs without OS support, so it cannot make a system call or easily use devices like a NIC for network communication. This paper also discusses best practices for TPM-enabled systems that can prevent the class of attacks we present.

This paper makes the following contributions.

- It specifies a protocol that runs on current TPM implementations that allows a malware developer to execute code in an environment that is guaranteed to be not externally observable, e.g., by a malware analyst. Our protocol adapts TPM-based remote attestation for use by the malware distribution platform.
- It presents the model of cloaked execution and measures the implementation of a malware distribution protocol that uses the TPM to cloak its computation.
- It provides several real-world use cases for TPM-based malware cloaking, and describes how to adapt malware to use TPM cloaking for those cases. These include: worm command and control, selective data exfiltration, and a DDoS timebomb.
- It discusses various defenses against our attacks and their tradeoffs with TPM security and usability.

Organization In Section 2 we describe our threat model and different attack scenarios for TPM cloaked malware. Then in Section 3 we give TPM background information. We then describe and analyze a general TPM cloaked malware attack in Section 4 and follow with a description of a prototype implementation in Section 5.

We then turn to discussing future defenses against such attacks in Section 6; describe related work in Section 7 and finally conclude in Section 8.

2 Threat Model and Attack Scenarios

We begin by describing our threat model for an attacker that wishes to use the TPM for cloaked computations. Then we describe multiple attack scenarios that can leverage TPM cloaked computations.

2.1 Threat model and goals

We consider an **attacker** who wishes to infect machines with malware. His goal is to make a portion of this malware unobservable to any **analyst** (e.g., white-hat security researcher, or IT professional) except for its input and output behavior.

We assume an attacker will have the following capabilities on the compromised machine.

- **Kernel-level compromise.** We assume our attack has full access to the OS address space. Late launch computation is privileged and can only be started by code that runs at the OS privilege level. Exploits that result in kernel-level privileges for commodity OSes are common enough to be a significant concern. For example, in September and October 2010, there were 13 remote code execution vulnerabilities and 2 privilege escalation vulnerabilities that could provide a kernel-level exploit for Microsoft's Windows 7 [13]. Kernel-level exploits for Linux are reported more rarely, but do exist, e.g., the recent Xorg memory management vulnerability [54]. There are many examples of malware using kernel vulnerabilities [34, 3].
- **Authorization for TPM capabilities.** We further assume our attack can authorize the TPM commands in our protocol. TPM commands are authorized using **AuthData**, which are 160-bit secrets that will be described further in Section 3. The difficulty of obtaining AuthData depends on how TPMs are used in practice. To our knowledge, the TCG does not provide concrete practices for protecting AuthData. Most TPM commands do not require AuthData to be sent on wire, even in encrypted form. However, knowing AuthData is necessary for certain common TPM operations like using TPM-controlled encryption keys. We discuss acquiring the AuthData needed by the attack in Sections 3.6 and 4.

An analyst will see all non-blackbox behavior of the attacker's cloaked computation. In our model, the analyst is allowed full access to systems that run our malware. We assume that all network traffic is visible, and that the analyst will attempt to exploit any attack protocol weaknesses. In particular, an analyst might run a honeypot that is intended to be infected so that he can observe and analyze the malware. A honeypot may use a virtual machine (including those that use hardware support for virtualization like VMWare Workstation and KVM [33]), and may include any combination of emulated and real hardware, including software-based TPM emulators [50] and VM interfaces to hardware TPMs like that of vTPMs [17].

We assume the analyst is neither able to mount physical attacks on the TPM nor is able to compromise the TPM public key infrastructure. (We revisit these assumptions when discussing possible defenses in Section 6.) While there are known attacks against Intel's late launch environment [55] and physical attacks against the TPM [51, 32], manufacturers are working to eliminate such attacks. Manufacturers have significant incentive to defeat these attacks because they compromise the TPM's guarantee that is currently its most commercially important: preventing data leakage from laptop theft.

Our attack may be detectable because it increases TPM use. Nonetheless, frequent TPM use might be the norm for some systems, or users and monitoring tools may simply be unaware that increased TPM use is a concern.

A cloaked computation is limited to a computational kernel. It cannot access OS services or make a system call. Any functional malware must have extensive support code beyond the cloaked computation. The support code performs tasks like communication over the network or access to files. The attacker must design malware to split functionality into cloaked and observable pieces. Arguments can be passed to the computational kernel via memory, and may be encrypted or signed off-platform for privacy or integrity.

2.2 Attack Scenarios

We now describe various attack scenarios that leverage TPM cloaking.

2.2.1 Worm command and control

We consider a modification of the Conficker B worm. The worm has an infection stage, where a host is exploited and downloads command and control code. Then the infection code runs a rendezvous protocol to download and execute signed binary updates. Engineers halted the propagation of Conficker B by reverse engineering the rendezvous protocol and preventing the registration of domain names that Conficker was going to generate.

Defeating Conficker requires learning in advance the rendezvous domain names it will generate. The sequence of domain names can be determined in two ways; first by directly analyzing the domain name generation implementation or second by running the algorithm with inputs that will generate future domain names. Cloaked computation prevents the static analysis and dynamic emulation required to reverse engineer binary code, eliminating the first option of analyzing the implementation.

Conficker uses as input to its domain name generation algorithm the current day (in UTC). It establishes the current day by fetching data from a variety of web sites. White hat researchers ran Conficker with fake replies to these http requests, tricking the virus into believing it was executing in the future.

However, malware can obtain timestamps securely at day-level granularity. Package repositories for common Linux distributions provide descriptions of repository contents that are signed, include the date, and are updated daily. (See `http://us.archive.ubuntu.com/ubuntu/dists/lucid-updates/Release` for Ubuntu Linux, which has an accompanying ".gpg" signature file.) This data is mirrored at many locations worldwide and is critical for the integrity of package distribution[1], so taking it offline or forging timestamps would be both difficult and a security risk.

Conficker is not alone in its use of domain name generation for rendezvous points. The Mebroot rootkit [31] and Kraken botnet [5] both use similar techniques to contact their command and control servers.

[1]Although individual packages are signed, without signed release metadata a user may not know whether there is a pending update for a package.

Using cloaked computations for malware command and control does not *ipso facto* make malware more dangerous. Cloaked computations must be used as part of a careful protocol in order to be effective.

2.2.2 Selective data exfiltration

An infection program can exfiltrate private financial data or corporate secrets. To minimize the probability of detection, the program rate limits its exfiltrated data. The program searches and prioritizes data inside a cloaked computation, perhaps using a set of regular expressions.

Cloaked computation can obscure valuable clues about the origin and motivation of the infection authors. The regular expressions might target information about a particular competitor or project. If white hats can sample the exfiltrated data, this would also provide clues; however, it would give less direct evidence than a set of search terms, and output could be encrypted.

Stuxnet and Aurora are recent high profile attacks that exfiltrate data [38]. Stuxnet seeks out specific industrial systems and sends information about the infected OS and domain information to one of two command servers [26]. A program without cloaked computation could use cryptographic techniques [59, 18, 28] to keep search criteria secret while being observed in memory, but their performance currently makes them impractical.

2.2.3 Distributed denial-of-service timebomb

A common malware objective is to attack a target at a certain point in time. Keeping the time and target secret until the attack prevents countermeasures to reduce the attack's impact. A cloaked computation can securely check the day (as above), and only make the target known on the launch day.

Malware analysis has often been important for stopping distributed denial-of-service (DDoS) attacks. One prominent example is MyDoom A. MyDoom A was first identified on January 26, 2004 [2]. The worm caused infected computers to perform a DDoS on `www.sco.com` on February 1, 2004, less than a week after the virus was first classified. However, the worm was an easy target for analysts because its target was in the binary obscured only by ROT-13 [1]. Since the target was identified prior to when the attack was scheduled, SCO was able to remove its domain name from DNS before a DDoS occurred [57].

The Storm worm's targeting of `www.microsoft.com` [46], Blaster's targeting of `windowsupdate.com` [34], and Code Red's targeting of `www.whitehouse.gov` [22] are other prominent examples of DDoS timebombs whose effects were lessened by learning the target in advance of the attack. If timebomb logic is contained in cloaked code, then it increases the difficulty of detecting the time and target of an upcoming attack. Since the target is stored only in encrypted form locally on infected machines, the infected machines do not have to communicate over the network to receive the target at the time of the attack.

Not every machine participating in a DDoS coordinated by cloaked computation must have a TPM. A one-million machine botnet could be coordinated by one-thousand machines with TPMs (to pick arbitrary numbers). The TPM-containing machines would repeatedly execute a cloaked computation, as above, to determine when to begin an attack. These machines would send the target to the rest when they detect it is time to begin the DDoS. In the example, all million machines must receive the DDoS target, but the topology of communication is specialized to the DDoS task and therefore is more difficult to filter and less amenable to traffic analysis than a generic peer-to-peer system.

3 TPM background

This section describes the TPM hardware and support software in sufficient detail to understand how it can be used to make malware more difficult to analyze.

3.1 TPM hardware

TPMs are usually found in x86 PCs as small integrated circuits on motherboards that connect to the low pin count (LPC) bus and ultimately the southbridge of the PC chipset. Each TPM contains an RSA (public-key) cryptography unit and platform configuration registers (PCRs) that maintain cryptographic hashes (called measurements by the TCG) of code and data that has run on the platform.

The goal of the TPM is to provide security-critical functions like secure storage and attestation of platform state and identity. Each TPM is shipped with a public encryption key pair, called the Endorsement Key (**EK**), that is accompanied by a certificate from the manufacturer. This key is used for critical TPM management tasks, like "taking ownership" of the TPM, which is a form of initialization. During initialization the TPM creates a secret, $tpmProof$, that is used to protect keypairs it creates.

The TPM 1.2 specification requires PC TPMs to have at least 24 PCRs. PCRs 0–7 measure core system components like BIOS code, PCRs 8–15 are OS defined, and PCRs 16–23 are used by the TPM's late launch mechanism, where sensitive software runs in complete hardware isolation (no other program, including the OS, may run concurrently unless specifically allowed by the software). PCRs cannot be set directly, they can only be **extended** with new values, which sets a PCR so that it depends on its previous value and the extending value in a way that is not easily reversible. PCR state can establish what software has been run on the machine since boot, including the BIOS, hypervisor and operating system.

3.2 Managing and protecting TPM storage

The TPM was designed with very little persistent storage to reduce cost. The PC TPM specification only mandates 1,280 bytes of non-volatile RAM (NVRAM), so most data that the TPM uses must be stored elsewhere, like in main memory or on disk. When we refer to an object as **stored in the TPM**, we mean an object stored externally to the TPM that is encrypted with a key managed by the TPM. By contrast, data stored in locations physically internal to the TPM is **stored internal to the TPM**.

Concatenation of A and B	$A \,\|\|\, B$
Public/private keypair for asymmetric encryption named $name$	$(PK_{name}, SK_{name}) \equiv (PK, SK)_{name}$
Encryption of $data$ with a public key	Enc$(PK, data)$
Signing of $data$ with a signing key	Sign$(SK, data)$
Symmetric key	K (no P or S at front)
Symmetric encryption of $data$	EncSym$(K, data)$
One-way hash (SHA-1) of $data$	$H(data)$

Table 1: Notation for TPM data and computations.

AuthData controls TPM capabilities, which are the ability to read, write, and use objects stored in the TPM and execute TPM commands. AuthData is a 160-bit secret, and knowledge of the AuthData for a particular capability is demonstrated by using it as a key for calculating a hash-based message authentication code (HMAC) of the input arguments to the TPM command. [2]

Public signature and encryption key pairs created by a TPM are stored as **key blobs** only usable with a particular TPM. The contents of a key blob are shown in Figure 2. A hash of the public portion of a key blob is stored in the private portion, along with $tpmProof$ (mentioned above); $tpmProof$ is an AuthData value randomly generated by the TPM and stored internally to the TPM when someone takes ownership. It protects the key blob from forgery by adversaries and even the TPM manufacturer.[3]

In addition, a TPM user can use the PCRs to restrict use of TPM-generated keypairs to particular pieces of software that are identified via a hash of their code and initial data. For example, the TPM can configure a key blob so that it can only be used when the PCRs have certain values (and therefore only when certain software is running).[4]

[2]Since AuthData is used as an HMAC key, it does not need to be present on the same machine as the TPM for it to be used. For example, a remote administrator might hold certain AuthData and use this to HMAC input arguments and then send these across a network to the machine containing the TPM. However, AuthData does need to be in memory (and encrypted) when the secret is first established for a TPM capability as part of a TPM initialization protocol. We investigate further the implications of this nuance in our discussions of defenses in Section 6.

[3]Migratable keys are handled somewhat differently, but they are beyond the scope of this paper.

[4]Restricting a TPM-generated key to use with certain PCR values is not the same as the `TPM_Seal` command found in related literature. The two are similar, but the former places restrictions on a key's use, while the later places restrictions on the decryption of a piece of data (which could be a key blob).

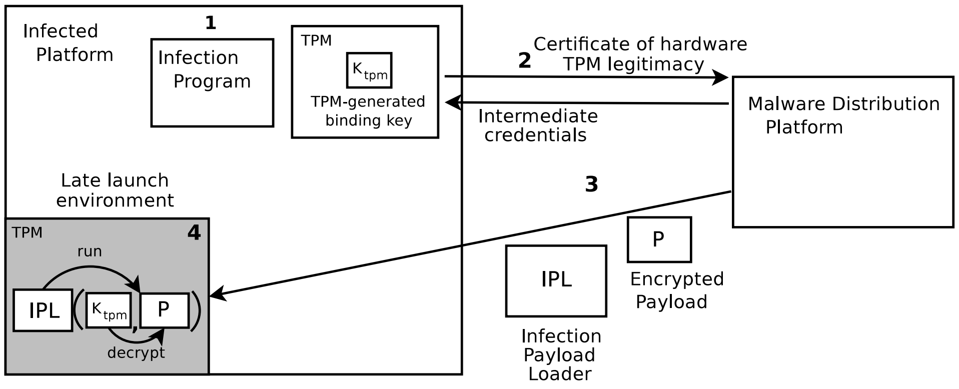

Figure 1: The overall flow of the attack is 1) Infecting a system with local malware capable of kernel-level exploitation to coordinate the attack 2) Establishing a legitimate TPM-generated key usable only by the Infection Payload Loader in late launch via a multistep protocol with a Malware Distribution Platform 3) Delivering a payload that can be decrypted using the TPM-generated key 4) Using a late launch environment to decrypt the payload with the TPM-generated key, and running it with inputs passed into memory by local malware 5) Retrieving output from payload, potentially repeating step 4 with new inputs. Boxes with "TPM" indicate parts of the protocol that use the TPM.

$$\begin{aligned}
\mathrm{Blob}((PK,SK)_{ex}) &\equiv \mathrm{PubBlob}((PK,SK)_{ex})\,||\,\mathrm{Enc}(PK_{parent},\mathrm{PrivBlob}((PK,SK)_{ex}))\\
\mathrm{PubBlob}((PK,SK)_{ex}) &\equiv PK_{ex}\,||\,\text{PCR values}\\
\mathrm{PrivBlob}((PK,SK)_{ex}) &\equiv SK_{ex}\,||\,H(\mathrm{PubBlob}((PK,SK)_{ex})\,||\,tpmProof
\end{aligned}$$

Figure 2: Contents of TPM key blob for an example public/private key pair named ex that is stored in the key hierarchy under a key named $parent$. For our purposes the parent key of most key blobs is the SRK. (Note that the PCR values themselves are not really stored in the key blob. Rather the blob contains a bitmask of the PCRs whose values must be verified and a digest of the PCR values.)

TPM key storage is a key hierarchy: a single-rooted tree whose root is the Storage Root Key (**SRK**), and is created upon the take ownership operation described below. The private part of the SRK is stored internal to the TPM and never present in main memory, even in encrypted form. Since the public part of the SRK encrypts the private part of descendant keys (and so on), all keys in the hierarchy are described as "stored in the TPM," even though all of them, except the SRK, are stored in main memory. Using the private part of any key in the hierarchy requires using the TPM to access the private SRK to decrypt private keys while descending the hierarchy.

It is impossible to use private keys for any of the keypairs stored in the TPM apart from using TPM capabilities: obtaining the private key for one key would entail decrypting the private portion of a key blob, which in turn requires the private key of the parent, and so on, up to the SRK, which is special in that its private key is never stored externally to the TPM (even in encrypted form). A TPM key hierarchy is illustrated in Figure 3.

3.3 Initializing the TPM

To begin using a TPM, the user (or administrator) must first take ownership of it. Taking ownership of the TPM establishes three important AuthData values: the owner AuthData value, which is needed to set TPM policy, the SRK AuthData value, which is needed to use the SRK, and $tpmProof$. $tpmProof$ is generated internal to the TPM and stored in NVRAM. It is never present in unencrypted form outside the TPM.

While it is easy for a professional administrator to take ownership of a TPM securely, taking ownership of a TPM is a security critical operation that is exposed in a very unfriendly way to average users. For example, Microsoft's BitLocker full-disk encryption software uses the TPM. When a user initializes BitLocker, it reboots the machine into a BIOS-level prompt where the user is presented cryptic messages about TPM initialization. BitLocker performs the initial ownership of the TPM, and it acquires privilege to do so with TPM mechanisms for asserting physical presence at the platform via the BIOS. An inexperienced user could probably be convinced to agree to allow assertion of physical presence by malware similar to how rogue programs convince users to install malicious software and input their credit card numbers [44]. The function of the TPM is complicated and flexible, making a simple explanation of it for an average user a real challenge.

Furthermore, malware could also gain use of physical presence controls in BIOS by attacks that modify

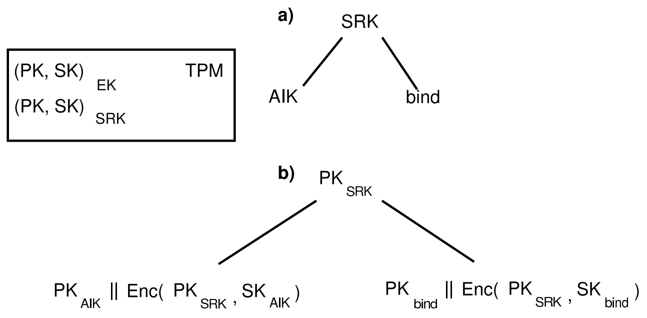
TPM
(PK, SK) EK
(PK, SK) SRK
a)
SRK
AIK
bind
b)
PK SRK
PK AIK || Enc(PK SRK , SK AIK)
PK bind || Enc(PK SRK , SK bind)

Figure 3: The part of the TPM key hierarchy relevant to our attack. The TPM box illustrates keying material stored internal to the TPM, which is only the endorsement key (EK) and storage root key (SRK). Part (a) shows the conceptual key hierarchy, while part (b) shows how the secret keys of children are encrypted by the public keys of their parents so keys can be safely stored in memory. More detail on key formats is found in Figure 2.

the BIOS itself [48]. Recent work has even demonstrated attacks against BIOS update mechanisms that require signed updates [56].

3.4 Platform identity

TPMs provide software attestation, a proof of what software is running on a platform when the TPM is invoked. The proof is given by a certificate for the current PCR values, which contain hashes of the initial state of all software run on the machine. This certificate proves to another party that a TPM-including platform is running particular software. The receiver must be able to verify that the certificate comes from a legitimate TPM, or the quoted measurements or other attestations are meaningless.

A user desiring privacy cannot directly use her platform's EK for attestation. (EKs are linked to specific platforms, and additionally multiple EK uses can be correlated.) Instead, she can generate attestation identity keys (**AIK**s) that serve as proxies for the EK. An AIK can sign PCR contents to attest to platform state. However, something must associate the AIK with the EK.

A trusted privacy certificate authority (Privacy CA) provides certificates to third parties that an AIK corresponds to an EK of a legitimate TPM. While prototype Privacy CA code exists [27], Privacy CAs appear to be unused in practice. In our attack, the malware distributor acts as a Privacy CA and only trusts AIKs that it certifies.

We emphasize that our proposed attack does not require or benefit from the anonymity guarantees provided by a Privacy CA. However, the TPM does not permit a user to directly sign an arbitrary TPM-generated public key with the EK, so our attack must use an intermediate AIK.

3.5 Using the TPM

Typical uses of the TPM are to manipulate the key hierarchy, to obtain signed certificates of PCR contents or of authenticity of TPM data, and to modify PCRs to describe platform state as it changes. Keys are created in the key hierarchy by "loading" a parent key and commanding the TPM to generate a key below that parent, resulting in a new key blob. Loading a key entails passing a key blob to the TPM to obtain a key handle, which is an integer index into the currently loaded keys. Only loaded keys can be used for further TPM commands. Loading a key requires loading all keys above it in the hierarchy, so loading any key in the key hierarchy requires loading the SRK.

The TPM can produce signed certificates of key authenticity. To do so, a user specifies a certifying key, and the TPM produces a hash of the public key for the key to be certified, along with a hash of a bitmask describing the selected PCRs and those PCR values, and signs both hashes with the selected key.

PCRs can be modified by the TPM as platform state changes. They cannot be set directly, and are instead modified by extension. A PCR with value PCR extended by a 160-bit value val is set to value $\mathrm{Extend}(PCR, val) \equiv H(PCR\,||\,val)$. Late launch extends the PCRs with the hash of the state of the program run in the late launch environment. Thus the TPM can restrict access to keys to a particular program. Our malware protocol uses this ability to prevent analyst use of a payload decryption key.

3.6 TPM functionality evolving and best practices unknown

Despite the widespread availability of trusted computing technology as embodied by the TPM, its implications are not well understood. The specification for the TPM and supporting software is complicated; version 1.2 of the TPM specification for the PC/BIOS platform with accompanying TCG Software Stack is over 1,500 pages [52]. Additionally, there are few guidelines for proper use of its extensive feature set. It is quite believable that such a complicated mechanism has unintended consequences that undermine its security goals. In this paper, we propose such a mechanism: that the TPM can be used as a means to thwart analysis of malware.

Key hierarchy The lack of guidance on the usage of TPM capabilities makes it difficult to determine what information an attacker might reasonably acquire. For example, the key hierarchy has a single root. Therefore, different users must share at least one key, and every use of a TPM key requires loading the SRK. Loading the SRK requires SRK AuthData, and thus the SRK AuthData is likely well-known, making it possible for users to impersonate the TPM, as others have previously indicated [21].

EK certificates As another example of capabilities in flux, EK certificates critical to identifying TPMs as legitimate are not always present, and it is not always clear how to verify those that are. TPM manufacturers are moving toward certifying TPMs as legitimate by including certificates for EKs in TPM NVRAM. Infineon gives the most detail on their EK certification policy, in which the certificate chain extends back to a new VeriSign TPM root Certificate Authority [11]. ST Microelectronics supplies TPMs used in many workstations from Dell. They state that their TPMs from 2010 onward contain certificates [9]. While no certificates were present on our older machines, we did find certificates for our newer Dell machines and manually verified the legitimacy of the EK certificate for one of our TPMs (which we describe further in Section 5).

Protecting AuthData Many uses of the TPM allow AuthData to be snooped if not used carefully. For example, standard use of TPM tools with TrouSerS prompts the user to enter passwords at the keyboard to use TPM capabilities. These passwords can be captured by a keylogger if the system is compromised. Thus, despite that TPM commands may not require AuthData to appear, entry of this data into the system for usage can be insecure.

4 Malware using cloaked computations

We now describe an architecture and protocol for launching a TPM-cloaked attack.

Our protocol runs between an **Infection Program**, which is malware on the attacked host, and a **Malware Distribution Platform**, which is software executed on hardware that is remote to the attacked host. The goal of the protocol is for the Infection Program to generate a key. The Infection Program attests to the Malware Distribution Platform that TPM-based protection ensures only it can access data encrypted with the key. The Malware Distribution Platform verifies the attestation, and then sends an encrypted program to the Infection Program. The Infection Program decrypts and executes this payload. This protocol enables long-lived and pernicious malware, for example, turning a computer into a botnet member. The Infection Program can suspend the OS (and all other software) through use of processor late launch capabilities to ensure unobservability when necessary, like when the malicious payload is decrypted and executing.

4.1 Late launch for secure execution

The protocol uses late launch to suspend the OS to allow decryption and execution of the malicious payload without observation by an analyst. Late launch creates an execution environment where it is possible to keep code and data secret from the OS.

Late launch transfers control to a designated block of user-supplied code in memory and leaves a hash of that code in TPM PCRs. Specifically, with Intel's Trusted Execution Technology, a user configures data structures to describe the Measured Launch Environment (**MLE**), the program to be run (which resides completely in memory). She then uses the `GETSEC[SENTER]` instruction to transfer control to chipset-specific code, signed by Intel, called `SINIT` that performs pre-MLE setup such as ensuring correctness of MLE parameters. The exact functionality of `SINIT` is not known, as its source code is not public. `SINIT` then passes control to the MLE. When the MLE runs, no software may run on any other processor and hardware interrupts and DMA transfers are disabled. To exit, the MLE uses the `GETSEC[SEXIT]` instruction.

Pack$(data, extra, PK)$:

1. Generate symmetric key K
2. Asymmetric encrypt K to form $\mathrm{Enc}(PK, K \,||\, extra)$
3. Symmetric encrypt $data$ to form $\mathrm{EncSym}(K, data)$
4. Output $\mathrm{EncSym}(K, data) \,||\, \mathrm{Enc}(PK, K \,||\, extra)$

Unpack$(\mathrm{EncSym}(K, data) \,||\, \mathrm{Enc}(PK, K \,||\, extra), SK)$:

1. Asymmetric decrypt $\mathrm{Enc}(PK, K \,||\, extra)$ with SK to obtain K and $extra$
2. Symmetric decrypt $\mathrm{EncSym}(K, data)$ with K to obtain $data$
3. Output $data, extra$

Figure 4: Subroutines used in main protocol. $extra$ is needed for `TPM_ActivateIdentity`, and can be empty (ϕ). Running Unpack on the TPM uses `TPM_Unbind`.

4.2 Malware distribution protocol

The Infection Program first establishes a proof that it is using a legitimate TPM. It uses the TPM to generate two keys. One is a "binding key" that the Malware Distribution Platform will use to encrypt the malicious payload. The other is an AIK that the TPM will use in the Privacy CA protocol, where the Malware Distribution Platform plays the role of the Privacy CA. The Malware Distribution Platform will accept its own certification that the AIK is legitimate in a later phase. As stated before, the Privacy CA protocol enables indirect use of the private EK only kept by the TPM. A valid private EK cannot be produced by an analyst; it is generated by a TPM manufacturer and only accessible to the TPM hardware. This part of the Infection Program is named "Infection Keygen".

Our description of the protocol steps will elide lower-level TPM authorization commands like `TPM_OIAP` and `TPM_OSAP` that are used to demonstrate knowledge of authorization data and prevent replay attacks on TPM commands.

We use subroutines $\mathrm{Pack}(data, extra, PK)$ and $\mathrm{Unpack}(data, PK)$, which use asymmetric keys with intermediate symmetric keys. Symmetric keys increase the efficiency of encryption, are required for certain TPM commands, and circumvent the limits (due to packing mechanisms) on the length of asymmetrically encrypted messages. These subroutines are shown in Figure 4 and the main protocol is in Figure 5.

4.3 Analyzing the resilience of the protocol

A malware analyst can attempt to subvert the protocol by tampering with data or introducing keys under her control. We now analyze the possibilities for subversion.

Infection Keygen: Generate binding key that Malware Distribution Platform will eventually use to encrypt malicious payload, AIK that certifies it, and request for Malware Distribution Platform to test AIK legitimacy

1. Create binding keypair $(PK, SK)_{bind}$ under the SRK with `TPM_CreateWrapKey`(SRK, $PCR18 = \text{Extend}(0_{160}, H($**Infection Payload Loader**$)))$ (requires SRK AuthData), store in memory
2. Create identity key $(PK, SK)_{AIK}$ under SRK in memory as $\text{Blob}((PK, SK)_{AIK})$ with `TPM_MakeIdentity` (requires owner AuthData)
3. Retrieve EK certificate $C_{EK} = PK_{EK} \,||\, \text{Sign}(SK_{manufacturer}, H(PK_{EK}))$, which certifies that the TPM with that EK is legitimate (requires owner AuthData to obtain from NVRAM with `TPM_NV_ReadValue` from EK index or needs to be on disk already)
4. Send $M_{req} \equiv \text{PubBlob}((PK, SK)_{AIK}) \,||\, C_{EK}$ to Malware Distribution Platform as a request to link AIK and EK

Malware Distribution Platform Certificate Handler: Give Infected Platform credential only decryptable by legitimate TPM

1. Receive M_{req}
2. Verify $\text{Sign}(SK_{manufacturer}, H(PK_{EK}))$ with manufacturer CA public key
3. Generate hash $H_{aik\_cert} \equiv H(\text{PubBlob}((PK, SK)_{AIK}))$
4. Sign H_{aik_cert} with $SK_{malware}$, a private key known only to the Malware Distribution Platform whose corresponding public key is known to all, to form $\text{Sign}(SK_{malware}, H_{aik\_cert})$. $\text{Sign}(SK_{malware}, H_{aik\_cert})$ is a credential of AIK legitimacy.
5. Run $\text{Pack}(\text{Sign}(SK_{malware}, H_{aik\_cert}), H_{aik\_cert}, PK_{EK})$ to form $M_{req\_resp} \equiv \text{Enc}(PK_{EK}, K_2 \,||\, H_{aik\_cert}) \,||\, \text{EncSym}(K_2, \text{Sign}(SK_{malware}, H_{aik\_cert}))$. M_{req_resp} contains the credential in a way such that it can only be extracted by a TPM with private EK SK_{EK} when the credential was created for an AIK stored in that TPM.
6. Send M_{req_resp} to Infected Platform

Infection Proof: Decrypt credential, assemble certificate chain from manufacturer certified EK to binding key (including credential)

1. Receive M_{req_resp}
2. Load AIK $(PK, SK)_{AIK}$ and binding key $(PK, SK)_{bind}$ with `TPM_LoadKey2`
3. Use `TPM_ActivateIdentity`, which decrypts $\text{Enc}(PK_{EK}, K_2 \,||\, H_{aik\_cert})$ and retrieves K_2 after comparing H_{aik_cert} to that calculated from loaded AIK located in internal TPM RAM. If comparison fails, abort. (requires owner AuthData)
4. Symmetric decrypt $\text{EncSym}(K_2, \text{Sign}(SK_{malware}, H_{aik\_cert}))$ to retrieve $\text{Sign}(SK_{malware}, H_{aik\_cert})$
5. Certify $(PK, SK)_{bind}$ with `TPM_CertifyKey` to produce $\text{Sign}(SK_{AIK}, H(PCRs(\text{PubBlob}((PK, SK)_{bind}))) \,||\, H(PK_{bind})) \equiv \text{Sign}(SK_{AIK}, H_{bind\_cert})$
6. Send $M_{proof} \equiv \text{Sign}(SK_{malware}, H_{aik\_cert}) \,||\, \text{PubBlob}((PK, SK)_{AIK}) \,||\, \text{Sign}(SK_{AIK}, H_{bind\_cert}) \,||\, \text{PubBlob}((PK, SK)_{bind})$, all the evidence needed to verify TPM legitimacy, to Malware Distribution Platform

Malware Distribution Platform Payload Delivery: Verify certificate chain, respond with encrypted malicious payload if successful

1. Receive M_{proof}
2. Verify signatures of H_{aik_cert} by $SK_{malware}$ using $PK_{malware}$, of H_{bind_cert} using PK_{AIK}. Check that H_{bind_cert} corresponds to the binding key by comparing hash of public key, PCRs to $\text{PubBlob}((PK, SK)_{bind})$. Use $\text{PubBlob}((PK, SK)_{bind})$ to determine if binding key has a proper constraint for $PCR18$. Abort if verification fails or binding key improperly locked.
3. Hash and sign the payload with $SK_{malware}$ to form $\text{Sign}(SK_{malware}, H(payload))$ (only needs to be done once per payload)
4. Run $\text{Pack}(payload \,||\, \text{Sign}(SK_{malware}, H(payload)), \phi, PK_{bind})$ to form $M_{payload} \equiv \text{EncSym}(K_3, payload \,||\, \text{Sign}(SK_{malware}, H(payload))) \,||\, \text{Enc}(PK_{bind}, K_3)$
5. Send $M_{payload}$ to Infected Platform

Infection Payload Execute: Use late launch to set PCRs to allow use of binding key for decryption and to prevent OS from accessing this key during use

1. Receive $M_{payload}$
2. Late launch with MLE $\equiv$ **Infection Payload Loader**

Infection Hidden Execute: Infection Payload Loader decrypts and executes the payload in the late launch environment.

1. Load $(PK, SK)_{bind}$ with `TPM_LoadKey2`
2. Run $\text{Unpack}(M_{payload}, SK_{bind})$. This operation can succeed (and only in this program) because in **Infection Hidden Execute**, $PCR18 = \text{Extend}(0_{160}, H($**Infection Payload Loader**$))$. Obtain $payload \,||\, \text{Sign}(SK_{malware}, H(payload))$.
3. Verify signature $\text{Sign}(SK_{malware}, H(payload))$ with $PK_{malware}$. Abort if verification fails.
4. Execute $payload$
5. If return to OS execution is desired, scrub payload from memory and extend random value into PCR18, then exit late launch

Figure 5: The cloaked malware protocol.

key blob = `TPM_CreateWrapKey`(parent key, PCR constraints)	Generate new key with PCR constraints under the parent key in hierarchy. The resultant key may be used for encryption and decryption, but not signing.
key handle = `TPM_LoadKey2`(key blob)	Load a key for further use.
key blob = `TPM_MakeIdentity`()	Generate an identity key under SRK that may be used for signing, but not encryption and decryption.
sym_key = `TPM_ActivateIdentity`(identity key handle, CA response)	Verify that asymmetric CA response part corresponds to identity key. If agreement, decrypt response and retrieve enclosed symmetric key.
(certificate, signature) = `TPM_CertifyKey`(certifying key handle, key handle)	Produce certificate of key contents. Sign certificate with certifying key.
value = `TPM_NV_ReadValue`(index)	Retrieve data from TPM NVRAM.

Table 2: Additional functions in the main protocol. Keywords that are in fixed-width font that begin with `TPM_` are TPM commands defined in the TPM 1.2 specification.

The analyst's goal is to cause the malicious payload to be encrypted with a key under her control, or to observe a decrypted payload. She could try to create a binding key blob during **Infection Proof**, and certify it with a legitimate TPM. However, the analyst does not know the value of $tpmProof$ for any TPM because it is randomly generated within the TPM and is never present (even in encrypted form) outside the TPM. Without $tpmProof$, the analyst cannot generate a key blob that the TPM will certify, even under a legitimate AIK. This argument relies on the fact that the encryption system is non-malleable [25] and chosen ciphertext secure. Otherwise, an attacker might be able to take a legitimately created ciphertext with $tpmProof$ in it and modify it to an illegitimate ciphertext with $tpmProof$ in it, without knowing $tpmProof$.

The analyst could attempt to modify PCR constraints on the binding key by tampering with the the public part of the key. However, the TPM will not load the key in the modified blob because a digest of the public portion of the blob will not match the hash stored in the private portion. Thus, storing the binding key in the public part of the blob where it is accessible to the analyst does not compromise the security of the protocol. If the binding key is a legitimate TPM key with PCR constraints that do not lock it to being observed only during **Infection Hidden Execute**, the Malware Distribution Platform will detect it during **Malware Distribution Platform Payload Delivery**, and the platform will not encrypt the payload with that key.

The analyst could attempt to forge keys at other points in the hierarchy: she could attempt to certify a binding key she creates with an AIK that she creates. The Malware Distribution Platform only obtains the public portions of these key blobs, and so cannot check directly in **Malware Distribution Certificate Handler** that the AIK is legitimate. The Malware Distribution Platform could not verify the legitimacy of key blobs even with their private portions as the Platform can neither decrypt the private portions, nor know the value of $tpmProof$ for the Infected Platform. However, it encrypts with the EK a credential that is a signed hash of the AIK it is sent by **Infection Keygen** running on an infected platform. The EK is proven legitimate by a certificate of authenticity signed by the TPM manufacturer's private key and verified by the Malware Distribution Platform. The private EK is only stored internal to the TPM, and only usable under controlled circumstances like `TPM_ActivateIdentity`; to our knowledge, there is no way to compel the TPM to decrypt arbitrary data with the private EK. `TPM_ActivateIdentity` will only decrypt a public EK-encrypted blob of the form $(K \,||\, H_{aik\_cert})$ where H_{aik_cert} is the hash of the public portion of an AIK key blob where the AIK has been loaded into the TPM (and thus has not been tampered with). Therefore, K cannot be recovered for an illegitimate AIK, and the credential $\text{Sign}(SK_{malware}, H_{aik\_cert})$ cannot be recovered. Without this credential, the protocol will abort in **Malware Distribution Platform Payload Delivery** (step 2). The credential cannot be forged as it contains a signature with a private key known only by the Malware Distribution Platform.

The analyst could try to execute forged payloads with **Infection Hidden Execute** because the public binding key is visible. However, because **Infection Hidden Execute** will only execute payloads signed by a key unknown to the analyst, this will not work. No program other than **Infection Hidden Execute** and the programs it executes can access the binding key.

The analyst could try to set the PCR values to those specified in $(PK, SK)_{bind}$, but run a program other than Infection Payload Loader. This would allow her to decrypt the payload (step 2 in Infection Hidden Execute). The values of PCRs are affected by processor events and the `SINIT` code module. The CPU instruction `GETSEC[SENTER]` sends an LPC bus signal to initialize the dynamically resettable TPM PCRs (PCRs 16-23) to 160 bits of 0s. No other TPM capability can reset these PCRs to all 0s; a hardware reset sets them to all 1s. So an analyst can only set PCR 18 to all 0s with a late launch executable. SINIT extends PCR 18 with a hash of the MLE. Therefore, to set PCR 18, the analyst must run an MLE with the correct hash. Assuming the hash function is collision resistant, only the Infection Payload Loader will hash to the correct value, so the analyst cannot run an alternate program that passes the PCR check. The payload loader terminates at payload end by extending a random value into PCR 18, so the analyst cannot use the key after the late launch returns.

4.4 Prevention of malware analysis

Having described our protocol for cloaked malware execution, we review how it defeats conventional malware analysis. While our list of malware analysis techniques may not be exhaustive, to our knowledge, TPM cloaking can be defeated only by TPM manufacturer intervention, or by physical attacks, like direct monitoring of hardware events or tampering with the TPM or system buses. Both of these are discussed in more detail in Section 6.

Static analysis. Cloaked computations are encrypted and are only decrypted once the TPM has verified that the PCRs match those in the key blob. The malware author specifies PCR values that match only the Infection Payload Loader, so no analyst program can decrypt the code for a cloaked computation.

Honeypots. Honeypots are open systems that collect and observe malware, possibly using some combination of emulation, virtualization and instrumented software. Purely software-based honeypots can try to follow our protocol without using a legitimate hardware TPM, but will fail to convince a malware distributing machine of their authenticity. This failure is due to their inability to decrypt $\mathrm{Enc}(PK_{EK}, K_2 \,||\, H_{aik\_cert})$, which is encrypted with the public EK that is certified by a TPM manufacturer in C_{EK}, and the private part of which is not present outside of a TPM. Thus these honeypots will never receive the malicious payload. If a honeypot uses a legitimate hardware TPM, it will obtain a malicious payload. However, it can only execute the payload with late launch, which prevents software monitoring of the unencrypted payload.

Virtualization. Software-based TPMs, virtualized TPMs, and virtual machine monitors communicating with hardware TPMs cannot defeat cloaking. Hardware TPMs have certificates of authenticity that are verified in our malware distribution protocol. A software-based TPM either will not have a certificate, or will have a certificate that is distinguishable from a hardware TPM. Either way, it will fail to convince a malware distribution platform of its authenticity. An analyst cannot use a virtual machine to defeat cloaking.

Hardware TPM manufacturers should not certify software-based TPMs as authentic hardware TPMs. Software-based TPMs cannot provide the same security guarantees as hardware-based TPMs. The PCRs of software-based TPMs might not correspond to platform state in any way, as they can be modified by sufficiently privileged software. A software TPM cannot attest to a particular software environment, because it does not know the true software environment—it could be executing in a virtual environment. Any certificate for a software-based TPM must identify the TPM as software otherwise the chain of trust is broken, defeating remote attestation (a major purpose of TPMs). No TPM manufacturer currently signs software TPM EKs, nor (to our knowledge) do any plan to do so. Prior work on virtualizing TPMs emphasizes that virtual TPMs and their certificates must be distinguishable from hardware TPMs, as the two do not provide the same security guarantees [17]. A malware distribution platform can avoid software and virtual TPM certificates by using a whitelist of known-secure hardware TPM certificate distributors compiled into the malware.

Software, such as a virtual machine monitor, cannot communicate with a legitimate hardware TPM to obtain and decrypt the malicious payload without running the payload in late launch. The only way that the malicious payload can be decrypted is through use of a private key stored in the TPM that can only be used when the TPM PCRs are in a certain state. This state can only be achieved through late launch, which is a *non-virtualizable* function, and it prevents software monitoring of the unencrypted payload. TPM late-launch is designed to be non-virtualizable, so that TPM hardware can provide a complete and reliable description of platform state.

4.5 Attack assumptions

Like any attack, ours has particular assumptions. As discussed in Section 2.1, our protocol requires late launch instructions, which are privileged, so **Infection Hidden Execute** must run at kernel privilege levels.

More importantly, our attack requires knowledge of SRK and owner AuthData values. There are two main possibilities for acquiring this AuthData previously mentioned in Section 3: snooping and overriding with physical presence.

AuthData can be snooped from kernel or application (e.g. TrouSerS) memory or from logged keystrokes, which are converted into AuthData by a hash. The likelihood of successful AuthData snooping depends on the particular AuthData being gathered. The SRK must be loaded to load any other key stored in the TPM, so there will be regularly occurring chances to snoop the SRK AuthData. Owner AuthData, on the other hand, is required for fewer, and generally more powerful, operations. It is then liable to be more difficult to acquire.

One could enter all AuthData remotely to a platform that contains a TPM, but we consider it unlikely that this is done in practice. TPM arguments could be HMACed by a trusted server, but such a server can become a performance or availability bottleneck. Use of a trusted server is also problematic for use of laptops that may not always have network connectivity. For these cases, it may be possible to enter AuthData into a separate trusted device that then can assist in authorizing TPM commands. However, such devices are currently not deployed. It is currently more likely that AuthData would be presented through a USB key or entered at the keyboard, and in both cases it can be snooped. In addition, applications and OS services used to provide AuthData to the TPM may not sufficiently scrub sensitive data from memory.

To demonstrate the possibility of acquiring AuthData from the OS, we virtualized a Windows 7 instance, and used OS-provided control panels to interact with the TPM. When AuthData was read from a removable drive, it remained in memory for long periods of time on an idle system, even after the relevant control panels were closed. The entire contents of the file containing the AuthData were present in memory for up to 4 hours after the AuthData was read, and the removable drive ejected from the system. The AuthData itself remained in memory for several days, before the system was eventually shut down.

If malware can use mechanisms for asserting physical presence at the platform, it can clear the current TPM owner and install a new owner, preventing the need to snoop any AuthData. While physical presence mechanisms should be tightly controlled, their implementation is left up to TPM and BIOS manufacturers. Our experience setting up BitLocker (see Section 3.3) indicates that the process can be confusing, and that it may be possible to convince a user to enable malware to obtain the necessary authorization to use TPM commands.

4.6 Distributing the malware distribution platform

As written, the malware distribution platform consists of a host (or small number of hosts) controlled by the attacker and trusted with the attacker's secret key ($SK_{malware}$). This design creates a single point of failure.

The Malware Distribution Platform computation consists of arithmetic and cryptographic work (with no OS involvement) with an embedded secret. It is a perfect candidate to run as a cloaked computation. An attacker can distribute work done on the Malware Distribution Platform to compromised hosts using cloaked computations.

5 Implementation and Evaluation

We implemented a prototype of our attack, which contains implementations of the establishment of a TPM-controlled binding key, the decryption and execution of payloads in late launch, and sample attack payloads. In this section, we describe each of these pieces in turn.

The prototype implementation consists of five programs for the key establishment protocol (described in Table 3), the Infection Payload Loader PAL and ported TrouSerS TPM utility code, payload programs, and supporting code to connect the pieces. The key establishment programs are about 3,600 lines of C, the Infection Payload Loader is another 400 lines of C, with another 150 lines of C added to provide TPM commands through selections of TrouSerS TPM code which themselves required minor modifications. The payloads were about 50 lines apiece with an extra 75 line supporting DSA routine, which was necessary for verifying Ubuntu's repository manifests. All code size measurements are as measured by SLOCCount [53].

5.1 Binding key establishment

We implemented a prototype of the protocol described in Figure 5 using the TrouSerS [6] (v0.3.6) implementation of the TCG software stack (TSS) to ease development.

Our implementation follows the protocol, except steps 2 to 3 in **Infection Keygen** which use TSS API call `Tspi_CollateIdentityRequest`. This call does not produce M_{req} (step 4), but instead produces $\mathrm{EncSym}(K, \mathrm{PubBlob}((PK, SK)_{AIK}))$ and $\mathrm{Enc}(PK_{malware}, K)$ that must be decrypted in the Malware Distribution Platform Certificate Handler. While the protocol specifies network communication, the prototype communicates via files on one machine. TrouSerS is not necessary for malware cloaking; TPM commands made by TrouSerS could be made directly by malware.

5.1.1 EK certificate verification

We verified the authenticity of our ST Microelectronics TPM endorsement key (EK). However, we had to overcome obstacles along the way, and there may be obstacles with other TPM manufacturers as well. For example, we needed to work around unexpected errors in reading the EK certificate from TPM NVRAM. Reads greater than or equal to 863 bytes in length return errors, even though the reads seem compatible with the TPM specification, and the EK certificate is 1129 bytes long. We read the certificate with multiple reads, each smaller than 863 bytes.

The intermediate certificates in the chain linking the TPM to a trusted certificate authority were not available online, and we obtained them from ST Microelectronics directly. However, some manufacturers (e.g. Infineon) make the certificates in their chains available online [11]. To deploy TPM-based cloaking on a large scale, the verification process for a variety of TPMs should be tested.

For the TPM we tested, the certificate chain was of length four including the TPM EK certificate and rooted at the GlobalSign Trusted Computing Certificate Authority. There were two levels of certificates within ST Microelectronics: Intermediate EK CA 01 (indicating there are likely more intermediate CAs) and a Root EK CA.

5.2 Late launch environment establishment

We modified code from the Flicker [40] (v0.2) distribution to implement our late launch capabilities. Flicker provides a kernel module that allows a small self-contained program, known as a Piece of Application Logic or **PAL**, to be started in late launch with a desired set of parameters as inputs in physical memory. The kernel module accepts a PAL and parameters through a `sysfs` filesystem interface in Linux, then saves processor context before performing a late launch, running the PAL in late launch, and then restoring the processor context after the PAL completes. Output from PALs is available through the filesystem interface when processor context is restored.

We implemented the Infection Payload Loader as a PAL, which takes the encrypted and signed payload, the symmetric key used to encrypt the payload encrypted with the binding key, and the binding key blob as parameters. We used the PolarSSL [15] embedded cryptographic library for all our cryptographic primitives (AES encryption, RSA encryption and signing, SHA-1 hashing and SHA-1 HMACs).

We ported code from TrouSerS to handle use of TPM capabilities that were not implemented by the Flicker TPM library (`TPM_OIAP`, `TPM_LoadKey2`, `TPM_Unbind`). We replaced the TrouSerS code dependence on OpenSSL with PolarSSL. We fixed two small bugs in Flicker's TPM driver that seem to be absent from the recent 0.5 release due to use of an alternate driver.

5.3 Payloads

We implemented payloads for the three examples from Section 2.2. Here we describe the payloads in detail.

Domain generation The domain generation payload provides key functionality for a secure command and control scheme, in which malware generates time-based domain names unpredictable to an analyst. As input, the payload takes the contents of a package release manifest for the Ubuntu distribution, and its associated signature. The payload verifies the signature against a public key within itself. If the signature verifies correctly, the payload extracts the date contained in the manifest. The payload outputs an HMAC of the date with a secret key contained in the encrypted payload.

Assuming an analyst is unable to provide correctly signed package manifests for future dates, this payload provides a secure random value unpredictable to an analyst, but generatable in advance by the payload's author (because the author knows the secret HMAC key). Such a random value can be used as a seed in a domain generation scheme similar to that of the Conficker worm.

Data exfiltration The data exfiltration payload searches for sensitive data (we looked for credit card numbers), and returns results in encrypted form. To avoid analysis by correlating input with the presence or absence of output, the payload generates some output regardless of whether sensitive data is present in the file.

Timebomb This payload implements key cloaked functionality necessary for a timed DDoS attack that keeps the target and time secret until the attack begins. Like the domain generation payload, it uses signed package release manifests to establish an authenticated current timestamp. Once the payload has verified the signature on the manifest, it extracts the date. If the resultant date is later than a value encoded in the encrypted payload, it releases the time-sensitive information as output. This payload outputs a secret AES key contained in the encrypted payload. The key can be used to decode a file providing further instructions, such as the DDoS target, or a list of commands.

5.4 Evaluation

We tested our implementation on a Dell Optiplex 780 with a quad-core 2.66 Ghz Intel Core 2 CPU with 4 GB of RAM running Linux 2.6.30.5. We used a ST Microelectronics ST19NP18 TPM, which is TCG v1.2 compliant. Elapsed wallclock times for protocol phases are indicated in Table 4. We used 2048-bit RSA encryption and 128-bit AES encryption. The malicious payloads varied in size from 2.5 KB for the command and control to 0.5 KB for the text search.

Costs for infecting a machine	
Action	**Time (s)**
Infected Platform generates binding key	19.4 ± 11.2
Infected Platform generates AIK and credential request	31.6 ± 17.9
Malware Distribution Platform processes request	0.07 ± 0
Infected Platform certifies key	5.9 ± 0.012
Infected Platform decrypts credential	6.0 ± 0.010
Malware Distribution Platform verifies proof	0.04 ± 0
Total	63.1 ± 22.2

Per-payload execution statistics	**Time (s)**
MLE setup	1.05 ± 0.01
Time to decrypt payload	3.07 ± 0.01
Command and Control	0.008 ± 0
DDoS Timebomb	0.008 ± 0
Text Search	0.004 ± 0
Time system appears frozen	3.22
Total MLE execution time	4.27

Table 4: Performance of different phases. Error bars are standard deviations of sample sets. A standard deviation of "0" indicates less than 1 ms. Statistics for the protocol up to late launch were calculated from 10 protocol cycles run one immediately after the other, while late launch payload statistics were calculated from 10 other runs per payload, one immediately after the other.

The main performance bottleneck is TPM operations, especially key generation. We verified that the significant and variable duration of key generation was directly due to underlying TPM operations. The current performance, one minute per machine infection, allows rapid propaga-

Program	**Purpose**	**Correspondence to Protocol**
`tpm_genkey`	Generates the binding key and output key blob to a file.	**Infection Keygen** step 1
`aik_gen`	Generates an AIK and accompanying certification request. Outputs key blob and request to files.	**Infection Keygen** steps 2– 4
`tpm_certify`	Certifies the binding key under the AIK.	**Infection Proof** step 5
`infected`	Two modes: `proof` which generates a proof of authenticity to convince the Malware Distribution Platform to distribute an encrypted payload and `payload` which loads the binding key and decrypts the payload.	`proof`: **Infection Proof** steps 1–4 and 6, `payload`: **Infection Hidden Execute** steps 1–3
`platform`	Two modes: `req` which handles a request from the Infected Platform and returns an encrypted credential and `proof` which validates a proof of authenticity from the Infected Platform	`req`: **Malware Distribution Platform Certificate Handler**, `proof`: **Malware Distribution Platform Payload Delivery**

Table 3: Programs that comprise the key establishment part of the implementation and their functions.

tion of malware (hosts can be compromised concurrently).

Performance is most important for operations on the Malware Distribution Platform, which may have to service many clients in rapid succession, and in the final payload decryption, as it occurs in late launch with the operating system suspended. The payload decryption must occur per payload execution, which in our motivating scenarios will be at least daily. The slowest operation on the Malware Distribution Platform can handle tens of clients per second with no optimization whatsoever.

We provide several numbers that characterize late launch payload performance. The MLE setup phase of the Flicker kernel module involves allocation of memory to hold an MLE and configures MLE-related structures like page tables used by `SINIT` to measure the MLE. The Flicker module then launches the MLE, which in our case contains the Infection Payload Loader PAL. This PAL first decrypts the payload, which occupies most MLE execution time for our experiments. The payload runs, the MLE exits, and the kernel module restores prior system state.

The late launch environment execution can be as long as 3.2 s, which is long enough that an alert user might notice the system freeze (since the late launch environment suspends the OS) and become suspicious. Then again, performance variability is a hallmark of best-effort operating systems like Linux and Windows. The rootkit control program can use heuristics to launch the payload when the platform is idle or the user is not physically present.

Payload decryption performance is largely based on the speed of asymmetric decryption operations performed by the TPM. The use of TPM key blobs here involves two asymmetric decryption operations, one to allow use of the private portion of the key blob (which is stored in encrypted form), and one to use this private key for decrypting an encrypted symmetric key. Symmetric AES decryption took less than 1% of total payload decryption time in all cases, and is unlikely to become more costly even with significant increases in payload size: We found that a 90 KB AES decryption with OpenSSL (36$\times$ larger than our largest payload), took only 650 microseconds.

6 Defenses

We now examine defenses against the threat of using TPMs to cloak malware. We present multiple potential directions for combating this threat. In general, we find that there is no clear "silver bullet" and many of the proposed solutions require tradeoffs in terms of the security or usability of the TPM system.

6.1 Restricting late launch code

One possibility would be to restrict the code that can be used in late launch. For example, a system could implement a security layer to trap on `SENTER` instructions. With recent Intel hardware, a hypervisor could provide admission control, gaining control whenever `SENTER` is issued and protecting its memory via Extended Page Table protections. The hypervisor could enforce a range of policies with its access to OS and user state. For example, the TrustVisor [39] hypervisor likely enforces a policy to deny all MLEs since its goal is to implement an independent software-based trusted computing mechanism.

Restricting access to the hardware TPM is one of the best approaches to defending against our attack, but such a defense is not trivial. Setup and maintenance of this approach may be difficult for a home or small business user. Use of a security layer is more plausible in an enterprise or cloud computing environment. In that setting, the complexity centers on policy to check whether an MLE is permitted to execute in late launch. The most straightforward methods are whitelisting or signing MLEs. These raise additional policy issues about what software state to hash or sign, how to revoke hashes or keys, and how to handle software updates. Any such system must also log failed attempts and delay or ban abusive users.

It is possible to use other system software to control admission to MLEs. `SINIT`, which itself is signed by Intel, could restrict admission to MLEs since all late launches first transfer control to `SINIT`. However, this would require `SINIT`, which is low-level system software, to enforce access control policy. It would most likely do this by only allowing signed MLEs to run. There are then two options: either MLEs must be signed by a key that is known to be trusted, or `SINIT` must also contain code for key management operations like retrieving, parsing, and validating certificates. In the former case, the signing key is most likely to be from Intel; Intel chipsets can already verify Intel-signed data [12]. However, this makes third party development more difficult; code signing is most effective when updates are infrequent and the signing party is the code developer. For late launch MLEs, it is quite possible that neither will be the case. The latter case, having `SINIT` manage keys, is likely to be difficult to implement, especially since `SINIT` cannot use OS services.

6.2 TPM Manufacturer Cooperation

A malware analyst could defeat our attack with the cooperation of TPM manufacturers. Our attack uses keys certified to be TPM-controlled to distinguish communication with a legitimate TPM from an analyst forging responses from a TPM. A TPM manufacturer cooperating with analysts and certifying illegitimate EKs would defeat our attack, by allowing the analyst to create a software-controlled late-launch environment. However, any leak of a certificate for a non-hardware EK would undermine the security of all TPMs (or at least all TPMs of a given manufacturer). Malware analysis often occurs with the cooperation of government, academic, and commercial institutions, which raises the probability of a leak.

Alternately, a manufacturer might selectively decrypt data encrypted with a TPM's public EK on-line upon request. Such a service would compromise the Privacy CA protocol at the point where the Privacy CA encrypts a credential with the EK for a target TPM-containing platform. The EK decryption service would allow an analyst to obtain a credential for a forged (non-TPM-generated) AIK. This is less dangerous than the previous situation, as now only parties that trust the Privacy CA (in our case the Malware Distribution Platform) could be mislead by the forged AIK. However, this approach also places additional requirements on the manufacturer, in that it must respond to requests for decryption once per Malware Distribution Platform, rather than once per analyst. Additionally, the EK decryption service has potential for abuse by an analyst if legitimate Privacy CAs are deployed.

6.3 Attacks on TPM security

Cloaking malware with the TPM relies on the security of TPM primitives. A compromise of one or more of these primitives could lead to the ability to decrypt or read an encrypted payload. For instance, the exclusive access of late launch code to system DRAM is what prevents access to decrypted malicious payloads. A vulnerability in the signed code module that implements the late launch mechanism (and enables this exclusive access) could allow an analyst to read a decrypted payload [55].

Physical access to a TPM permits other attacks. Some TPM uses are vulnerable to a reset of the TPM without resetting the entire system, by grounding a pin on the LPC bus [32]. Late launch, as used by our malware, is not vulnerable to this attack. LPC bus messages can be eavesdropped or modified [37], revealing sensitive TPM information. In addition, sophisticated physical deconstruction of a TPM can expose protected secrets [51]. While TPMs are not specified to be resistant to physical attack, the tamper-resistant nature of TPM chips indicates that physical attacks are taken seriously. It is likely that physical attacks will be mitigated in future TPM revisions.

One potential analysis tool is a cold boot attack [29] in which memory is extracted from the machine during operation and read on a different machine. In practice the effectiveness of cold boot attacks will be tempered by keeping malicious computations short in duration, as it is only necessary to have malicious payloads decrypted while they are executing. Additionally, it may be possible to decrypt payloads in multiple stages , so only part of the payload is decrypted in memory at any one time. Memory capture is a serious concern for data privacy in legitimate TPM-based secure computations as well. It is important for future trusted computing solutions to address this issue, and the addition of mechanisms that defend against cold boot attacks would increase the difficulty of avoiding our attack.

6.4 Restricting deployment and use of TPMs

Our attack requires that the malware platform knows SRK and owner AuthData values for the TPM. The danger of malware using TPM functionality could be mitigated by careful control of AuthData. Existing software that uses the TPM takes some care to manage these values. For instance, management software used in Microsoft Windows prevents the user from storing owner AuthData on the same machine as the TPM. Instead, it can be saved to a USB key or printed in hard copy. Administrators who need TPM functionality would ideally understand these restrictions and manage these values appropriately. Average users will be more difficult to educate.

The malware platform could initialize a previously uninitialized TPM, thereby generating the initial AuthData. For our test machines, TPM initialization is protected by a single BIOS prompt that can be presented on reboot at the request of system software. To prevent an inexperienced user from initializing a TPM at the behest of malicious software, manufacturers could require a more involved initialization process. The BIOS could require the user to manually enter settings to enable system software to assert physical presence, rather than presenting a single prompt. More drastically, a user could be required to perform some out-of-band authentication (such as calling a computer manufacturer) to initialize the TPM. However, all of these security features inhibit TPM usability.

6.5 Detection of malware that uses TPMs

Traffic analysis is a common malware detection technique. Malware that uses the TPM will cause usage patterns that might be anomalous and therefore could come to the attention of alert administrators. Of course detecting anomalous usage patterns is a generally difficult problem, especially if TPM use becomes more common.

7 Related Work

Malware Analysis. TPM cloaking is a new method for frustrating static and dynamic analysis that is more powerful than previous methods because it uses hardware to prevent monitoring software from observing unencrypted code. The most effective analysis technique would be a variant on the cold boot attack [29], where the infected machine's DRAM chips were removed during the late launch session. Note that a late launch session generally only lasts seconds. If the DRAM chips are pulled out too early, the payload will still be encrypted; too late and the payload is scrubbed out of memory. The analyst could also snoop the memory bus or the LPC bus. Note that both of these are hardware techniques, and they are both effective attacks against legitimate TPM use.

Our protocol does run substantial malware outside the cloaked computation. All such malware is susceptible to static analysis [30, 47, 23], dynamic analysis [19, 58, 36], hybrids [24, 35] , network filtering [16, 49], and network traffic analysis [20]. To effectively use the TPM the malware must only decrypt its important secrets within the cloaked computation.

Polymorphic malware changes details of its encryption for each payload instance to avoid network filtering. Our system falls partially into the polymorphic group as we encrypt our payload. However dynamic analysis techniques [36] are effective against polymorphic encryption because such schemes must decrypt their payload during execution. Conficker as well as other modern malware use public key cryptography to validate or encrypt a malicious payload [43], as our cloaking protocol does.

Trusted Computing. The TPM can be used in a variety of contexts to provide security guarantees beyond that of most general-purpose processors. For instance, it can be used to protect encryption keys from unauthorized access, as in Microsoft's BitLocker software [7], or to attest that the computer platform was initialized in some known state, as in the OSLO boot loader [32]. Flicker [40] uses TPM late launch functionality to provide code attestation for pieces of code that are instantiated by, and return to, a potentially untrusted operating system. Bumpy [41] uses late launch to protect sensitive input from potentially untrusted system software. Our prototype malware platform uses the same functionality, adding encryption to conceal the code payload.

Cryptography. Using cryptography for data exfiltration was suggested by Young and Yung [59]. Bethencourt, Song, and Waters [18] showed how using singly homomorphic encryption one could do cryptographic exfiltration. However, the techniques were limited to a single keyword search from a list of *known* keywords and the use of cryptography significantly slowed down the exfiltration process. Using fully homomorphic encryption [28] we could achieve expressive exfiltration, however, the process would be too slow to be viable in practice.

8 Conclusions

Malware can use the Trusted Platform Module to make its computation significantly more difficult to analyze. Even though the TPM was intended to increase the security of computer systems, it can undermine computer security when used by malware.

We explain several ways that TPM-enabled malware can be defeated using good engineering practice. TPMs will continue to be widely distributed only if they demonstrate value and do not bring harm. Establishing and disseminating good engineering practice for TPM management to both IT professionals and home users is an essential part of the TPM's future.

Acknowledgments

We thank the anonymous reviewers for their comments on an earlier version of this paper, and Jonathan McCune for access to the Flicker source code. This research is supported by NSF CNS-0905602, a Google research award, and the NSF Graduate Research Fellowship Program.

Waters is supported by NSF CNS-0915361 and CNS-0952692, AFOSR Grant No: FA9550-08-1-0352, DARPA PROCEED, DARPA N11AP20006, Google Faculty Research award, the Alfred P. Sloan Fellowship, and Microsoft Faculty Fellowship.

References

[1] MyDoom.C Analysis, 2004. `http://www.secureworks.com/research/threats/mydoom-c/`.

[2] W32/MyDoom@MM, 2005. `http://vil.nai.com/vil/content/v_100983.htm`.

[3] W32/AutoRun.GM. F-Secure, 2006. `http://http://www.f-secure.com/v-descs/worm_w32_autorun_gm.shtml`.

[4] Encryption of Sensitive Unclassified Data at Rest on Mobile Computing Devices and Removable Storage Media, 2007. `http://iase.disa.mil/policy-guidance/dod-dar-tpm-decree07-03-07.pdf`.

[5] Owning Kraken Zombies, a Detailed Discussion, 2008. `http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies`.

[6] TrouSerS - The open-source TCG Software Stack, 2008. `http://trousers.sourceforge.net`.

[7] BitLocker Drive Encryption Step-by-Step Guide for Windows 7, 2009. `http://technet.microsoft.com/en-us/library/dd835565(WS.10).aspx`.

[8] Intel Trusted Execution Technology (Intel TXT) MLE Developer's Guide, 2009.

[9] ST Microelectronics, 2010. Private communication.

[10] AMD64 Architecture Programmer's Manual, Volume 2: System Programming, 2010.

[11] Embedded security. Infineon Technologies, 2010. `http://www.infineon.com/tpm`.

[12] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, 2010.

[13] Microsoft Security Bulletin Search, 2010. `http://www.microsoft.com/technet/security/current.aspx`.

[14] Trusted Computing Whitepaper. Wave Systems Corporation, 2010. `http://www.wave.com/collateral/Trusted_Computing_White_Paper.pdf`.

[15] PolarSSL Open Source embedded SSL/TLS cryptographic library, 2011. `http://polarssl.org`.

[16] AH KIM, H., AND KARP, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security* (2004).

[17] BERGER, S., CACERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security* (2006).

[18] BETHENCOURT, J., SONG, D., AND WATERS, B. Analysis-Resistant Malware. In *NDSS* (2008).

[19] BRUMLEY, D., HARTWIG, C., LIANG, Z., NEWSOME, J., SONG, D., AND YIN, H. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection*. Springer, 2008.

[20] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *CCS* (2009).

[21] CHEN, L., AND RYAN, M. Attack, Solution, and Verification for Shared Authorisation Data in TCG TPM. vol. 5983 of *Lecture Notes in Computer Science*. Springer, 2010.

[22] CHIEN, E. CodeRed Worm, 2007. `http://www.symantec.com/security_response/writeup.jsp?docid=2001-071911-5755-99`.

[23] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *USENIX Security* (2003).

[24] COMPARETTI, P. M., SALVANESCHI, G., KIRDA, E., KOLBITSCH, C., KRUEGEL, C., AND ZANERO, S. Identifying Dormant Functionality in Malware Programs. In *IEEE S&P* (2010).

[25] DOLEV, D., DWORK, C., AND NAOR, M. Nonmalleable cryptography. *SIAM J. Comput. 30*, 2 (2000), 391–437.

[26] FALLIERE, N., MURCHU, L. O., AND CHIEN, E. W32.Stuxnet Dossier, 2010. Version 1.3 (November 2010).

[27] FINNEY, H. PrivacyCA, 2009. `http://www.privacyca.com`.

[28] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC* (2009), pp. 169–178.

[29] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CAL, J. A., FELDMAN, A. J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security* (2008).

[30] HU, X., CKER CHIUEH, T., AND SHIN, K. G. Large-scale Malware Indexing Using Function-call Graphs. In *CCS* (2009).

[31] KASSLIN, K., AND FLORIO, E. Your Computer is Now Stoned (...Again!). The Rise of the MBR Rootkits, 2008. `http://www.f-secure.com/weblog/archives/Kasslin-Florio-VB2008.pdf`.

[32] KAUER, B. OSLO: Improving the security of trusted computing. In *USENIX Security* (2007).

[33] KIVITY, A. kvm: The Linux Virtual Machine Monitor. In *Ottawa Linux Symposium* (2007).

[34] KNOWLES, D., AND PERRIOTT, F. W32.Blaster.Worm, 2003. `http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99`.

[35] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and Efficient Malware Detection at the End Host. In *USENIX Security* (2009).

[36] KOLBITSCH, C., HOLZ, T., KRUEGEL, C., AND KIRDA, E. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *IEEE S&P* (2010).

[37] KURSAWE, K., SCHELLEKENS, D., AND PRENEEL, B. Analyzing Trusted Platform Communication. In *ECRYPT Workshop, CRASH CRyptographic Advances in Secure Hardware* (2005).

[38] MATROSOV, A., RODIONOV, E., HARLEY, D., AND MALCHO, J. Stuxnet Under the Microscope, 2010. Revision 1.2.

[39] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE S&P* (2010).

[40] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys* (2008).

[41] MCCUNE, J. M., PERRIG, A., AND REITER, M. K. Safe passage for passwords and other sensitive data. In *NDSS* (2009).

[42] MITCHELL, C. J., Ed. *Trusted Computing*. Institution of Electrical Engineers, 2005.

[43] NAZARIO, J. The Conficker Cabal Announced, 2009. `http://asert.arbornetworks.com/2009/02/the-conficker-cabal-announced/`.

[44] O'DEA, H. The Modern Rogue - Malware with a Face. In *Virus Bulletin Conference* (2009).

[45] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. An Analysis of Conficker's Logic and Rendezvous Points, 2009. `http://mtc.sri.com/Conficker/`.

[46] POST, A. W32.Storm.Worm, 2007. `http://www.symantec.com/security_response/writeup.jsp?docid=2001-060615-1534-99`.

[47] PREDA, M. D., CHRISTODORESCU, M., JHA, S., AND DEBRAY, S. A Semantics-based Approach to Malware Detection. In *POPL* (2007).

[48] SACCO, A. L., AND ORTEGA, A. A. Persistent BIOS Infection. In *CanSecWest Applied Security Conference* (2009). `http://www.coresecurity.com/content/Persistent-Bios-Infection`.

[49] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm fingerprinting. In *OSDI* (2004).

[50] STRASSER, M., STAMER, H., AND MOLINA, J. TPM Emulator, 2010. `http://tpm-emulator.berlios.de/`.

[51] TARNOVSKY, C. Hacking the Smartcard Chip. In *Black Hat* (2010).

[52] TRUSTED COMPUTING GROUP. *TPM Main Specification*, 2007.

[53] WHEELER, D. A. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2001.

[54] WOJTCZUK, R. Exploiting large memory management vulnerabilities in Xorg server running on Linux. Invisible Things Lab, 2010.

[55] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Another Way to Circumvent Intel Trusted Execution Technology. Invisible Things Lab, 2009.

[56] WOJTCZUK, R., AND TERESHKIN, A. Attacking Intel BIOS. Invisible Things Lab, 2010.

[57] WONG, C., BIELSKI, S., MCCUNE, J. M., AND WANG, C. A Study of Mass-mailing Worms. In *ACM Workshop On Rapid Malcode* (2004).

[58] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *CCS* (2007).

[59] YOUNG, A., AND YUNG, M. Malicious Cryptography: Exposing Cryptovirology. Wiley, 2004.