The Dissertation Committee for Alan Mark Dunn
certifies that this is the approved version of the following dissertation:

Private Environments for Programs

Committee:

Emmett Witchel, Supervisor

Mathieu Baudet

Warren A. Hunt Jr.

Vitaly Shmatikov

Brent Waters

Private Environments for Programs

by

Alan Mark Dunn, B.S.; B.S.; M.A.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2014

To everyone

Acknowledgments

I'd like to thank Emmett Witchel for encouraging my interest in computer systems and for teaching me how to be a researcher. I've had many wonderful collaborators here at UT Austin and abroad that I want to thank for sharing their knowledge and experience with me: Owen Hofmann, Suman Jana, Jon Katz, Sangman Kim, Mike Lee, David Molnar, Alex Moshchuk, Don Porter, Indrajit Roy, Vitaly Shmatikov, Mark Silberstein, Helen Wang, Brent Waters, and Yuanzhong Xu. If I left anyone out, know that my memory fails only because of the sheer number of people I have to thank, not the magnitude of your contribution, so please excuse the omission. Finally, I want to thank my parents and brothers; I could not have done this without their support and encouragement.

Alan Mark Dunn

The University of Texas at Austin
August 2014

Private Environments for Programs

Alan Mark Dunn, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Emmett Witchel

Commodity computer systems today do not provide system support for privacy. As a result, given the creation of new leak opportunities by ever-increasing software complexity, leaks of private data are inevitable.

This thesis presents Suliban and Lacuna, two systems that allow programs to execute privately on commodity hardware. These systems demonstrate different points in a design space wherein stronger privacy guarantees can be traded for greater system usability.

Suliban uses trusted computing technology to run computation-only code privately; we refer to this protection as "cloaking". In particular, Suliban can run malicious computations in a way that is resistant to analysis. Suliban uses the Trusted Platform Module and processor late launch to create an execution environment entirely disjoint from normal system software. Suliban uses a remote attestation protocol to demonstrate to a malware distribution platform that the environment has been correctly created before the environment is allowed to receive

a malicious payload. Suliban's execution outside of standard system software allows it to resist attackers with privileged operating system access and those that can perform some forms of physical attack. However, Suliban cannot access system services, and requires extra case-by-case measures to get outside information like the date or host file contents. Nonetheless, we demonstrate that Suliban can run computations that would be useful in real malware. In building Suliban, we uncover which defenses are most effective against it and highlight current problems with the use of the Trusted Platform Module.

Lacuna instead aims at achieving **forensic deniability**, which guarantees that an attacker that gains full control of a system after a computation has finished cannot learn answers to even binary questions (with a few exceptions) about the computation. This relaxation of Suliban's guarantees allows Lacuna to run full-featured programs concurrently with non-private programs on a system. Lacuna's key primitive is the **ephemeral channel**, which allows programs to use peripherals while maintaining forensic deniability. This thesis extends the original Lacuna work by investigating how Linux kernel statistics leak private session information and how to mitigate these leaks.

Contents

Chapter 1

Introduction

The complexity of modern computer systems makes it difficult to provide privacy guarantees for programs that use them. Programs can send data to a variety of system servers (e.g., the display server) or other applications, write it to disk, send it across the network, and even keep it in memory for arbitrarily long periods of time [CPGR05]. Even systems that specifically aim to keep application secrets often leak information in unexpected ways because they are not comprehensive in scope. For example, private browsing modes in modern web browsers reveal information about which sites have been visited by leaving behind indicators of which self-signed certificates they have encountered [ABJB10], and common desktop applications can leak the existence of hidden files on a hard disk through saved settings like recently visited files [CHK+08].

In light of the variety of ways that applications fail to contain their secrets, many have suggested building support for privacy into systems themselves [CPGR05, OMRK13, TAB+12]. System support allows programs to avoid making privacy mistakes by using a single well-tested implementation of privacy primitives. However, it is not immediately clear what privacy support systems should provide. There are many possible notions of privacy that one might want and ensuring that a system

protects privacy can have implications for its usability and programmability.

This thesis examines the possibilities for system support for privacy by presenting two systems, Suliban [DHWW11] and Lacuna [DLJ+12], that allow programs to execute privately on commodity hardware. Suliban uses trusted computing technology, in particular, the Trusted Platform Module (TPM) and processor late launch [int14a, int09, amd10], to establish a private environment entirely disjoint from the operating system (OS) where purely computational programs that do not require OS services can run. Lacuna, on the other hand, uses virtualization and OS modifications to provide privacy for full-featured programs that run within an OS. Suliban and Lacuna help to illustrate the full spectrum of the tradeoff between privacy guarantees and usability; Suliban provides strong privacy guarantees for limited sets of programs, while Lacuna allows a wide variety of programs to run with a more limited (yet still useful) notion of privacy.

Suliban allows computation-only code to run in a minimal environment where it will be the only code running on the system at that time. Suliban uses processor late launch to ensure the proper setup of an environment protected from snooping by concurrent code and corruption by system devices.

Note that Suliban was designed with additional security guarantees in mind: Suliban was designed to allow malware to thwart analysis on platforms that the malware writer does not control. As a result, it is not enough that certain code *can* be run in a private manner with Suliban; it is necessary to ensure that certain code *can only* be run when its privacy is guaranteed. Suliban uses the TPM to ensure that code that needs privacy is restricted to systems where privacy is guaranteed.

Lacuna is a system that can run full-featured applications in "private sessions" that protect application secrets. These private sessions can run concurrently with other programs on the system. Lacuna provides **forensic deniability**, which is the property that no traces of application secrets are left behind after application

termination.

The main technical challenge in running private applications within a host OS that can be later examined for traces is letting private applications use system devices. To use system devices, private applications need to transfer sensitive data to and from system devices in a way that does not leave traces in the OS that can be read by an attacker at a later point. One of Lacuna's key technical contributions to allow transfer of sensitive data to and from devices is an abstraction called the **ephemeral channel**. Lacuna's ephemeral channels allow private applications to privately use a wide variety of system devices, like user input devices, graphics cards, network devices, and sound cards, while still allowing non-private applications to use those devices as normal.

The contrast between Suliban and Lacuna can be seen by comparing their usability and resistance against various attack types. Suliban is only capable of running computational code, and only in an environment that is separate from the host OS and monopolizes control of a host during execution. Lacuna, on the other hand, can run full-featured applications, and can be used concurrently with non-private applications.

The two systems also have differing attack resistance; Suliban was designed to be more attack resistant, while Lacuna instead is still useful in a threat model where many attacks against Suliban are out of scope. In particular, in Lacuna, nothing prevents malicious code from being run concurrently with private programs. Concurrently running malicious code has an easier time of extracting information via **side-channel attacks**, which involve determining program secrets unwittingly reflected in publicly available information. For example, it is possible for a malicious process to determine the web access patterns of a browser merely by being able to sample the amount of memory that the browser occupies over time [JS12]. Side-channel attacks often will not work without the ability to run malicious code

concurrently with code being attacked.

Despite the fact that Lacuna's forensic deniability is a weakening of the privacy guarantees provided by Suliban, it is still a useful privacy guarantee. The insights that forensic deniability is a useful privacy guarantee and that it is easier to build usable privacy systems that provide this guarantee than it is to build those that provide stronger guarantees are contributions of work on Lacuna.

Outline This thesis proceeds by describing related work motivating privacy protection and techniques both for attacking and protecting program privacy.

The thesis continues on to describe Suliban (§3) and Lacuna (§4) each in turn. These chapters flesh out what the privacy guarantees provided by Suliban and Lacuna entail by developing detailed examples of applications that are protected by each.

The chapter on Suliban describes a protocol in terms of network messages, cryptographic functions, and TPM operations to accomplish Suliban's security goals. It provides enough background on the TPM and processor late launch to be entirely self-contained. It also includes an extensive analysis of possible subversions of Suliban and why they fail.

The chapter on Lacuna describes novel attacks that when paired with desired usability constraints motivate forensic deniability as a privacy goal. It describes and evaluates a prototype system that achieves forensic deniability for a wide variety of applications. The design of Lacuna makes it potentially vulnerable to a limited form of side-channel attack from statistics that the Linux kernel keeps, and the chapter empirically demonstrates that these leaks are not severe and can largely be mitigated.

Finally, we summarize the content of the thesis and describe some ways in which its results will impact the design of future privacy-enhancing systems.

Chapter 2

Related work

Related work for this thesis includes studies that show the degree to which programs currently leak secrets, techniques for determining program secrets, and techniques for protecting program secrets.

2.1 Motivation for private environments for computation

2.1.1 Lifetime of sensitive data

Copies of sensitive data can remain in memory buffers, file storage, database systems, crash reports, etc. long after they are no longer needed by the application [Vie01, GPCR04, CPG+04, SML07, BHS03], or leak through accidentally disclosed kernel memory [HX07, OR11]. To reduce the lifetime of sensitive data, Chow et al. proposed **secure deallocation** of memory buffers [CPGR05]. Chow et al. focus on reducing average data lifetime, whereas forensic deniability requires minimizing worst-case data lifetime. A position paper [KAMC11] identifies the problem of worst-case data lifetime and suggests using information flow and replay to solve it.

2.1.2 Data remanence

There has been much work on data remanence in RAM, magnetic, and solid-state memory [Gut96, Gut01, HSH+08], as well as secure deletion techniques focusing on flash memory [LYH+10, LSWK11, SW10, WGSS11, RCB12]. Lacuna ensures that program secrets are in well-defined places in memory (program address spaces) or are on storage encrypted with keys that are in well-defined places in memory (structures we created in the kernel), so all program secrets can be made irrecoverable with no writes to storage. Lacuna also does not store data beyond program execution, and so does not need techniques to securely erase from storage.

2.2 Determining program secrets

2.2.1 Malware analysis

If an application binary is available for direct analysis or can be run, then prior work in malware analysis can determine much about it. All such binaries are susceptible to static analysis [HcCS09, PCJD07, CJ03], dynamic analysis [BHL+08, YSE+07, KHKK10], hybrids [CSK+10, KCK+09], network filtering [aKK04, SEVS04], and network traffic analysis [CPKS09]. These techniques are strong motivation to ensure that programs for which privacy is required cannot be observed while executing and leave no traces of their state. All code contained in initial VM images used with Lacuna is available to an attacker, but the forensic deniability threat model excludes observation of code while it is running.

2.2.2 Side-channel attacks

Several prior works have shown how different pieces of accounting information stored in the `proc` filesystem in Linux and other variants of UNIX can be exploited by concurrently running malicious processes to learn private information from other

victim processes [JS12, QM12, ZW09, ZDH+13]. Additionally, analysis of other forms of seemingly innocuous information available during program execution, like the inter-packet timing of network traffic [SWT01] and the speed of operations that affect system caches [OST06, ZJRR12], can reveal program secrets.

2.2.3 Physical attacks

There are a number of physical attacks against trusted computing hardware that are relevant to Suliban's privacy properties. Suliban is not able to prevent all possible physical attacks, like physically deconstructing a TPM to retrieve its secrets [Tar10]. Nonetheless, other prior attacks against the TPM like a manual reset to change PCR values have been mitigated in hardware, and others like LPC bus snooping [Kau07] are important for manufacturers to mitigate in the future to prevent unsophisticated attacks against trusted computing hardware. 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.

There are other possible physical attacks that do not directly target the TPM, like data recovery from RAM by moving it to a new platform [HSH+08]. For Suliban, one would have to do this while a cloaked computation is in progress, which can be made difficult by keeping cloaked computations short in duration.

2.3 Protecting program secrets

2.3.1 Full-system approaches

PrivExec [OMRK13] adds OS-level support to create private process groups wherein processes can only communicate with each other and file and swap data are encrypted with a per-process group key. PrivExec does not prevent leaks through the

OS or any non-storage I/O channels. Additionally, the threat model described in PrivExec allows a concurrent unprivileged attacker, which leaves the system vulnerable to side-channel attacks. By contrast, Lacuna's concept of forensic deniability is a strong guarantee with a well-motivated threat model.

CleanOS [TAB+12] helps mobile applications protect their secrets from future compromise by encrypting sensitive data on the phone when applications are idle. It does not prevent leaks through the OS and I/O channels.

2.3.2 Protecting file data

Boneh and Lipton observed that data can be "cryptographically erased" by encrypting it first and then erasing the key [BL96]. Many cryptographic file systems use encryption to (1) protect the data after the computer has been compromised, and/or (2) delete the data by erasing the key [Bla94, ZBS98, efs, PBH+05, Per05]. Recently, encrypted file systems have been proposed for secure deletion of flash memory [LYH+10, LSWK11, RCB12]. Encrypted file systems that derive encryption keys from user passwords are not coercion-resistant. ZIA relies on a hardware token to provide the decryption key when the token is in physical proximity to the machine [CN04].

In contrast to block storage-level encryption, filesystem-level encryption does not provide forensic deniability. For example, early versions of the encrypted file system in ChromeOS on a Cr-48 laptop were based on eCryptfs [ecr] which reveals sizes of individual objects, allowing easy identification of many visited websites in the encrypted browser cache using standard fingerprinting techniques based on HTML object sizes [SSW+02, Dan10].

Provos observed that application data stored in memory may leak out via OS swap and proposed encrypting memory pages when they are swapped out [Pro00]. Lacuna uses a similar idea for implementing encrypted swap.

Steganographic and deniable file systems aim to hide the existence of certain files [MK99, PTZ03, GAB10]. This is a stronger privacy property than forensic deniability. Czeskis et al. showed that the OS and applications can unintentionally reveal the existence of hidden files [CHK+08]. Deniable file systems can be used in combination with Lacuna for stronger privacy protection.

2.3.3 Hardware support

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 [bit09], or to attest that the computer platform was initialized in some known state, as in the OSLO boot loader [Kau07]. Flicker [MPP+08] 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 [MPR09] uses late launch to protect sensitive input from potentially untrusted system software. Suliban uses the same functionality, adding encryption to conceal code payloads.

Since the time that the original research on Suliban was performed, other hardware has emerged that also tries to provide assurance of code isolation and integrity. The TPM 2.0 specification [Tru13] is a significant change to the way TPMs operate. Additionally, Intel's future Software Guard Extensions (SGX) [int13, MAB+13, AGJS13] will provide processor-based mechanisms for isolating code and attesting its initial state to a remote platform. However, we show that even hardware that differs significantly from TPM 1.2 can likely be used for malware cloaking in Section 3.10.

The goal of DRM is to restrict users' control over digital content. Some DRM systems encrypt application data which may reduce its lifetime, but any resulting deniability is incidental. For example, high-bandwidth digital content protection

(HDCP) is a cryptographic protocol that prevents content from being displayed on unauthorized devices, but the content is still exposed to the X server and GPU device drivers. DRM is controversial [Fel], and we believe that solutions for protecting user privacy should not be based on proprietary DRM technologies.

2.3.4 Isolation

Lampson [Lam09] discusses the idea of two separate systems, only one of which ever sees sensitive data (that one is red, the other green). Several systems switch between "secure" and regular modes [SWZS12, Moh11, BWLP09, VPQP09]. They do not provide forensic deniability for the red system and often require all activity on the green system to cease when the red one is active. Pausing the green system can disrupt network connections, e.g., to a cloud music service. Lacuna supports concurrent, finely interleaved private and non-private activities.

Xoar [CNZ+11] and Qubes [qub] break up the Xen control VM into security domains to minimize its attack surface and enforce the principle of least privilege; Qubes also facilitates partitioning of user applications. These systems provide an implementation of an inferior VM [PJ10] (aka disposable VM) that isolates untrusted programs in a fast-booting,[1] unprivileged, copy-on-write domain. Although not designed for minimizing data lifetime per se, these systems could be Lacuna's underlying virtualization mechanism instead of QEMU. Lacuna's ephemeral channels can support private sessions regardless whether the underlying hypervisor is monolithic or compartmentalized.

Tahoma [CHGL06] and the Illinois Browser OS [TMK10] increase the security of Web applications using a combination of hypervisors and OS abstractions. They do not limit data lifetime within the host system.

Systems with multi-level security (MLS) and, in general, mandatory access

[1] 4-5 seconds, per `http://theinvisiblethings.blogspot.com/2010/10/qubes-alpha-3.html`

control (MAC) can control information flow to prevent information from disclosure. Some MAC systems separate trusted and untrusted keyboard input [KZB+90] as Lacuna does. We are not aware of any MAC, MLS, or more modern (e.g., [MAF+11, KAMC11, YMC07]) system that provides deniability against an attacker who compromises the system after a private session is over.

2.3.5 Cryptographic techniques

Prior work has examined using cryptographic techniques to accomplish similar goals to Suliban's payloads. Using cryptography for data exfiltration was suggested by Young and Yung [YY04]. Bethencourt, Song, and Waters [BSW08] 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 [Gen09] we could achieve expressive exfiltration, however, the process would be too slow to be viable in practice.

Additionally, one area of cryptographic research has examined the ability to **obfuscate** programs. Intuitively, the goal of program obfuscation is to take programs and produce "obfuscated versions" that behave like a black box; an obfuscated version of a program computes the same function as the original, and no more should be learnable from the obfuscated version of a program than from observing the input/output behavior of the original program. Program obfuscation accomplishes a similar function to Suliban, and cryptographically obfuscated programs would not suffer from some of Suliban's limitations like necessitating a per-infected platform communication with a remote malware distribution platform. Despite initial negative results [BGI+12], that show a general obfuscator for all programs is impossible, recent work [GGH+13, BGK+14] shows that a general obfuscator is possible in limited attacker models. It may be possible to obfuscate significant classes

of program without these attacker restrictions. Suliban can also be viewed as a way of experimenting with program obfuscation while program obfuscation algorithms are still impractical.

2.3.6 Mitigating side-channel attacks

One class of mitigations to side-channel attacks is to degrade the information available to the attacker. As timing information is critical to most side-channel attacks, a number of papers propose degrading timing information available to an OS [MDS12, VDS11]. Lacuna's mitigations also degrade information available to an attacker, but mostly involve either not collecting information when possible or changing information exactly at private session termination. When information is changed exactly at private session termination, Lacuna does not affect measurement differences in the period after a private session, unlike perturbing a system time source. Prior work has also proposed giving false data to applications that demand readings but can operate (potentially with reduced functionality but at a level acceptable to a user) without accurate data [BRSS11, HHJ+11].

Another class of mitigations is to instead change systems to cause programs to not leave information that can be used for side-channel attacks. This can be done at the application level, e.g., by designing timing-channel resistant cryptographic primitives [OST06], or at the system level, e.g., by giving applications a way to get locked cache lines [KPMR12].

Chapter 3

Cloaking malicious computation with trusted computing

The TPM is a piece of hardware deployed in commodity computer systems to improve system security by tracking platform state and tying the availability of secrets to this platform state. The TPM's low cost and potential to provide greater control over program secrets have led to it becoming widespread; over 350 million deployed computers have TPM hardware [wav10]. Despite the intention of the TPM to improve computer security, Suliban[1] was designed to show that careful use of the TPM can actually *harm* computer security by helping malware to resist analysis.

Suliban can take a computation-only piece of code (one that does not access system services), and run it in an environment that is shielded from analysis (we refer to this protection as "cloaking"). As we will see, preventing analysis of small computation-only pieces of code is sufficient to severely impair malware analysts' ability to stop real-world attacks.

Suliban occupies a unique space among both attacks related to the TPM (and late launch) and program obfuscation techniques. Every other TPM-related attack

[1]This system is named after the owners of one of the first cloaking devices in Star Trek.

is centered around trying to compromise TPM security properties. By contrast, we aim to show that due to the way use of TPM capabilities is authorized in practice that we can engineer an attack that *uses* TPM security properties. In turn, the TPM's security properties give the attack resilience to a wide variety of well-known malware analysis techniques (for reference, a number of classes of these techniques are listed in §2.2.1).

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 cloaking 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 work 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 not to be externally observable, e.g., by a malware analyst. Our protocol adapts TPM-based remote attestation for use by a 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 describes experiences using TPMs in practice that highlight security issues in their use and deployment.

- 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 security and usability.

3.1 Motivation: Cloaking Conficker B

To see how cloaking can affect efforts to combat malware, it is helpful to consider an example. We will discuss Conficker B, one variant of a worm that was widespread in the late 2000s. The worm has an infection stage, where a process on the host is exploited and the process 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 where Conficker was going to look for updates [PSY09, Naz09].

Conficker's domain name generation mechanism works by scraping websites to find the current date (in UTC). It then uses the date to seed a random number generator. Conficker generates a list of 250 domains, each of which is a randomly chosen sequence of lowercase letters and top-level domain. Conficker visits each of these domains in sequence, trying to download updates.

Defeating Conficker requires learning in advance the rendezvous domain names it will generate. The sequence of domain names can be determined in two ways: directly analyzing the domain name generation code, or running the domain name generation implementation with inputs that will generate future domain names.

Cloaking the domain name generation code in Conficker helps to eliminate both of these attacks. Firstly, if the domain name generation code is cloaked, then the binary itself cannot be examined, and the code's behavior as it runs cannot be analyzed. Additionally, cloaking the domain name generation code provides integrity; once the domain name generation code has obtained the date, an analyst

cannot change this value in memory to a future value before the algorithm runs.

Cloaked computation alone does not fully prevent attacks on Conficker because an analyst can modify the information that Conficker receives to calculate the date. However, other measures can be combined with cloaked computation to prevent running Conficker with future date values. Malware can obtain digitally signed timestamps that it can then verify with a key included in the binary. 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.) Package data is mirrored at many locations worldwide and is critical for the integrity of package distribution[2], so taking it offline or forging timestamps would be both difficult and a security risk.

Cloaked computation thus provides a key component to making Conficker more resistant to analysis. Using cloaked computation for domain name generation is also likely to be more widely applicable as Conficker is not alone in its use of domain name generation for rendezvous points. The Mebroot rootkit [KF08] and Kraken botnet [kra08] both use similar techniques to contact their command and control servers.

3.2 Threat model

Having motivated the need to protect a small computational portion of a piece of malware, we describe more generally the setting where Suliban is useful and the type of attacker it is meant to protect against.

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-

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

hat security researcher, or IT professional) except for its input and output behavior.

We assume the attacker will have the following capabilities on compromised machines:

- **Kernel-level privilege.** 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.
- **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 3.3. We revisit this assumption in Section 3.8.

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. We assume the analyst is neither able to mount physical attacks on the TPM itself nor is able to compromise the TPM public key infrastructure (i.e., the analyst cannot break the cryptosystems that the TPM uses).

3.3 TPM background

It is necessary to understand some background about the TPM before one can comprehend protocols that use it.

3.3.1 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. TPMs are built to meet certain standards put forth by the Trusted Computing Group (TCG). 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. TPMs are standardized by the Trusted Computing Group, and Suliban was designed to use version 1.2 TPMs [Tru07].

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. Our protocol concerns only PCRs numbered in the range 16–23 (the "dynamic" PCRs), which are used by the TPM's late launch mechanism. PCRs cannot be set directly, they can only be **extended** with new values. Extending a PCR causes it to depend on its previous value and the extending value in a way that is not easily reversible by further extend operations. PCR state can establish what software has been run on the machine since boot, including the BIOS, hypervisor (if present) and operating system.

3.3.2 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**.

AuthData controls TPM capabilities, which are the ability to read, write, and use objects stored in the TPM and execute TPM commands. Each AuthData value 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.

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 3.1. A hash of the public portion of a key blob (PubBlob in the figure) is stored in the private portion (PrivBlob in the figure), 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. $tpmProof$ protects the key blob from forgery by adversaries and even the TPM manufacturer: Firstly, without knowing $tpmProof$ an adversary cannot form a new key blob with a keypair it knows. Secondly, the TPM uses non-malleable encryption [DDN00], which means it is cryptographically hard for an adversary to form a new ciphertext from a given ciphertext such that the plaintexts are related (e.g., to mutate the private part of a blob containing $tpmProof$ to contain a different secret key, making it appear as though a key were generated by the TPM when an attacker has access to the secret portion).[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

[3]The TPM can produce migratable keys that can be moved to other TPMs and work somewhat differently. However, the details of these keys are beyond the scope of this work and any potential protocol flaws induced by these keys can be eliminated by examining TPM key certifications to find these keys and exclude them.

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

Table 3.1: Notation for TPM data and computations.

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

Figure 3.1: 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, which equivalently allow a program to check whether a key is constrained to use with known PCR values and the TPM to check whether current PCR values satisfy key constraints.)

only be used when the PCRs have certain values (and therefore only when certain software is running).

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

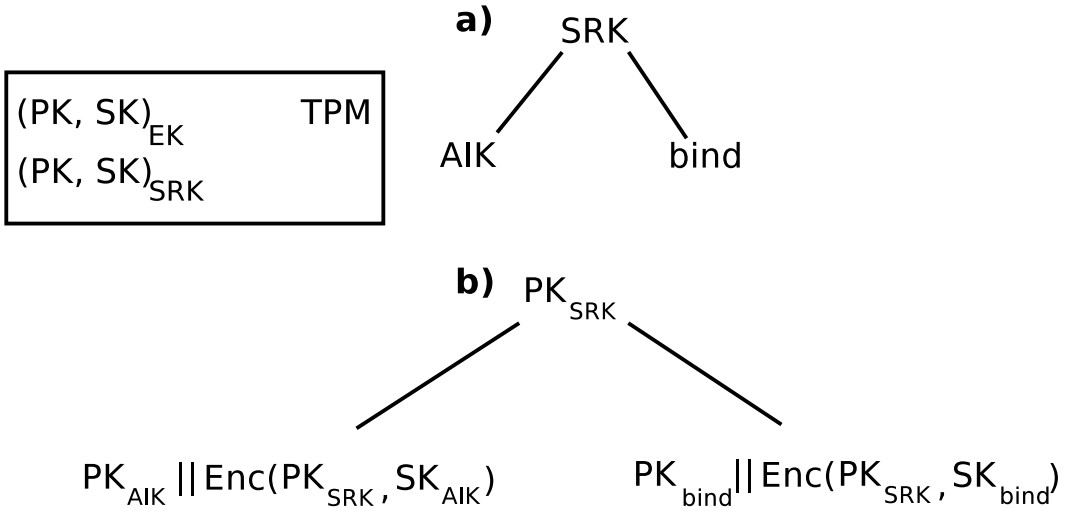

Figure 3.2: The part of the TPM key hierarchy relevant to our attack. The TPM box illustrates key material stored internal to the TPM, which is only the endorsement key (EK) and storage root key (SRK). Part (a) shows the conceptual key hierarchy, while part (b) shows how the secret keys of children are 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 3.1.

requires using the TPM to access the private SRK to decrypt private keys while descending the hierarchy.

It is cryptographically hard 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 without breaking TPM cryptographic schemes 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 3.2.

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.

3.3.4 Platform identity and attestation

TPMs provide the ability to attest parts of their state, that is, they provide cryptographic proofs of the contents of TPM state that are verifiable by remote parties. Most important for this work is the ability to attest that a key in a TPM hierarchy is stored in a particular TPM and has particular constraints on its usage. This is accomplished by having the TPM produce a signature on that key with a key that is known to be stored in the TPM.

While a TPM's EK is a key that is stored in the TPM, 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** (AIKs) that serve as proxies for the EK. However, something must associate an AIK with the EK.

A trusted privacy certificate authority (Privacy CA) is supposed to provide certificates to third parties that an AIK corresponds to an EK of a legitimate TPM. While prototype Privacy CA code exists [Fin09], 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.

3.3.5 Using the TPM

Typical uses of the TPM are to manipulate the key hierarchy, to obtain signed certificates 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.

PCRs can be modified by the TPM as platform state changes. They can only be set to specific values by platform reset and late launch, and are modified in software by extension. A PCR with value PCR extended by a 160-bit value val is set to value $\text{Extend}(PCR, val) \equiv H(PCR \,||\, val)$. The dynamic PCRs are set to all 1s on platform boot. Late launch sets the dynamic PCRs to all 0s and then extends PCR 18 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.

3.4 Protocol

We now describe an architecture and protocol for launching a TPM-cloaked attack. An overview of the protocol is pictured in Figure 3.3.

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

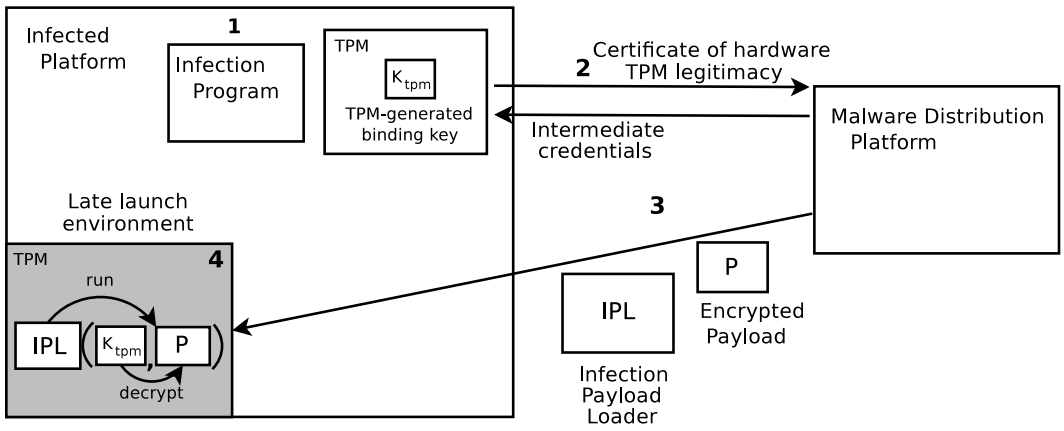

Figure 3.3: The overall flow of the attack is 1) Infecting a system with malware capable of kernel-level exploitation to coordinate the attack 2) Establishing a legitimate TPM-generated key usable only by the Infection Payload Loader in late launch via a multistep protocol with a Malware Distribution 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.

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.

3.4.1 Late launch for secure execution

The Infection Program has a module called the **Infection Payload Loader** that is responsible for decrypting and running an encrypted payload. The Infection Payload Loader runs in late launch to prevent an analyst from observing the decrypted payload and its execution. At the same time, we need to ensure that only the Infection Payload Loader can decrypt the encrypted payload. That means that the late launch process has to indicate somehow that the Infection Payload Loader is running in a way that is tied to the ability to use a cryptographic key.

On Intel x86 platforms[4], late launch is implemented with an instruction `GETSEC[SENTER]` (often referred to as `SENTER`). This instruction ensures that control is securely transferred to a block of code in memory and that the contents of this block of code are noted for later comparison.

Specifically, with Intel's late launch mechanism, the user configures a data structure describing the code to be launched (a **Measured Launch Environment** (MLE) in Intel parlance) and uses `GETSEC[SENTER]` to transfer control to an Intel-provided code module called `SINIT` that initiates the late launch. In order to

[4]AMD's implementation of late launch is similar.

securely transfer control to a block of code, it is necessary to ensure that the code remains intact. Code used in late launch is protected both from other cores on the system and from platform devices that can independently write into physical memory through Direct Memory Access (DMA). The former is accomplished by forcibly halting all but one core that will be used to run the code in late launch. (The halted cores can be restarted in the late launch environment if desired.) Protection from devices is ensured by placing the code in a chipset-defined DMA-protected region; the late launch process will refuse to execute if the code is not in such a region. Once code integrity is ensured, `SINIT` extends the contents of the code into PCR 18. As previously discussed, this means that the ability to use TPM keys can be made contingent on the late launch of a specific program by key constraints. `SINIT` itself is loaded into internal processor memory and integrity-checked via a public-key signature after loading, thus `SINIT` will itself properly act to protect the late launched code.

This means that the goal of the protocol is to encrypt a payload binary with a key that is known to be stored in a TPM and whose private portion can only be used when PCR 18 is H(Infection Payload Loader). An encrypted payload can be passed to the Infection Payload Loader in memory, and the Infection Payload Loader will be able to use the TPM to decrypt the payload and run it.

3.4.2 The main protocol

We now describe how the Infection Program and Malware Distribution Platform cooperate to run malware securely. The protocol uses a number of TPM operations that are described in Table 3.2. In the description of the protocol, we will elide lower-level TPM operations like `TPM_OIAP` and `TPM_OSAP` that are used to establish sessions in which TPM commands can be executed. The correctness of the protocol is not affected by these commands because they control only how one authorizes

key blob = `TPM_CreateWrapKey`(parent key handle, 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 3.2: 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.

TPM commands and integrity of responses from the TPM. For the former, we are assuming that we can authorize TPM commands (by knowing the appropriate AuthData). For the latter, while it is possible for an adversary to fake responses from the TPM while the infected platform is not in late launch, as we will see, this cannot compromise the protocol because the part of the protocol outside late launch results in data that are cryptographically verifiable. Once the infected platform is in late launch, the analyst has no further software control over it. The protocol is shown in full in Figure 3.4.

Conceptually, there are four phases of the protocol as will be described below. However, Figure 3.4 follows the actual protocol implementation, which combines pieces of some of the phases for ease of implementation.

1. The Infection Program and Malware Distribution Platform establish an AIK belonging to a legitimate TPM

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, $PCR18 = \text{Extend}(0_{160}, H(\textbf{Infection Payload Loader}))$) (requires SRK AuthData), store in memory
2. Create identity key $(PK, SK)_{AIK}$ under SRK in memory as 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. 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)

Figure 3.4: The cloaked malware protocol.

Infection Proof (continued):

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 legitimacy of binding key, 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 constrained.
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. Let $M_{payload\_plaintext} \equiv payload \,||\, \text{Sign}(SK_{malware}, H(payload))$. Form $M_{payload} \equiv \text{EncSym}(K_3, M_{payload\_plaintext}) \,||\, \text{Enc}(PK_{bind}, K_3)$, a message encrypted with a symmetric key which itself is encrypted with an asymmetric key (often called a hybrid encryption).
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. Use `TPM_Unbind` to recover K_3. This operation can succeed (and only in this program) because in **Infection Hidden Execute**, $PCR18 = \text{Extend}(0_{160}, H(\textbf{Infection Payload Loader}))$. Then use K_3 to decrypt $M_{payload\_plaintext} = payload \,||\, \text{Sign}(SK_{malware}, H(payload))$.
3. Verify signature $\text{Sign}(SK_{malware}, H(payload))$ with $PK_{malware}$. Abort if verification fails.
4. Execute *payload*
5. Scrub payload from memory and extend random value into PCR18, then exit late launch

Figure 3.4 (continued): The cloaked malware protocol.

TPMs can be programmed to release credentials that show their authenticity:

$$C_{EK} = PK_{EK} \,||\, \text{Sign}(SK_{manufacturer}, H(PK_{EK}))$$

The Malware Distribution Platform can check the validity of the signature on PK_{EK}. If it is valid, then the manufacturer certifies that PK_{EK} is the EK for a legitimate TPM.

A request to the Privacy CA (here the Malware Distribution Platform) to link an AIK to an EK is then of the form

$$M_{req} = C_{EK} \,||\, \text{PubBlob}((PK, SK)_{AIK}).$$

Define $H_{aik\_cert} \equiv H(\text{PubBlob}((PK, SK)_{AIK}))$.

The Malware Distribution Platform responds to this request with

$$\text{Enc}(PK_{EK}, \text{Sign}(SK_{malware}, H_{aik\_cert}) \,||\, H_{aik\_cert}).$$

Messages of the form $\text{Enc}(PK_{EK}, cred \,||\, hash)$ can be given to a TPM with public EK PK_{EK}, which will decrypt them internally and release *cred* if and only if *hash* is the hash of the public blob for some AIK generated with that TPM. Thus if the *cred* values are chosen such that they are known only to the Malware Distribution Platform, the release of *cred* for such a message indicates that *hash* was a valid hash of an AIK public blob.

The Infection Program sends the Malware Distribution Platform's response to the TPM, retrieving a value *cred* if the step succeeds. The Infection Program sends this back to the Malware Distribution Platform, which considers the AIK as linked to EK if it receives *cred*.

2. Infection Program generates binding key that is restricted to use during late launch of Infection Payload Loader

   The Infection Program uses the Infected Platform's TPM to generate a binding key that is restricted to use in the late launch environment (via a constraint PCR 18 = $H(0_{160} \,||\, H(\text{Infection Payload Loader}))$) and a certification under the AIK that the binding key is stored in the TPM and has the PCR constraint. The Infection Program sends the resultant key, $PK_{binding}$ (as part of PubBlob($(PK, SK)_{binding}$)) and certification to the Malware Distribution Platform.

   The Malware Distribution Platform can verify the signature with the AIK and then has $PK_{binding}$ where it knows decryption with $SK_{binding}$ can only be performed in the late launch environment.

3. Malware Distribution Platform produces payload

   The Malware Distribution Platform can now create encrypted payloads to run in the late launch environment. However, it is important that the Malware Distribution Platform limit the programs that can be run in the late launch environment, or else an analyst could equally well run programs in the late launch environment with her access to $PK_{binding}$. Such a program could use the late launch environment's access to $SK_{binding}$ to decrypt other payloads. As a result, payloads are signed with $SK_{malware}$ prior to encryption with $PK_{binding}$. $PK_{malware}$ is part of the Infection Payload Loader itself.

4. Infection Program runs payload via Infection Payload Loader

   The Infection Program late launches into the Infection Payload Loader, passing an encrypted payload P through memory.

   The Infection Payload Loader works as follows:

   (a) Use TPM to decrypt payload P with binding key.

(b) Verify signature on decrypted payload using $PK_{malware}$, if verification fails, abort.

(c) Transfer control to decrypted payload. Decrypted payload can output result R in memory.

(d) Erase code and data for decrypted payload. Extend random value into PCR 18. Return R.

3.5 Resilience of the Protocol

We now discuss how the main protocol resists a variety of forms of attack. Using the protocol with particular payloads may require further measures to resist attacks beyond the initial protocol.

The first phase of the protocol guarantees that the EK is from a legitimate TPM, due to the manufacturer certificate, assuming that the manufacturer has not signed invalid EKs. The validity of the EK guarantees that the receipt of $\text{Sign}(SK_{malware}, H_{aik})$ in turn guarantees that PK_{AIK} in M_{req} is the public key for an AIK stored in the same TPM as EK. Note that an AIK and EK accepted in this phase may be from a TPM that is not on a machine infected with the Infection Program due to tampering by an analyst. Since the AIK is stored in the TPM, its private portion is inaccessible to analysts.

The signature by the AIK on the public portion of the binding key and its PCR constraints ensures that the binding key is stored in the TPM where the AIK was generated. Since the signature covers the PCR constraints of the binding key, the binding key can only be used when PCR 18 has the value that would be achieved by late launching with the Infection Payload Loader.

Finally, when a platform late launches into the Infection Payload Loader, PCR 18 is set to the correct value to use the binding key. At no other point is this true: PCR 18 begins on TPM reset with value 1_{160}, and on late launch of

binary *program* is set to $H(0_{160} \,||\, H(program))$. To get extend PCR 18 to have the value achieved with late launch of the Infection Payload Loader after a different program is late launched instead involves finding the preimage for a hash function value that the attacker does not control. Late launch protects the integrity of the Infection Payload Loader, which in turn securely loads only Malware Distribution Platform-signed payloads to prevent analyst use of the binding key via analyst-provided payloads and attacks that might reveal payload secrets through program modification (i.e., modifying a decrypted payload in memory via concurrently running software). Additionally, the value of PCR 18 is extended after the payload runs, so when control is returned to the host OS, decryption cannot be performed with the binding key.

3.6 Implementation

We implemented a prototype of our attack, both of our protocol to decrypt and execute payloads in late launch, and sample attack payloads. We now describe each of these pieces in turn.

The prototype implementation consists of five programs (described in Table 3.3) for the key establishment protocol, the Infection Payload Loader, payload programs, and supporting code to connect the pieces. The key establishment portion is about 3,600 lines of C, the Infection Payload Loader is another 550 lines of C. We did not send data across a network as would be required for real operation, but instead communicated data between programs via files. Since network use only occurs with access to the full facilities of an OS, implementation of network capabilities would be straightforward. 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 [Whe01].

Program	Purpose	Correspondence to Protocol
`tpm_genkey`	Generates the binding key and outputs 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**
`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 3.3: Programs that comprise the implementation outside of the payloads and their functions.

3.6.1 Late launch environment establishment

We modified code from the Flicker [MPP+08] (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** (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 en-

crypted with the binding key, and the binding key blob as parameters. We used the PolarSSL[5] embedded cryptographic library for all our cryptographic primitives (AES encryption, RSA encryption and signing, SHA-1 hashing and SHA-1 HMACs).

3.6.2 Payloads

We implemented payloads for three example attacks. 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 (as described in §3.1).

Data exfiltration It is useful for malware to search through data on a host and exfiltrate it. Stuxnet and Aurora are examples of high profile attacks that exfiltrate information [MRHM10]. If malware exfiltrated all data that it found, this would use more network resources and make the malware more detectable. As a result, it

[5] `https://polarssl.org/`

makes sense for malware to exfiltrate only data it finds interesting, perhaps through use of regular expressions for matching patterns in data or through targeting certain filenames. However, knowing what data the malware is exfiltrating provides clues as to its goals and potentially its authors. Thus it is useful for malware to be able to obscure which data it is exfiltrating.

We demonstrate that malware cloaking can be used to perform selective data exfiltration with obscured output. The contents of files can be presented to our data exfiltration payload, which 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. A program without cloaked computation could use cryptographic techniques [YY04, BSW08, Gen09] to keep search criteria secret while being observed in memory, but their performance currently makes them impractical.

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 in the domain generation case), and only make the target known on launch day. A number of high-profile malware-based DDoS attacks have had their impacts lessened by detection of their targets in advance (e.g., MyDoom's targeting of `www.sco.com` [myd04], Storm's targeting of `www.microsoft.com` [Pos07], and Blaster's targeting of `windowsupdate.com` [KP03]).

We implement a cloaked payload that provides 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.

3.7 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 provides the TCG v1.2 TPM API. Elapsed wallclock times for protocol phases are indicated in Table 3.4. We used 2048-bit RSA encryption and 128-bit AES encryption. The malicious payloads varied in size from 2.5 KB for the largest payload (which was the command and control payload) to 0.5 KB for the smallest (which was the text search payload).

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 propagation 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

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

Per-payload execution statistics	
Action	**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 3.4: 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.

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× larger than our largest payload), took only 650 microseconds.

3.8 Attack Feasibility

Our attack is contingent on being able to acquire the proper AuthData: SRK AuthData, and owner AuthData. The ability for malware to obtain access to these AuthData has two aspects: the way that AuthData are entered into the system, and the degree of usage of the different AuthData values.

AuthData only need to be used to calculate HMAC values authorizing commands, so they do not need to be present on a machine where they are being used to authorize TPM commands. However, in practice, AuthData seems to be entered into systems where it is used. TrouSerS, the standard suite of software for TPM management for Linux, asks users for a password locally on a TPM-containing machine which is converted into AuthData. While Windows 7 does not allow the user to store owner AuthData locally, it is typically stored on removable media (i.e., a USB stick) that is inserted into the system when necessary.

The use of a TPM key requires the loading of all keys above it in the hierarchy, the use of any TPM key requires use of SRK AuthData. As a result, the SRK AuthData is likely to be used during most operations that involve the TPM. On the

other hand, owner AuthData is mainly used for important management operations, like taking ownership of the TPM. Thus there are likely to be fewer opportunities to obtain owner AuthData. Nonetheless, poor data handling practices can lead to AuthData being resident in a system for periods far beyond its use. We used a virtualized Windows 7 instance to measure the amount of time owner AuthData remains resident in a system after use in a system control panel, and saw that on an idle system the data can remain resident for days.

Our attack also depends on the ability to prove to a remote platform that a TPM is legitimate based on the certification of its EK by the TPM manufacturer. We believe that this is reasonable given that it is also required for securing normal TPM operation. Nonetheless, in practice, we encountered problems that indicate that TPM users may not be verifying the legitimacy of their TPMs. Firstly, we experienced difficulties in retrieving TPM EK certificates from TPM NVRAM. Reads of the appropriate location in NVRAM would consistently experience errors for reads of greater than or equal to 863 bytes. This behavior was consistent across multiple TPMs of the same model. The inability to perform reads of greater than or equal to 863 bytes was problematic because the TPM EK certificate was larger than this size. We were able to recover the TPM EK certificate by instead using overlapping reads of smaller size. Secondly, the certificate chain between the TPM EK certificate and any publicly verifiable certificate was not immediately available. It took months of correspondence with the TPM manufacturer for them to produce the appropriate certificate chain[6]. While the difficulty of obtaining a certificate chain to verify TPM EK authenticity may vary by manufacturer (e.g., Infineon's intermediate certificates are online[7]), in any case it appears that in practice there may be obstacles to verifying the legitimacy of TPM EKs. However, as long as it is possible to verify the legitimacy of some TPM EKs our attack can still be used.

[6]Personal communication. November 3, 2010 through April 9, 2011.

[7]`http://www.infineon.com/tpm`

Manufacturers also have incentive to fix EK certificate verification for normal TPM operation.

3.9 Defenses

While there are a number of possible defenses against using the TPM to cloak malware, none is without difficulties or downsides. Many possible defenses undermine the security guarantees that trusted computing is supposed to provide.

3.9.1 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[8]. 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.

3.9.2 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

[8]The security layer would need to operate despite compromise of the system by kernel-level malware. This could be accomplished with a lightweight hypervisor.

TPM manufacturer cooperating with analysts and certifying illegitimate EKs would defeat our attack, by allowing the analyst to create a software-controlled environment that mimics a late launch environment but that she can observe. However, any leak of a certificate for a non-TPM 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.

3.9.3 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 pay-

load [WRT09].

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 [Kau07]. Late launch, as used by our malware, is not vulnerable to this attack. LPC bus messages can be eavesdropped or modified [KSP05], revealing sensitive TPM information. In addition, sophisticated physical deconstruction of a TPM can expose protected secrets [Tar10]. 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 [HSH+08] 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.

3.9.4 Restricting deployment and use of TPMs

As discussed in Section 3.8, our attack requires that malware know SRK and owner AuthData values for the TPM. The danger of malware using TPM functionality could be mitigated by careful control of AuthData. Owner AuthData could be used only remotely to a TPM-containing platform. While an IT administrator might understand this restriction and properly implement it, comprehension and

implementation would be more difficult for average users.

Malware 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.

3.9.5 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. Detecting anomalous usage patterns is a generally difficult problem, and if TPM use becomes more common it could make cloaked malware's TPM use yet more difficult to detect.

3.10 Applicability to newer hardware

Since the time that the original research on Suliban was performed, other hardware has emerged that also tries to provide assurance of code isolation and integrity. Intel's future Software Guard Extensions (SGX) [int13] will provide processor-based mechanisms for isolating code and attesting its initial state to a remote platform. SGX's core concept is that of the **enclave**, wherein a known code and data image (with a particular hash) can be encrypted and integrity protected, and stored back into memory. At that point, the code can be run such that any modifications

it makes to its data are only stored in encrypted and integrity protected form in main memory. SGX tracks entrance to and exit from enclaves, even by non-code-generated means (e.g., interrupt handling), and restricts access to enclave secrets (e.g., code and data) appropriately. Note that any discussion about SGX is somewhat speculative as SGX is not yet deployed. All discussion here is based on Intel documentation and research papers.

Although SGX uses different mechanisms, it has a similar capacity to the TPM in combination with processor late launch to cloak malware. SGX does not directly provide a mechanism for generating encryption keypairs where the private key's access is restricted. Initial enclave code and data are also entirely visible, so private keys cannot be put into initial enclave data. However, enclaves are free to generate their own keypairs, and SGX provides a method to attest that arbitrary data came from a particular enclave. In more detail, to establish a keypair where only a particular enclave has access to the private key, the enclave can take the following steps:

- The enclave runs a keypair generation algorithm. This algorithm requires randomness, which must be obtained from a source that an attacker cannot access or modify. For example, for the cloaked malware threat model, a processor-based randomness mechanism like Intel's `RDRAND` is appropriate as an attacker would not be able to influence the outcome.

- The enclave hashes the public key and requests that the hash be cryptographically bound to the identity of the enclave via a `EREPORT` processor instruction. `EREPORT` allows an enclave to send integrity-protected data to another enclave. Here the enclave selects as its target a special enclave called the **quoting enclave**, which holds platform-wide credentials for remote attestation.

- The quoting enclave uses the **Enhanced Privacy ID** scheme to create a form of signature on the public key hash that binds the hash to the identity of the

enclave. This signature only allows a verifier to determine that the public key hash came from some instantiation of the enclave on an SGX-enabled platform, but not which platform in particular. A remote platform can then verify this signature and know that the accompanying public key can be used to send data privately to some instance of an enclave.

By use of restricted access public keys, a remote platform can put code that only it knows into an enclave and execute it.

However, there are some differences between the security guarantees provided by the TPM with late launch and SGX. Using late launch to keep secrets relies on the secrecy of regions of platform RAM, which as discussed in Section 3.9, may not be true for attackers with some form of physical access, either to snoop the memory bus or to remove the RAM of the platform at precise times. Our protocol also relies on the secrecy of information transmitted across the platform LPC bus, which may also not be the case for a sophisticated physical attacker. SGX protects against attackers with the ability to snoop the memory bus or take a platform's RAM chips at a precise time since enclave data and code in RAM are encrypted. However, unlike late launch, launching an enclave does not halt other processors on the system, so enclave code is potentially vulnerable to concurrent monitoring of its memory access patterns.

Chapter 4

Achieving forensic deniability with Lacuna

Many programs, even when using special modes meant to improve user privacy like web browsers' "private browsing" modes, leave evidence of user actions. The problem is that modern computer systems do not provide system-level support for privacy; as a program interacts with the OS and other applications, private data leaks into the rest of the system with no way to remove it.

Lacuna provides users with the ability to run a program in a "private session" that ensures that no traces from the program are left behind after the program has terminated. We consider an attacker who can observe all system state after a private session ends. Lacuna's guarantee, which we call **forensic deniability**, is that the attacker cannot figure out which program was used or recover information about actions taken in the program.

While prior work exists on erasing sensitive user data from a system, it does not apply a systematic approach to catch all traces. Work on secure memory deallocation [CPGR05] assumes that memory with sensitive data is deallocated, while we find examples to the contrary wherein long-lived servers (including the OS)

retain user secrets (§4.1). Furthermore, the PaX patch, a common implementation of secure deallocation for Linux [pax], does not apply it pervasively and leaves sensitive data, such network packets, in memory.

This work makes the following contributions:

- It describes multiple novel attacks wherein application secrets are recovered after application termination. These attacks are not prevented by methods in the literature for reducing data lifetime, and hence show new techniques are required.
- It introduces forensic deniability as a useful and achievable privacy guarantee. We argue that with forensic deniability as a privacy goal it is easier to obtain greater system usability.
- It details the design and implementation of Lacuna, a system that provides forensic deniability for a wide variety of applications. It identifies privacy of peripheral I/O as a key challenge to providing forensic deniability and shows how an abstraction it introduces, the **ephemeral channel**, solves this problem.
- It empirically evaluates the degree to which Lacuna meets its privacy and usability goals.

4.1 Motivation: Leaks from "private" browsing

As a motivating example, consider a user that wants to browse a protest website that is restricted by the government and contains a variety of different types of media. Figure 4.1 shows an example of such a website, which contains an audio player, pictures, and accompanying text. A user may want to be able to view this website in a way such that an attacker cannot determine the website contents (which may

Figure 4.1: A protest website with audio player, pictures, and text.

reveal sensitive aspects of the protest movement) and cannot even determine that the user has visited the website (which may be illegal to view).

We modeled a user's interaction with this website on a Linux desktop system and looked for ways in which information about this interaction leaks into the rest of the system. We used both private browsing mode and the PaX patch to test the limits of current systems. While our experiments were performed on a Linux system, we expect similar results on other OSes.

4.1.1 Graphical data

Multiple levels of system software retain portions of what is displayed on the screen during program execution. We examined the source code for a recent version of the Linux graphics stack and found the information leaks described below. We used X.org X server 1.10.6 (referred to as X below), Nouveau open-source NVIDIA GPU DRI2 module 0.0.16, kernel module 1.0.0, and Linux 3.3.0 with the PaX patch.

The X server allocates memory[1] as part of the EXA acceleration layer, a

[1] `exaPrepareAccessReg_mixed()` allocates memory for each pixel on the screen (file `exa/exa_migration_mixed.c`, line 203). The pointer to the memory is stored in the `Client` data structure for X's own X client and referenced from the global array of pointers to the `Client` data structures for all active X clients.

standard part of the modern X server architecture used by many open-source GPU drivers. EXA accelerates 2D graphics operations performed during screen updates when application windows are moved or their visibility changes. EXA uses memory allocated by the X server as a cache—for example, to cache the bitmap representation of window contents when part of the window is obscured. When an occluding window is relocated, the exposed part of the screen is recovered by fetching the bitmap from the EXA cache instead of redrawing the entire application window (assuming that the window's contents are unchanged). The cache is not invalidated when an application terminates, and is kept allocated until the last X client terminates. Typically, the last X client is an X window manager whose termination coincides with the termination of the X server itself.

The EXA subsystem cache contains desktop contents only for certain window managers that employ 2D acceleration, such as TWM and FVMW2. We also recovered window bitmaps from an X server without any window manager. With Xfce 4 and the Gnome/Unity environments, however, this memory buffer contains only a static desktop wallpaper image. Furthermore, we observed this leak when using the open-source Nouveau graphics driver deployed by all major Linux distributions, but not with the proprietary NVIDIA driver because the latter does not use the EXA cache.

Window contents of terminated applications can also be retrieved from kernel memory in a way that does not depend on X's user-space behavior. We examined the TTM module, which is a memory manager for the Direct Rendering Manager (DRM)[2] subsystem used by most modern open-source GPU drivers in Linux. The TTM module manages a DMA memory pool for transferring data between the host and GPU memories[3]. Scanning the pages in this memory pool reveals bitmaps rendered on the screen by previously terminated applications, including a QEMU

[2] `http://dri.freedesktop.org/wiki/DRM`

[3] See `drivers/gpu/drm/ttm/ttm_page_alloc_dma.c` in the Linux kernel source.

Virtual Machine (VM) and VNC (used for remote access to graphical desktops).

This technique works for the Gnome/Unity environment (the current Ubuntu default) and likely works for all window managers because all of them can work with a graphics driver using the TTM module. The lifetime of data recovered this way is measured in hours if the system is idle, but is sensitive to the churn rate of windows on the desktop and applications' behavior. For example, the display contents of a terminated VM remain in memory almost intact after running various desktop applications, such as terminal emulators and word processors, that do relatively little image rendering. Only about half of the contents remain after invoking a new VM instance, but some remnants survive all the way until the DMA memory pool is cleared as a result of the X server's termination or virtual console switch.

We also found a similar leak with the proprietary NVIDIA driver when displaying static images outside the QEMU VM. Its lifetime was limited to about 10 minutes. Without the driver's source code, however, we are unable to identify the exact reasons for the leak.

4.1.2 Audio data

Most popular Linux distributions use the PulseAudio server, which provides a uniform interface for advanced audio functions like mixing and resampling. We examined PulseAudio 1.1 in the following. PulseAudio uses shared memory segments of at most 64 MB to communicate with applications. These segments are allocated when applications create "PulseAudio streams" by calling `pa_simple_new` and `pa_stream_new`. If an application crashes or exits without freeing its segment via `pa_simple_free` or `pa_stream_free`[4], its audio output remains in PulseAudio's memory. PulseAudio lazily garbage-collects segments whose owners have exited, but only when a new shared segment is mapped.

[4]See `src/pulse/stream.c` and `src/pulse/simple.c` in the PulseAudio source.

Sound streams recovered from PulseAudio shared segments after the application terminated are noisy because the PulseAudio client library stores memory management metadata inline with stream contents in the same segment.[5] Nevertheless, we were able to recover up to six seconds of audio generated by Skype (sufficient to reveal sensitive information about the conversation and its participants) and the mplayer music player. In general, duration of the recovered audio depends on the sampling rates of the application and any input files.

4.1.3 System caches

Not all system memory caches are explicitly freed when no longer in use, thus secure deallocation is not sufficient for forensic deniability. For example, PaX leaves file data read from disk in the system buffer cache because those pages are not freed on program exit. Buffer cache pages compromise forensic deniability even for programs inside a VM. We ran LibreOffice in an Ubuntu 11.10 guest VM on a host without LibreOffice installed, then shut down the VM and dumped the host's physical memory. Examination of the memory image revealed symbol names from the `libi18nisolang1gcc3.so` library, disclosing (with the help of `apt-file`) that LibreOffice had run.

4.1.4 Network data

Contrary to the advice from [CPGR05], PaX does not clean `sk_buff` structures which store network packets. In general, PaX does not appear to eagerly erase any `kmem_cache` memory at all, which can completely compromise forensic deniability. For example, we visited websites with Google Chrome in private mode running inside a VM with NAT-mode networking on a PaX-enabled host. After closing Chrome and shutting down the VM, a physical memory dump revealed complete packets

[5] See `src/pulsecore/memblock.c` in the PulseAudio source.

with IP, TCP, and HTTP headers.

4.2 Goals

We see that typical applications can leak a lot of information into the rest of the system. Here we formulate a set of sensible goals for our system, Lacuna, that aims to prevent application information from leaking to an adversary.

4.2.1 Threat model and privacy goals

We would like to be able to ensure that application traces are unavailable even to a strong adversary. It is unlikely that we can defend against an attacker with concurrent OS-level access to the system on which private applications are run, because such an attacker can observe the memory of applications that we would like to keep private. Prior work [OST06, JS12] shows that even with less unprivileged access, an attacker with concurrent access to a system (i.e., running as some user-level process) can determine a significant amount of information about other processes running on the system. This prior work generally makes use of side-channels, which are public information that correlates with program activity, like cache activity or process memory footprint.

It is difficult to defeat side-channels since they originate from a variety of different sources of information that often appear innocuous. Rather than trying to eliminate a long list of side-channels and producing a system whose guarantees could be easily undone by a new side-channel, we opt instead for a threat model that does not allow an attacker to exploit most side-channels. One aspect common to most side-channels is that they involve measurement of a system over time while target code is running. Instead, we consider an adversary that only gains access to a system after a target private program has terminated. We refer to the ability to the ability to prevent such an adversary from learning information about the private

program's execution as forensic deniability. We use the term forensic deniability because with this guarantee a user can maintain plausible deniability of her actions in a private program in the face of a forensic examination of her system after the program has terminated.

Lacuna aims to prevent an attacker from learning even one bit of information about the execution of a private program, that is, the attacker should not be able to answer even yes-or-no questions about the execution. However, there are a few caveats to this guarantee. Firstly, Lacuna does not aim to make it impossible for the attacker to tell that Lacuna was used. Secondly, for reasons that will become clearer in Section 4.3, Lacuna will not prevent an attacker from knowing which peripherals were used by a private program. Finally, Lacuna does not prevent against the use of non-public hardware APIs to learn information. For example, if a network card has an interface known to the attacker to dump the complete contents of the past 1000 packets sent through it, then Lacuna will not prevent leaks through this interface.

Forensic deniability must be **coercion-resistant**: the user herself should not be able to recover any evidence from her private sessions. Lacuna does not persist secrets from one private session to another (e.g., a program in a Lacuna private session cannot save encrypted state to be reused during its next invocation). The attacker controls the host, and if a secret is kept by the user instead—e.g., as a password or in a hardware device—she can be coerced to open the persistent state.

Data leaks due to network traffic analyzed by an adversary are out of scope for Lacuna. It is worth noting that an adversary may not have access to an application's network traffic, as would be the case for malware infecting a user's machine after the application terminates. Users may need to use Lacuna in combination with techniques to hinder network traffic analysis, like Tor[6], depending on their privacy goals.

[6] `http://www.torproject.org`

4.2.2 Usability goals

In order for Lacuna to be useful, it is insufficient to have it just meet certain privacy goals. We have a number of usability goals in mind as well to minimize the inconvenience of using Lacuna on a system:

- **Support simultaneous private and non-private applications**

  Users should not have to interrupt their normal tasks to use private applications.

- **Support a wide variety of applications**

  Users should be able to run applications of their choosing privately, not merely specific applications like a web browser.

- **Incur performance cost only for private applications**

  Non-private applications will run exactly as before, with any system modifications only affecting private sessions. We refer to this as ensuring that privacy is "pay as you go".

- **Incur low performance cost even for private applications**

  If the performance impact of Lacuna is too great, then users may well opt to not use Lacuna to improve usability at the expense of privacy.

4.3 Design

The key challenge for Lacuna is to be able to use system services while ensuring that application secrets are not leaked in the process. First, we use virtualization as a way of isolating a private application from the rest of the system. However, private applications still need to be able to use system devices for display, storage, and other needs. Lacuna makes use of an abstraction called ephemeral channels to

provide private applications access to devices while ensuring that application secrets are not leaked.

4.3.1 Private process isolation

Modern applications often consist of a number of OS processes that communicate via OS Inter-Process Communication (IPC) mechanisms. These communications can contain application secrets, and so cannot be allowed to pass through complicated paths in the host OS or system services that can cause them to persist. As a result, Lacuna uses virtualization as a way of containing IPC: private applications are put into VMs using a hosted hypervisor (also called a Type II hypervisor), and all of their IPC is contained in guest VMs. A VM runs in the host OS as just another process, and then the in-memory state of the entire VM becomes easy to erase.

4.3.2 Ephemeral channels

Applications in private sessions cannot be completely cut off from the host while maintaining functionality. Private applications still need to use system peripherals to store files, display graphics, and so on.

There are two types of devices that Lacuna needs to support. First, storage devices are different than all others because storage devices do not interpret the data that they store . As a result, we can encrypt data bound for a storage device before it leaves a VM and decrypt it whenever it is requested by the VM again. Ensuring private access to storage devices is thus relatively straightforward, so we defer discussion of this class of devices to Section 4.5.

By contrast, devices that are not storage devices need to interpret the data that they are sent. For example, data sent to a graphics card needs to be interpreted by the graphics card to produce the proper image on the screen. As a result, the strategy we outlined for storage devices does not work here. Ephemeral channels

are meant to help with this type of device.

Ephemeral channels help ensure that there are no data left behind from peripheral paths that are readable by an attacker. One way of ensuring no data are left behind that are readable by an attacker is to leave no traces of sensitive data behind in the first place (outside of the private VM that can be erased). This can be accomplished any time that a guest can take full control over a system peripheral. Lacuna gives VMs control of system peripherals via peripheral component interconnect (PCI) device assignment. PCI device assignment requires an IOMMU to prevent a buggy VM from corrupting host memory with bad device requests.

Guest control of a device is sensible for some cases, like input peripherals, which do not need to be shared by multiple applications at the same time. However, for most cases, guest control of a device prevents multiplexing of the device by the system. Here, hardware virtualization support allows guest control of a device while still allowing the system to multiplex it. The goal of hardware virtualization support is to create several different hardware interfaces to a device that can be controlled by different drivers (e.g., one by a host driver, and others by drivers in guest OSes) at the same time, and where interactions are resolved in hardware. For example, a network interface card (NIC) with hardware virtualization support can present separate hardware NIC interfaces to different VMs and the packets sent by each VM are enqueued for transmission over some set of physical interfaces. In particular, **Single Root I/O Virtualization** (SR-IOV) technology allows a PCI device to appear as multiple PCI devices in hardware. When a guest controls leakage of private data by taking direct control of hardware, we refer to this as a **hardware ephemeral channel**. For hardware ephemeral channels, no host code manipulates sensitive data, so there is no opportunity for data leaks.

Unfortunately, not all classes of device have hardware virtualization support available, and hardware virtualization support is often restricted to high-end server

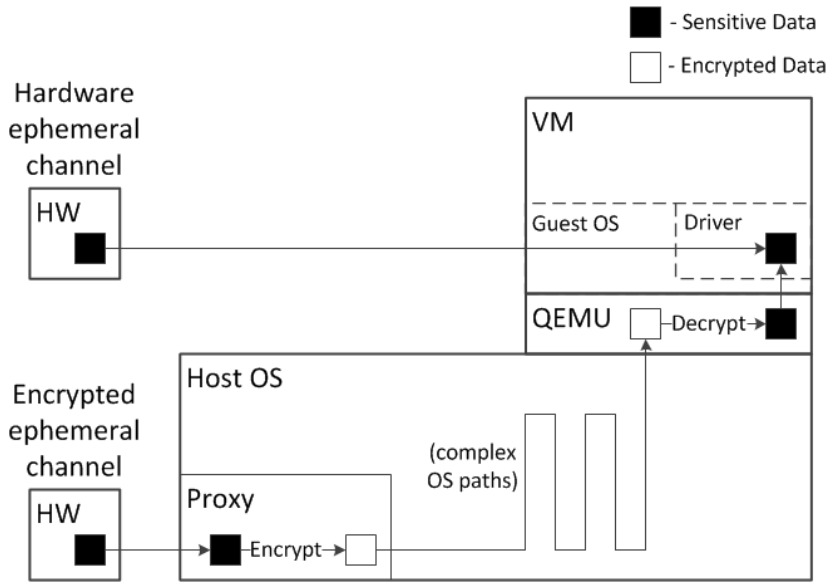

Figure 4.2: Overview of ephemeral channels. Sensitive data flow is shown for both types of channels. Hardware ephemeral channels connect guest system software directly to hardware, while encrypted ephemeral channels connect guest system software to small software proxies on the host or peripheral device. Black boxes represent unencrypted data, white boxes encrypted data.

components. As a result, it is important to be able to protect data even when hardware virtualization support is unavailable. We accomplish this with **encrypted ephemeral channels**. For an output channel, sensitive data is encrypted right before it leaves the hypervisor. We modify host device drivers to add a small **proxy** that decrypts data right before it is used by the target device. Input channels are similar, with encryption happening first in the proxy, followed by decryption in the hypervisor. Encryption keys for encrypted ephemeral channels are destroyed upon application exit, leaving all traces **cryptographically erased**, making them cryptographically hard to read [BL96]. Figure 4.2 shows an overview of the operation of the different types of ephemeral channel.

4.3.3 Side-channel mitigation

We cannot claim that Lacuna prevents all information leaks via side-channels. However, adopting forensic deniability as a privacy goal means that many side-channel attacks are prevented by construction as they require access to information acquired at a variety of times during program operation. Despite this, it is possible that the host system state available to an attacker after a private program runs contains information that inadvertently reveals some aspects of the private program's behavior. A significant class of system state that is available to an attacker is statistics that are collected by the host OS of a machine running Lacuna. In Section 4.7, we perform an empirical analysis of OS statistics in Linux. We discuss the few relevant statistics that we find and implement mitigations for any potential leaks we discover.

4.4 Design of ephemeral channels

Here we discuss design aspects of ephemeral channels for particular classes of device. The design of an encrypted ephemeral channel encompasses exactly what data needs to be encrypted, and where the endpoints of the encrypted ephemeral channel are to be placed. For hardware ephemeral channels, design includes specifying what hardware will be controlled by a guest.

4.4.1 Display devices

Display devices do not have hardware virtualization support, and so need to be accommodated by an encrypted ephemeral channel. A natural unit of data to encrypt for display devices is a frame of data before it is to be drawn: VMMs often make graphical data available in a virtual framebuffer within the hypervisor that is then forwarded to the graphics card[7]. However, it is less clear where the data should

[7]Since graphical data is first rendered in the hypervisor, Lacuna does not support graphics acceleration in VMs.

be decrypted; we do not want to decrypt data before it reaches caches in Linux graphics driver code, and some graphics drivers are closed-source and thus difficult to modify. Here we opt to support a method where it seems most likely that traces of decrypted graphics data will not escape: we use programmable graphics card support, often called GPGPU support, and decrypt graphics data on the graphics card itself. GPU memory, including where decryption occurs, is exclusively owned by the GPU and is not directly addressable from the CPU. After decryption, data can be directly rendered onto the screen via an OpenGL shader. Decryption on the GPU also allows easy multiplexing of the display with other programs, as the GPU can combine decrypted sensitive graphics data with that from non-private applications.

4.4.2 Audio devices

Lacuna encrypts the audio sent to virtual sound hardware in guest VMs. Lacuna multiplexes private application and non-private application audio via a new software mixer written into the host kernel that decrypts private application audio and mixes it into the non-private application audio. The result is sent to the host sound hardware.

4.4.3 USB input devices

Lacuna supports a wide variety of USB input devices, including keyboards and mice, which are critical for interaction with a system. Lacuna supports both hardware and encrypted ephemeral channels for USB devices as they have different advantages and disadvantages. Since for many types of input device input only needs to be sent to one application at a time, it is possible to create a hardware ephemeral channel in which the USB controllers (which are PCI devices) for these input devices become controlled by the guest VM. However, safe device assignment requires an IOMMU. Furthermore, if an input device is connected to a USB hub, then all the devices

on the hub would have to be assigned to the guest at the same time. Thus it is useful to also have encrypted ephemeral channels for USB devices in case hardware ephemeral channels cannot be used due to the above restrictions.

Encrypted ephemeral channels for USB devices work by switching the driver on the host that processes USB data to one which passes through data into the hypervisor. Unlike hardware ephemeral channels, a particular USB device, rather than a USB host controller which may be responsible for multiple USB ports, may be switched. When data are bound for the hypervisor, contents of USB Request Buffers (URBs) are encrypted as they leave the generic portion of the host USB driver (the portion that is not specific to the USB host controller and thus also the USB protocol version) and decrypted when they enter the private VM. Modifying only the generic portion of the host USB driver minimizes host code modifications. Both types of ephemeral channel allow the destination of USB data to be toggled back and forth between a private VM and the host at will.

4.4.4 Network devices

Network support is important for both usability and privacy. Some attacks that are within scope for Lacuna (e.g., malware infecting the host after the private session is over) do not control the network, but can learn private information from IP headers leaked by the VM.

Lacuna creates an ephemeral channel from the host NIC driver to the VMM where it delivers the packet to the virtual network card. This channel can be based on either encryption, or SR-IOV hardware. Encrypted ephemeral channels connect to the host in layer 2: each VM outputs Ethernet packets via a software TAP device, which connects to the NIC via a software bridge. Packet contents and headers for layer 3 (IP) and above are encrypted while they pass through the host. Hardware ephemeral channels based on SR-IOV network cards give a VM direct control over a

virtual network PCI device in the card hardware that multiplexes a single network connection.

To minimize the changes to specific device drivers, we encapsulate most routines for MAC registration and encryption/decryption in a generic, device-independent kernel module. This module checks whether a MAC address belongs to some VM and encrypts or decrypts a packet when needed.

4.5 Implementation

We implement Lacuna with the QEMU-KVM hypervisor. Our implementation is based on the Linux 3.0.0 kernel and QEMU 0.15.1. We ported the PaX kernel patch option `CONFIG_PAX_MEMORY_SANITIZE` to zero out pages of application data (e.g., pages from the memory of a private VM) as they are freed. The code for our implementation is available online at `https://github.com/ut-osa/lacuna`.

4.5.1 Encrypted ephemeral channels

To implement encrypted ephemeral channels, the kernel and programmable devices maintain **cryptographic contexts**, one for each direction of each device's logical communication channel (input from the device or output to the device). The Lacuna prototype provides kernel and GPU implementations. For symmetric encryption, kernel cryptographic contexts use the Linux kernel's cryptographic routines, while GPU contexts use our own implementation of AES. To establish a shared secret key for each context, Lacuna uses the key exchange portion of TLS 1.1. We ported the relevant parts of the PolarSSL[8] cryptographic library (SHA1, MD5, multi-precision integer support) to run in the kernel.

These contexts are managed from userspace via our `libprivcrypt` library. We modified the QEMU VMM to use `libprivcrypt`. On initialization, the VMM

[8] `http://www.polarssl.org`

creates cryptographic contexts in the kernel and GPU and establishes shared parameters (algorithm, IV, secret key), allowing it to encrypt data destined to these contexts and decrypt data originating from them. To encrypt and decrypt, `libprivcrypt` uses `libgcrypt`[9] or ported kernel code and Intel's AES-NI hardware encryption support.

When a private session terminates, even abnormally (i.e., from `SIGKILL` or crash), all cryptographic contexts associated with it are zeroed, including those on the GPU. This, along with zeroing of the VMM's memory, ensures that all data that has passed through the ephemeral channels is cryptographically erased.

4.5.2 Storage

Lacuna stores writes to storage separately from the initial VM image. There are two uses of storage to accommodate: writes to storage devices, and portions of private applications that are swapped to disk. Writes to storage are put into a **diffs file** where they are encrypted. The diffs file is used in combination with the initial VM image to construct the proper values for sector reads from the VM virtual block device.

Lacuna supports swapping of portions of private applications to disk. A new flag `CLONE_PRIVATE` is added to the `clone` system call. When this flag is set, the kernel allocates a private swap context, generates a random key, and protects the swap contents for that kernel thread. When an anonymous page is evicted from memory, the kernel checks the virtual memory segment metadata (VMA in Linux) to see whether the page is part of a private process. If so, the kernel allocates a scratch page to hold the encrypted data and allocates an entry in a radix tree to track the private swap context. The tree is indexed by the kernel's swap entry so that it can find the context on swap-in. Our implementation reuses much of the existing

[9] `http://www.gnu.org/software/libgcrypt/`

swap code path. To help distinguish private pages during normal swap cache clean up, we add an additional bit in the radix tree to indicate when a particular entry may be removed and which entries to purge during process cleanup.

Writes to storage are not the only storage operations to leave changes to system state. Reads from the initial VM image file cause portions of that file to be read into the host OS buffer cache. Portions of the VM image file in the host OS buffer cache must be cleared or they would reveal what portions of the image file were read, indicating which files on disk were used. However, we avoid a global performance hit for all applications by clearing only the portion of the buffer cache associated with the VM image file of the private application. We accomplish this with a flag `O_PRIVATE` that is passed to `open` when the VM image file is opened.

Note that the techniques for privately writing and reading from storage are all pay as you go; they do not hurt performance of non-private applications. Only writes from private applications are encrypted, and only cached data from reads in private applications is purged on application shutdown.

4.5.3 Ephemeral channels for specific device types

Display devices For display ephemeral channels, the VMM polls the frame buffer, and, upon each update, encrypts the buffer contents and transfers the encrypted data to GPU memory. Lacuna then invokes its CUDA routine[10] to decrypt the guest's frame buffer in the GPU, maps it onto an OpenGL texture, and renders it on the host's screen with an OpenGL shader. The implementation consists of 10 LOC in the QEMU UI module and SDL library, and an additional QEMU-linked library for rendering encrypted frame buffers, with 691 LOC of CPU code for GPU management and 725 lines of GPU decryption and rendering code.

[10]While our implementation uses CUDA and is compatible only with NVIDIA GPUs, similar functionality can be also implemented for AMD GPUs using OpenCL [ope].

Audio Lacuna provides output and input audio channels for each VM and a small (approximately 550 LOC) software mixer that directly interacts with the audio hardware's DMA buffer (§4.4.2). We modified the widely used Intel HD-Audio driver to work with the mixer, changing fewer than 50 lines of code. Note that "Intel HD-Audio" refers to an audio standard rather than specific hardware, so this driver works for both Intel and non-Intel controller chips.[11]

Lacuna can send sound input to multiple VMs. For output (playback), the host kernel keeps a separate buffer for each VM to write raw encrypted audio. Linux's audio drivers provide a callback to update the pointers indicating where the hardware should fetch the samples from or where the application (e.g., PulseAudio) should write the samples. Our mixer takes advantage of this mechanism: upon pointer updates, samples in each encrypted output buffer are decrypted, copied to the DMA buffer between the old and new application pointers, and then zeroed in the encrypted output buffer. The DMA buffer is erased when the VM terminates.

USB Lacuna's encrypted ephemeral channel support for USB encrypts data in USB Report Buffers (URBs) as they are passed to system software from hardware control. Packets destined for the guest and the host may be interspersed, so Lacuna tracks which URBs it should encrypt by associating cryptographic contexts with USB device endpoints. An endpoint is one side of a logical channel between a device and the host controller; communication between a single device and the controller involves multiple endpoints.

We added 118 lines to the `usbcore` driver to encrypt URBs associated with cryptographic contexts as they are returned from hardware-specific host controller drivers. These URBs are decrypted in the VMM's virtual USB host controller before they are passed on to the guest USB subsystem. Our prototype has been tested only with USB 1.1 and 2.0 devices, but should work with USB 3.0. It does

[11] `http://www.kernel.org/doc/Documentation/sound/alsa/HD-Audio.txt`

not support USB mass storage devices and less common USB device classes (such as USB audio), but adding this support should require a reasonably small effort because our mechanism is largely agnostic to the contents of URBs.

When the user moves her mouse over a private VM's display and presses "Left-Control+Left-Alt", Lacuna engages a user-level USB driver, `devio`, to redirect the keyboard and mouse ports to the VMM.[12] The title bar of the VMM window indicates whether the keyboard and mouse input are redirected through ephemeral channels. When they are not redirected, the Lacuna VMM refuses input to avoid accidental leaks.

The same key combination toggles control of the keyboard and mouse back to the host. The VMM's virtual hardware detects the key combination by understanding the position of modifier key status in data packets common to USB HID devices. With a hardware ephemeral USB channel, detecting the combination requires guest OS modification (119 LOC). With hardware channels, errors that freeze the guest currently leave no way of restoring input to the host, but we believe that this limitation is not intrinsic to our architecture (e.g., the host could run a guest watchdog).

Network Lacuna VMs are networked in layer 2, and entire layer 3 packets are encrypted. Each VM is assigned its own MAC address controlled by our `privnet` module, which uses cryptographic contexts to do encryption in Intel's `e1000e` driver with 30 lines of glue code.

Outgoing packets are encrypted by the VMM. The host kernel places them in an `sk_buff`, the Linux network packet data structure. The driver maps each `sk_buff` to a DMA address for the NIC to fetch; right before it tells the NIC to fetch, it queries `privnet` whether the packets in the transmit queue come from a Lacuna

[12]The unmodified QEMU already uses this key combination for acquiring exclusive control of the keyboard, but it takes events from the X server and does not provide forensic deniability.

VM, and, if so, decrypts them in place. The driver zeroes `sk_buffs` on receipt of a "transmission complete" interrupt. Because decryption takes place right before the packets are written into hardware buffers, packets from a VM cannot be received by the host (and vice versa) at a local address.

For incoming packets, as soon as the driver receives the interrupt informing it that packets are transferred from the NIC to the kernel via DMA, it encrypts the packets destined for the Lacuna VMs. Encryption is done in place and overwrites the original packets. Decryption takes place in the VMM.

Although the layer 2 (Ethernet) header is not encrypted, its EtherType, an indicator of the layer 3 protocol it is encapsulating, is modified to prevent a checksum failure: a constant is added to it so that the resulting value is not recognized by the Linux kernel during encryption, and subtracted again during decryption. As a side benefit, this bypasses host IP packet processing, improving performance (§4.6.6).

While Lacuna currently reveals layer 2 packet headers, it could likely be modified to hide this information. The current implementation allows Linux's bridge code to be used without modification. Layer 2 address information is more important for mobile devices, wherein it can potentially reveal location information [ZDH+13].

4.6 Evaluation

We evaluate both the privacy properties and performance of Lacuna. We run all benchmarks except switch latency on a Dell Studio XPS 8100 with a dual-core 3.2 GHz Intel Core i5 CPU, 12 GB of RAM, an NVIDIA GeForce GTX 470, and an Intel Gigabit CT PCI-E NIC, running Ubuntu 10.04 desktop edition. The swap partition is on a 7200 RPM, 250GB hard drive with an 8MB cache. Switch latency to and from the private environment is benchmarked on a Lenovo T510 with a dual-core 2.67 GHz Core i7 CPU and 8GB of RAM, running Ubuntu 12.04 desktop edition. The Lenovo has a Microsoft USB keyboard (vendor/device ID `045e:0730`)

and mouse (vendor/device ID `045e:00cb`), as well as an IOMMU, which is required for the PCI assignment-based ephemeral channel. Both machines have AES-NI and use it for all AES encryption except where indicated. The guest VM runs Ubuntu 10.04 desktop edition, with 2 GB RAM and the Linux 3.0.0 kernel.

4.6.1 Validating privacy protection

Following the methodology of prior work [CPGR05], we inject 8-byte "tokens" into the display, audio, USB, network, and swap subsystems, then examine physical RAM for these tokens afterwards. Without Lacuna (but with QEMU and PaX), the tokens are present after the applications exit. With Lacuna, no tokens are found after the private session terminates. This experiment is not sufficient to prove forensic deniability, but it demonstrates that Lacuna plugs at least the known leaks.

One subtlety occurred with the video driver. We use the Nouveau open-source driver for the test without the display ephemeral channel and the NVIDIA proprietary driver for the test with the channel, because the NVIDIA driver is required for CUDA execution. To inject tokens, we run a program that displays a static bitmap inside a VM. With the ephemeral channel, no tokens from the bitmap are found after VM termination. Without the channel, we detect the tokens[13] after the VM termination—but not if we use the proprietary driver. This driver does leak data from other applications, but not from QEMU. Without the driver source code, we are unable to identify the causes for this observed behavior.

4.6.2 Measuring data exposure

To estimate the potential exposure of private-session data, Table 4.1 shows the size of driver code that handles it unencrypted. The graphics data is not exposed at all

[13]The tokens are slightly modified due to the display format conversion in QEMU, which adds a zero after every third byte.

Subsystem	LOC
Graphics	0 (725 CUDA)
Sound	200 (out), 108 (in)
USB	414
Network	208

Table 4.1: Lines of code (LOC) external to QEMU that handle unencrypted data. Line counts were determined by manual examination of data paths from interrupt handler to encryption using SLOCCount [Whe01].

	Video	Browser	LibreOffice
QEMU	32.2 ± 7.4	25.9 ± 1.3	8.1 ± 1.2
Lacuna	49.7±0.3 (Δ17.5)	46.2±1.5 (Δ20.3)	21.1±0.6 (Δ13.0)

Table 4.2: CPU utilization (%) for benchmarks with encrypted network, video, and sound channels. The performance of all benchmarks on Lacuna is identical to unmodified QEMU. The increase in CPU utilization is marked with Δ. Averages are calculated over 5 trials with standard deviations as shown.

because it is encrypted by the VMM, which then transfers it directly to the GPU memory and invokes the Lacuna implementation of the CUDA decryption and GL rendering routines on the GPU (implemented in 725 lines of code).

4.6.3 Full-system performance

We measure the overhead of Lacuna on a number of full-system tasks: watching a 854 × 480 video with mplayer across the network, browsing the Alexa top 20 websites, and using LibreOffice, a full-featured office suite, to create a document with 2,994 characters and 32 images. We sample CPU utilization at 1 second intervals. To avoid the effects of VM boot and to capture application activity, we omit the first 15 samples and report an average of the remaining samples.

The execution times of the video and LibreOffice benchmarks on Lacuna are within 1% of base QEMU. The performance of the browser benchmark varies due

to network conditions, but there is no difference in average execution time. The display—redrawn upon every contents change at the maximum rate of 63 frames/s—is not perceptibly sluggish in any of the benchmarks when using the encrypted GPU channel. Table 4.2 shows the CPU utilization of the workloads running on Lacuna and on unmodified QEMU.

4.6.4 Clean-up time

The clean-up after a private VM terminates is comprised of five concurrent tasks:

Clear VM memory Lacuna uses PaX to zero VM memory when the VM process exits and frees its address space. To measure the worst-case window of vulnerability, we run a program in the VM that allocates all 2 GB of available VM memory, then send the VMM a signal to terminate it and measure the time between signal delivery and process exit. Linux does not optimize process exit, often rescheduling a process during its death. In 10 trials, unmodified Linux required 2.1 ± 0.1 s to terminate a VM. The worst case we measured for Lacuna (USB passthrough mode with keyboard and mouse) is 2.5 ± 0.2 s.

Clear buffered disk image The Lacuna VMM opens disk image files with a privacy flag so that the kernel can securely deallocate all buffer cache pages for those files when the VMM exits without affecting the page cache contents for concurrent, non-private programs. Only clean pages need to be deallocated and zeroed because a private Lacuna session does not persist the modified disk image. This operation takes 0.111 ± 0.002 s in our video benchmark.

Clear swap cache memory Lacuna securely deallocates freed swap cache pages. A benchmark program allocates 12 GB of memory to force the system to swap, writing out an average of 677.8 ± 33.4 MB to the swap partition. However, because

the swap cache is used only for transient pages (those that have not completely swapped out or swapped in), the average number of memory pages remaining in the swap cache at program termination is only 50 or so (200KB). Clearing this data takes only 68.9 ± 44.6 μs.

Clear kernel stacks Lacuna zeroes the VMM's kernel stack, and also notifies and waits for each CPU to zero their interrupt and exception stacks. In our video benchmark, this takes 15.8 ± 1.15 μs.

Clear GPU memory Lacuna has a GPU memory scrubber which uses the CUDA API to allocate all available GPU memory and overwrites it with zeros. A similar GPU memory scrubbing technique is used in NCSA clusters.[14] Our scrubber zeroed 1.5GB of GPU memory in 0.170 ± 0.005 s.

4.6.5 Switch time

Table 4.3 shows how long it takes to switch into a private session and how the switch time depends on the number of devices and type of the ephemeral channel(s).

A significant portion of the switch time when using encrypted USB passthrough results from disabling the peripheral USB drivers (0.8 ± 0.1 s for keyboard alone, 1.0 ± 0.2 s for keyboard and mouse) to allow `devio` to take control. This time is affected by the number of USB devices that must be disconnected. Interestingly, it is also affected by the complexity of the USB device: keyboards with media keys often show up as two devices on the same interface, which necessitates disconnecting two instances of the peripheral driver.

We noticed an interaction between the guest USB drivers and QEMU that significantly affects switch time. Linux's USB drivers perform two device resets during device initialization. These resets in the guest are particularly costly be-

[14] `http://www.ncsa.illinois.edu/AboutUs/Directorates/ISL/software.html`

Channel type	Switch time (s)
USB passthrough	
keyboard only	1.4 ± 0.2
keyboard + mouse	2.3 ± 0.2
PCI assignment	
keyboard only	2.4 ± 0.2
keyboard + mouse	3.8 ± 0.2

Table 4.3: Switch time for different numbers of peripherals and ephemeral channel types (averages over 5 trials).

cause each results in QEMU performing an unnecessary (since QEMU has already performed a reset) unbinding of the `devio` driver and the reattachment of the device's initial `usbhid` driver. Eliminating QEMU's action upon these resets cuts this component of switch time by two thirds.

4.6.6 Network performance

We benchmark network performance between a private VM and a gateway connected by a switch: netperf and ping results are in Table 4.6, scp and netcat in Table 4.4.

There are several types of netperf tests. `TCP_STREAM` uses bulk transfer to measure throughput, the other types measure latency. `TCP_RR` (Request/Response) tests the TCP request/respond rate, not including connection establishment. `TCP_CC` (Connect/Close) measures how fast the pair of systems can open and close a connection. `TCP_CRR` (Connect/Request/Response) combines a connection with a request/response transaction. Ping measures round-trip time.

Neither latency, nor throughput is significantly affected when using AES-NI, except for a dip in throughput for receiving 300 byte packets. For small packets, performance with AES-NI encryption is slightly better than without encryption because encrypted packets bypass some host processing (since they appear to be of an unknown packet type). To verify this explanation, we did an additional experiment

File size	Transfer time (s)		
	scp	Ephemeral + netcat	
		AES-NI	Software
400MB	8.41	4.28	8.92
800MB	14.96	8.55	17.50

Table 4.4: netcat and scp test results.

	No encryption	AES-NI	PCI assignment
CPU util (%)	27.7±2.7	36.0±1.6	14.7±4.2

Table 4.5: CPU utilization for TAP networking without encryption, with encryption, and using PCI assignment when transferring an 800MB file via netcat. The throughput is 794±3 Mbps for all runs.

where we changed the EtherType of each packet without encrypting the content. We measured over 120Mbps throughput when sending 30-byte packets, which is about a 40% improvement. Software encryption achieves roughly half the throughput of AES-NI.

We also compare the file transfer time for netcat using an encrypted ephemeral channel and scp without using ephemeral channels (Table 4.4). File transfer with AES-NI encryption is twice as fast as software-only scp. These results also validate that our software encryption performance is comparable to scp.

Table 4.5 shows the measurements of CPU utilization when transferring an 800MB file using no encryption, AES-NI, and PCI assignment. This benchmark was run on a quad-core 3.6 GHz Dell OptiPlex 980 with 8 GB of RAM and an Intel Gigabit ET NIC.

While all methods have nearly identical throughput, PCI assignment significantly lowers CPU utilization.

Test type	Netperf throughput (Mbps)					
	TCP_STREAM send			TCP_STREAM recv		
Packet size	1400	300	30	1400	300	30
QEMU	788	516	86	827	829	226
Lacuna	769	419	89	819	820	231
HW encryption	2%	19%	-4%	1%	1%	-2%
Lacuna	373	242	54	373	370	168
SW encryption	53%	53%	37%	55%	55%	26%

Test type	Netperf latency^{-1} (Trans./s)			Ping		
	TCP_RR	TCP_CC	TCP_CRR	Round-trip time (ms)		
Packet size	1	1	1	1400	300	30
QEMU	5452	2530	2260	0.327	0.251	0.237
Lacuna	5312	2487	2180	0.366	0.253	0.219
HW encryption	3%	2%	4%	12%	1%	-8%
Lacuna	5206	2264	2029	0.408	0.277	0.244
SW encryption	5%	11%	10%	25%	10%	0.3%

Table 4.6: Netperf and ping test results for unmodified QEMU and Lacuna with hardware-assisted (HW) AES-NI encryption and software (SW) encryption. Reductions in performance are shown as percentages, where negative values indicate better performance than QEMU.

4.6.7 Audio latency

To measure output latency from the VM to the sound DMA buffer, we sent a known sequence through the sound channel and measured host timestamps for send and receive. The results are in Table 4.7, showing that the latency of the encrypted ephemeral audio channel is smaller than that of PulseAudio.

	Latency (ms)
Ephemeral channel	23.5 ± 8.6
PulseAudio	57.5 ± 11.3

Table 4.7: Audio latency comparison (averages over 10 trials).

There are counterbalancing effects at play here. The encrypted channel incurs additional computational overhead, but bypasses PulseAudio mixing and short-

ens the path from the VM to host audio DMA buffer.

4.6.8 Swap performance

Figure 4.3 compares the performance of plain Linux, Lacuna without encrypted swap, Lacuna with encrypted swap, and `dm-crypt`-protected swap. In the first three cases, a non-private process performs similarly to Linux. Our encrypted swap differs from standard swap in two ways whose effects are shown in the graph: it allocates a scratch page and bookkeeping for every private page swapped and encrypts the swapped-out pages.

`dm-crypt` has particularly bad performance in this microbenchmark. We verified that our installation of `dm-crypt` on ext4 adds, on average, 5% overhead when running file-system benchmarks such as IOzone[15].

4.6.9 Scalability

Table 4.8 shows the performance of multiple concurrent Lacuna VMs, all executing the LibreOffice workload in a private session. The performance overhead of one VM is negligible, but increases with eight concurrent VMs because the CPU is overcommitted. Our attempt to run more than eight VMs produced an unexplained CUDA error. Non-private VMs scale to 24 instances before Linux's out-of-memory killer starts killing them.

4.7 Study of statistics-based side channels

In this section we analyze possible side-channel information leaks in Lacuna from statistics that the Linux kernel collects. Statistics in the host Linux kernel represent an important class of potential information leak because the kernel collects a wide variety of statistics about many different subsystems. There is no central API in

[15] `http://www.iozone.org/`

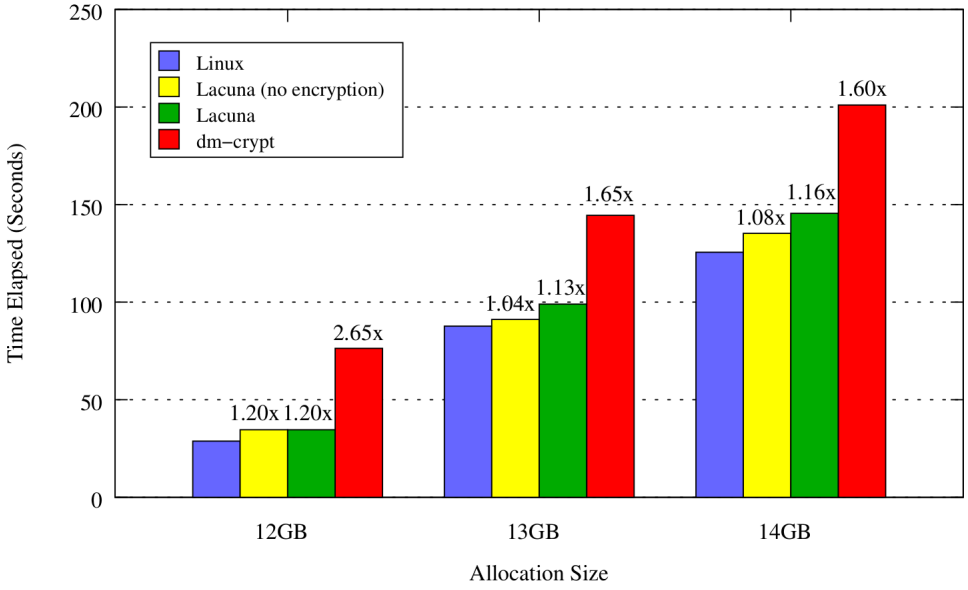

Figure 4.3: Average elapsed time for swap microbenchmarks (lower is better). This benchmark allocates a buffer using malloc, touches each page in pseudorandom order, and reads the pages of the buffer in order to check correctness. The numbers above the bars indicate relative slowdown relative to Linux.

the Linux kernel for statistics collection, so it is not easy to find all of the statistics that are collected. However, many statistics from the Linux kernel are made available to userspace via a small number of interfaces. Thus, we can cover a significant number of statistics by examining only a small number of interfaces. This section describes the statistics we found and how to mitigate resulting side-channel leaks. The resulting analysis does not exhaust statistics-based side-channel attacks, but provides some sense of the possibilities and raises the bar for an attacker by eliminating many of them. We re-emphasize here that Lacuna's threat model limits an attacker's access to kernel statistics to whatever remains after a private session terminates.

Setup	Running Time (s)
1 QEMU VM	189.3 ± 0.1
1 Lacuna VM	190.6 ± 0.1 (1.01×)
8 QEMU VMs	191.6 ± 0.1
8 Lacuna VMs	277.3 ± 1.1 (1.45×)

Table 4.8: Time to complete the LibreOffice workload under contention from other VMs (averages over 5 trials).

4.7.1 Finding statistics

The Linux kernel provides a number of statistics for the system through the `proc`[16] and `sysfs`[17] filesystems. While some information about what is available through `proc` and `sysfs` can be gleaned by reading documentation, source code analysis has the power to show more conclusively what can be found through these interfaces. There is a limited interface provided by the kernel to add entries into `proc` and `sysfs`[18]. As a result, we can use `grep` to search the Linux source code for calls to the interfaces for `proc` and `sysfs`.

`proc` We identified the relevant kernel interface for `proc` by examining `fs/proc/generic.c` and `include/linux/proc_fs.h` in the Linux kernel source. Using `grep` we identified 744 different uses of the `proc` interface in the kernel source. Note that some of these instances occur in functions that are themselves called in multiple functions. One important exception to this interface is the code that provides the `readdir` and `lookup` functions for the root directory of `proc`[19]. The `readdir` and `lookup` functions for the root `proc` directory add per-process statistics into `proc`.

[16] See `Documentation/filesystems/proc.txt` in the Linux kernel source and `http://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html` for some documentation on what is available in the `proc` filesystem.

[17] `Documentation/filesystems/sysfs.txt` in the Linux kernel source documents `sysfs` at a high level, and `Documentation/ABI` describes some of the contents.

[18] Due to the monolithic nature of Linux, it is possible for kernel code to ignore this interface and put entries into `proc` and `sysfs` through some other means, but we found no evidence of this.

[19] See `fs/proc/root.c` in the Linux kernel source.

While there are too many uses of the `proc` to exhaustively examine every one, we ensured that we covered every `proc` directory we found in the associated documentation and every per-process statistic. Beyond this, we focused on entries we believe to be the most widely applicable, which are those that are not specific to particular hardware.

`sysfs` We identified the relevant kernel interface for `sysfs` by examining `include/linux/sysfs.h` and `include/linux/kobject.h`. Using `grep` we identified 534 different uses of the `sysfs` interface in kernel source. We covered every documented `sysfs` interface, and again focused on interfaces that are not specific to particular hardware. Ultimately, in terms of statistics that reveal information, what we found are also reflected in `proc`.

4.7.2 Classifying statistics

In this section, we categorize kernel statistics into distinct types to help describe the statistics we found (§4.7.3) and how we mitigate information leaks from them (§4.7.4). Statistics that the kernel keeps that reflect behavior during private sessions can be divided into three categories depending on the scope of data on which they depend:

- **Since boot statistics**

  The kernel keeps a number of statistics that depend on the complete system behavior since boot, like the number of interrupts received for particular interrupt vectors. These statistics can provide information about the occurrences in a private session because they can bound system behavior during that session. For example, the number of interrupts received for a device since the last time a system was booted is an upper bound for the number of interrupts received for that device during a private session that has occurred since the

last system boot. However, if a system has seen a variety of use since last boot, then these statistics are less reflective of the actions taken in a private session.

- **Private session statistics**

  Some kernel statistics track actions for exactly the duration of a private session. For example, the kernel tracks the amount of disk I/O made per-process. Kernel statistics are often stored with the objects to which they pertain, and as a result, the kernel often frees memory containing these statistics after a private session ends. However, since the kernel does not securely deallocate all memory that it frees, it may be possible to retrieve these statistics after private session termination.

- **Partial private session statistics**

  It would be possible for the kernel to keep statistics that apply to only part of a private session. For example, if the kernel kept the average CPU utilization over a series of two second intervals for the past five minutes, then it would potentially be possible to build a profile of the CPU use for a program in a private session over time, which could be identifying.

4.7.3 Found statistics

We now categorize the statistics we found that most directly reveal information about private sessions by the type of information they reveal. We specify pieces of paths that should be filled in depending on the context with `<` and `>`, for example, `<pid>` indicates a particular process's numerical *pid*. We refer to paths in the `proc` and `sysfs` filesystems in their default locations below `/proc` and `/sys` respectively.

Session duration `/proc/uptime` is a since boot statistic that reveals the total amount of time the system has been online, which can both place a limit on the

duration of a private session and suggest how long a private session might have lasted. `/proc/loadavg` gives a measure of the average load on the system over the past 1, 5, and 15 minutes. This could be used to try and approximate the length of a session (since CPU use is likely higher during this period) and in this sense could be considered a private session statistic. However, the measure provided by `/proc/loadavg` is very coarse-grained.

`/proc/<pid>/stat` contains the time at which process *pid* was started. This time is put in a `struct task_struct` which is not zeroed when it is freed.

Session CPU use `/proc/loadavg` might also be used to estimate the CPU use of the tail end of actions in a private session, here acting as a partial private session statistic. Again, this is a very coarse-grained measure and system load could always be due to other processes besides a private session (though low load does indicate low load in a private session).

`/proc/<pid>/stat` contains per-process CPU use information, and `/proc/stat` contains per-CPU CPU use information which includes both total CPU use and a separate statistic for the amount of CPU time spent in a guest VM. Both of these could be used to obtain the amount of CPU time spent in a guest VM, though the latter is a since boot statistic that could be perturbed by other non-private VM use.

Session device use `/proc/stat` and `/proc/interrupts` give since boot interrupt counts for devices. These counts give a coarse-grained view of device use.

For storage devices, `/proc/diskstats` gives since boot per-disk-partition I/O, and `/proc/<pid>/io` provides per-process total I/O to block devices. Similarly, `/sys/block/<drive name>/<drive name><partition number>/stat` gives since boot per-disk-partition I/O. For `ext4` filesystems, `/sys/fs/ext4/<partition name>/session_write_kbytes` also provides since boot per-disk-partition I/O.

Statistic category	Modifications made
Duration	• Do not record process start times • Zero system load averages • Randomize initial system monotonic time • Randomize initial Time Stamp Counter
CPU use	• Zero system load averages • Zero per-CPU use statistics • Zero per-process CPU use on process exit
General device use	• Zero system interrupt counts
Storage device use	• Change per-process accounting code to ignore private sessions • Zero system per-partition block I/O statistics
Network device use	• Zero freed network interface statistics • Modify `e1000e` to clear driver-specific statistics structure

Table 4.9: Summary of changes made to mitigate information leaks from statistics.

For network devices, `/proc/net/dev` provides per-interface total I/O, which provides both since boot total network I/O and due to the fact that a separate TAP interface is created for each private session could potentially be used to find per-private session total network I/O. The same statistics are available in `sysfs` under `/sys/class/net/<interface name>/statistics/<statistic name>`.

4.7.4 Mitigating statistics-based side-channels

We now describe mitigations for the side-channels described in Section 4.7.3. We prototype these mitigations and test that they do not crash the host OS. We found no error messages in the system log (from system applications or daemons that could use the modified information) or kernel log that appear to be related to the changes. Our changes are summarized in Table 4.9.

Session duration The uptime from `/proc/uptime` is calculated from the Linux kernel's "monotonic time" value. Monotonic time is defined as zero at system boot time. Thus to obscure the uptime of the system we need to increase the system's monotonic time. The monotonic time is used in a variety of system timers, so increasing it during system operation is not an option. (Empirically, doing so caused a number of system errors.) Instead, we can set the monotonic time to a random value on boot, allowing continuity for system timers while obscuring uptime. However, at the point in the boot process where a random value is required, the Linux random number generator has not had time to collect any randomness. As a result, our current implementation only demonstrates that given a source of boot-time randomness we can obscure the uptime. However, we expect that there are many possible sources of boot-time randomness, like Intel's `RDRAND` instruction or a small TPM driver (which could be kept small since it needs a very limited set of TPM functionality).

We take one further step to conceal system uptime: When processes start in Linux, the monotonic time at which they begin is recorded. We eliminate this recording, so all processes appear to have begun at monotonic time zero. If we did not, since many processes begin near system boot (which has a newly-assigned random monotonic time value), it would be easy to approximate the initial random value given to the monotonic time. However, eliminating the recording of process start time can potentially hamper system functionality. It may be possible to create a compromise solution wherein processes that have start times within a certain range (a few seconds) of the initial random value are set to zero start time while those beyond that range keep their real start time. This way, the wide variety of processes that start on boot would not reveal the true system uptime, while users can still infer how long ago their processes were started for system management purposes.

While most of our work with Lacuna addresses leaks through system soft-

ware, there are counters that directly leak system uptime that it is worth attempting to mitigate. In particular, Intel x86 CPUs provide the Time Stamp Counter (TSC), which is a 64-bit value per physical processor that is initialized to zero on system reset and counts up at a constant rate while the system is not put to sleep[20]. However, the system-accessible TSC value is controllable by setting a register per core that affects TSC readings from that core. So, similarly to uptime (and with the same caveat about randomness), the TSC can be randomized on boot. In order to keep system functionality, we have to keep the per core TSC values close to each other. Under normal circumstances the per core TSC readings (even for cores on other processors) stay relatively close to each other; Linux verifies this and will refuse to use the TSC as a timing source if they do not (so if per-core TSC views do not stay in sync we can just randomize them all independently without affecting system functionality). We implement logic that calculates a random TSC increment and then schedules a task on all cores to read the TSC value and immediately (in the next instruction) to set that core's register to the read value plus the increment. Linux's TSC sanity test still passed after this modification.

Our TSC randomization does not work on more recent Intel hardware, as it introduces a new per-core register `IA32_TSC_ADJUST` that tracks changes made to the TSC. If the per-core view of the TSC is adjusted by a value x, then the value of this register in also adjusted by x, which means that any TSC adjustment can be detected.

Additionally, we note that there are other system timers that might reveal the system uptime. The widespread Advanced Configuration and Power Interface specification mandates a Power Management Timer that is can be up to 32-bits wide and counts up at a rate of about 3.6 MHz [acp10]. It appears not to be resettable in hardware without putting the system to sleep (which would harm system usability).

[20] In older Intel CPUs the TSC rate is affected by CPU clock scaling [int14b].

However, with this size and counting rate, the timer will overflow about every 20 minutes, so it can only provide accurate uptime for sessions less than this length. Furthermore, while the counter can be 32-bits wide, it is only required to be 24-bits wide, which would lead to an overflow time of 4.7 seconds, making it unhelpful for tracking uptime. It seems that future hardware modifications may be necessary to fully obscure system uptime.

We zero out the running average for `/proc/loadavg` on private session completion. The load averages thus reflect nothing about the private session and are usable during private sessions and far after private sessions.

Process start times from `/proc/<pid>/stat` are taken care of by the mitigations for uptime above.

Session CPU use Our handling of `/proc/loadavg` has already been discussed.

We zero out the per-CPU usage statistics in `/proc/stat`, again allowing forward use from the point of zeroing. The implementation is a little more involved than `/proc/loadavg` due to needing to clear counters per CPU, but it is still relatively straightforward.

We zero the per-process CPU usage, as is reported by `/proc/<pid>/stat` on process exit for private processes (as marked by `CLONE_PRIVATE` discussed in Section 4.5.2). This allows accounting code to correctly account for non-private processes while ensuring that statistics are not available for an attacker that can search through memory. Note that we clear statistics as soon as a process exits rather than wait for its `task_struct` to be released since the latter involves an extra delay until the process is reaped.

Session device use We zero out interrupt counts in `/proc/stat` and `/proc/interrupts`.

For storage devices, we modify the per-process accounting code to not collect

I/O statistics for private processes. This allows accounting from `/proc/<pid>/io` to work for non-private processes. We zero out block statistics used by `/proc/diskstats` to allow them to be used going forward from the end of a private session. We determined that the `sysfs` disk statistics paths all use the same underlying counters as `/proc/diskstats`. The `sysfs` path under `ext4` reads and stores the partition I/O counter on filesystem mount and reads the partition I/O counter again when the `sysfs` file is read. Zeroing the partition I/O counter ensures the `sysfs` file only reveals the value of the partition I/O counter at filesystem mount (which is entirely unrelated to private sessions).

For network devices, the source of what is read through `/proc/net/dev` and the equivalent `sysfs` interface are the result of the kernel function `dev_get_stats`. This function can trigger a driver-specific function in some cases. For the TAP devices that are created purely for the duration of a Lacuna session, there is no special function provided by the TAP driver, so the statistics are stored in the associated `struct net_device`. We added code that clears the statistics from every freed `struct net_device`. This should not have any functionality impact as it is an error to depend on the contents of freed memory, and the freeing of `struct net_device`s happens only on rare events like the powering down of network interfaces, so the performance impact is likely to be limited.

The `e1000e` network driver that Lacuna uses for an encrypted ephemeral channel does provide its own driver-specific statistics function which retrieves statistics from a driver-internal structure. We modify the `e1000e` driver so that the since-boot statistics present from this driver are cleared on private session termination. We verified that clearing these statistics while the card is running produces no negative side-effects.

4.7.5 Usability effects of leak mitigation

System statistics are presumably collected because they are useful, so any tampering with these statistics may harm system functionality. While we saw no immediate errors when implementing mitigation mechanisms, some of the mechanisms we introduced cause incorrect readings of system activity to appear. The goal for any mitigation mechanisms is to minimize the impact of any errors introduced into system statistics while still maintaining forensic deniability.

Since boot statistics are often single counters for the entire system, so changing these to erase information about private sessions does induce errors in future readings. However, all of the mitigations that we introduce merely change the values of these counters at a specific point. From that point onward, the counters are faithful to future activity. As a result, statistics that measure the difference of counters over time (e.g., the current rate of disk activity as reported by `iostat -d 2`) or gradually become less influenced by past activity (e.g., the system load averages) will still be useful.

As mentioned in Section 4.7.4, there is less impact to erasing or not collecting private session statistics once a private session has terminated, as these statistics tend to pertain to objects that no longer exist on the system (e.g., the QEMU process for the private session or a TAP interface created for a private session).

Chapter 5

Conclusion

We presented two systems, Suliban and Lacuna, that concretely illustrate the trade-offs that can be made between privacy and usability in systems that protect program privacy. Suliban shows the cost of defeating most known techniques for software analysis: limiting to computation-only code and delivering platform-specific encrypted payloads to only platforms that can be proven to protect privacy. In developing Suliban, we also uncovered important flaws in the way the TPM is currently used and managed by software whose correction is necessary to maintain security guarantees in the TPM and future trusted computing equipment. Our work on Lacuna illustrates new classes of privacy problems inherent in peripheral use and provides the ephemeral channel abstraction to solve them. Lacuna's notion of forensic deniability will be a useful design goal for privacy-enhancing systems going forward as it has a coherent stance against modern techniques for extracting program secrets and admits systems with reasonable usability.

Bibliography

[ABJB10] Gaurav Aggrawal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security*, 2010.

[acp10] Advanced Configuration and Power Interface Specification, Revision 4.0a, 2010.

[AGJS13] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[aKK04] Hyang ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security*, 2004.

[amd10] AMD64 Architecture Programmer's Manual, Volume 2: System Programming, 2010.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. *Journal of the ACM*, 59(2), 2012.

[BGK+14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and

Amit Sahai. Protecting Obfuscation against Algebraic Attacks. In *EUROCRYPT*, 2014.

[BHL+08] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection*. Springer, 2008.

[BHS03] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating secure crash information. In *USENIX Security*, 2003.

[bit09] BitLocker Drive Encryption Step-by-Step Guide for Windows 7, 2009. `http://technet.microsoft.com/en-us/library/dd835565(WS.10).aspx`.

[BL96] Dan Boneh and Richard Lipton. A revocable backup system. In *USENIX Security*, 1996.

[Bla94] Matt Blaze. A cryptographic file system for UNIX. In *CCS*, 1994.

[BRSS11] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: trading privacy for application functionality on smartphones. In *HotMobile*, 2011.

[BSW08] John Bethencourt, Dawn Song, and Brent Waters. Analysis-Resistant Malware. In *NDSS*, 2008.

[BWLP09] Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security*, 2009.

[CHGL06] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A Safety-Oriented Platform for Web Applications. In *S&P*, 2006.

[CHK+08] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *HotSec*, 2008.

[CJ03] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *USENIX Security*, 2003.

[CN04] Mark D. Corner and Brian D. Noble. Zero-interaction authentication. In *MOBICOM*, 2004.

[CNZ+11] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *SOSP*, 2011.

[CPG+04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, 2004.

[CPGR05] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *USENIX Security*, 2005.

[CPKS09] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *CCS*, 2009.

[CSK+10] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying Dormant Functionality in Malware Programs. In *S&P*, 2010.

[Dan10] George Danezis. Traffic analysis of the HTTP protocol over TLS. http://research.microsoft.com/en-us/um/people/gdane/papers/TLSanon.pdf, 2010.

[DDN00] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[DHWW11] Alan M. Dunn, Owen S. Hofmann, Brent Waters, and Emmett Witchel. Cloaking Malware with the Trusted Platform Module. In *USENIX Security*, 2011.

[DLJ+12] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In *OSDI*, 2012.

[ecr] eCryptfs. https://launchpad.net/ecryptfs.

[efs] The encrypting file system. http://technet.microsoft.com/en-us/library/cc700811.aspx.

[Fel] Edward Felten. USACM Policy Statement on DRM. https://freedom-to-tinker.com/blog/felten/usacm-policy-statement-drm/.

[Fin09] Hal Finney. PrivacyCA, 2009. http://www.privacyca.com.

[GAB10] Paolo Gasti, Giuseppe Ateniese, and Marina Blanton. Deniable Cloud Storage: Sharing Files via Public-key Deniability. In *WPES*, 2010.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits. In *Foundations of Computer Science*, 2013.

[GPCR04] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data Lifetime is a Systems Problem. In *ACM SIGOPS European Workshop*, 2004.

[Gut96] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *USENIX Security*, 1996.

[Gut01] Peter Gutmann. Data remanence in semiconductor devices. In *USENIX Security*, 2001.

[HcCS09] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-scale Malware Indexing Using Function-call Graphs. In *CCS*, 2009.

[HHJ+11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*, 2011.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Cal, Ariel J. Feldman, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security*, 2008.

[HX07] Keith Harrison and Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *DSN*, 2007.

[int09] Intel Trusted Execution Technology (Intel TXT) MLE Developer's Guide, 2009.

[int13] Software Guard Extensions Programming Reference, 2013.

[int14a] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2, 2014.

[int14b] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, 2014.

[JS12] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *S&P*, 2012.

[KAMC11] Jayanthkumar Kannan, Gautam Altekar, Petros Maniatis, and Byung-Gon Chun. Making programs forget: Enforcing lifetime for sensitive data. In *HotOS*, 2011.

[Kau07] Bernhard Kauer. OSLO: Improving the security of trusted computing. In *USENIX Security*, 2007.

[KCK+09] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security*, 2009.

[KF08] Kimmo Kasslin and Elia Florio. Your Computer is Now Stoned (...Again!). The Rise of the MBR Rootkits, 2008. `http://www.f-secure.com/weblog/archives/Kasslin-Florio-VB2008.pdf`.

[KHKK10] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *S&P*, 2010.

[KP03] Douglas Knowles and Frederic Perriott. W32.Blaster.Worm, 2003. `http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99`.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security*, 2012.

[kra08] Owning Kraken Zombies, a Detailed Discussion, 2008. `http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies`.

[KSP05] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing Trusted Platform Communication. In *ECRYPT Workshop, CRASH CRyptographic Advances in Secure Hardware*, 2005.

[KZB+90] Paul A. Karger, Mary Ellen Zurko, Douglas W. Benin, Andrew H. Mason, and Clifford E. Kahn. A VMM Security Kernel for the VAX Architecture. In *S&P*, 1990.

[Lam09] Butler Lampson. Usable security: How to get it. *Communications of the ACM*, 52(11), November 2009.

[LSWK11] Byunghee Lee, Kyungho Son, Dongho Won, and Seungjoo Kim. Secure data deletion for USB flash memory. *Journal of Information Science and Engineering*, 2011.

[LYH+10] Jaeheung Lee, Sangho Yi, Junyoung Heo, Hyungbae Park, Sung Y. Shin, and Yookun Cho. An efficient secure deletion scheme for flash file systems. *Journal of Information Science and Engineering*, 2010.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[MAF+11] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do you know where your data are? Secure data capsules for deployable data protection. In *HotOS*, 2011.

[MDS12] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *ISCA*, 2012.

[MK99] Andrew D. McDonald and Markus G. Kuhn. StegFS: A steganographic file system for Linux. In *IH*, 1999.

[Moh11] Mohammad Mannan and Beom Heyn Kim and Afshar Ganjali and David Lie. Unicorn: Two-factor attestation for data security. In *CCS*, 2011.

[MPP+08] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[MPR09] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe Passage for Passwords and Other Sensitive Data. In *NDSS*, 2009.

[MRHM10] Aleksandr Matrosov, Eugene Rodionov, David Harley, and Juraj Malcho. Stuxnet Under the Microscope, 2010. Revision 1.2.

[myd04] MyDoom.C Analysis, 2004. `http://www.secureworks.com/research/threats/mydoom-c/`.

[Naz09] Jose Nazario. The Conficker Cabal Announced, 2009. `http://asert.arbornetworks.com/2009/02/the-conficker-cabal-announced/`.

[OMRK13] Kaan Onarlioglu, Collin Mulliner, William K. Robertson, and Engin

Kirda. Privexec: Private execution as an operating system service. In *S&P*, 2013.

[ope] OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[OR11] Jon Oberheide and Dan Rosenberg. Stackjacking Your Way to grsecurity/PaX Bypass. `http://jon.oberheide.org/files/stackjacking-hes11.pdf`, 2011.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of Cryptographers' Track at the RSA Conference*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[pax] Homepage of the PaX team. `http://pax.grsecurity.net`.

[PBH+05] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubbleeld, and Aviel D. Rubin. Secure deletion for a versioning file system. In *FAST*, 2005.

[PCJD07] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A Semantics-based Approach to Malware Detection. In *POPL*, 2007.

[Per05] Radia Perlman. The Ephemerizer: Making Data Disappear. `http://www.filibeto.org/~aduritz/truetrue/smli_tr-2005-140.pdf`, 2005.

[PJ10] Matt Piotrowski and Anthony D. Joseph. Virtics: A system for privilege separation of legacy desktop applications. Technical Report UCB/EECS-2010-70, University of California, Berkeley, 2010.

[Pos07] Andre Post. W32.Storm.Worm, 2007. http://www.symantec.com/security_response/writeup.jsp?docid=2001-060615-1534-99.

[Pro00] Niels Provos. Encrypting virtual memory. In *USENIX Security*, 2000.

[PSY09] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points, 2009. http://mtc.sri.com/Conficker/.

[PTZ03] HweeHwa Pang, Kian-Lee Tan, and Xuan Zhou. StegFS: A steganographic file system. In *ICDE*, 2003.

[QM12] Zhiyun Qian and Z. Morley Mao. Off-Path TCP Sequence Number Inference Attack - How Firewall Middleboxes Reduce Security. In *S&P*, 2012.

[qub] Qubes. http://qubes-os.org/.

[RCB12] Jeff Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security*, 2012.

[SEVS04] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm fingerprinting. In *OSDI*, 2004.

[SML07] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to Privacy in the Forensic Analysis of Database Systems. In *SIGMOD*, 2007.

[SSW+02] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted Web browsing traffic. In *S&P*, 2002.

[SW10] Steven Swanson and Michael Wei. SAFE: Fast, Verifiable Sanitization for SSDs. Technical Report cs2011-0963, UCSD, 2010.

[SWT01] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security*, 2001.

[SWZS12] Kun Sun, Jiang Wang, Fengwei Zhang, and Angelos Stavrou. SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. In *NDSS*, 2012.

[TAB+12] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *OSDI*, 2012.

[Tar10] Christopher Tarnovsky. Hacking the Smartcard Chip. In *Black Hat*, 2010.

[TMK10] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.

[Tru07] Trusted Computing Group. *TPM Main Specification, Level 2, Version 1.2*, 2007.

[Tru13] Trusted Computing Group. *Trusted Platform Module Library Specification, Family "2.0", Level 00*, 2013.

[VDS11] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *ACM Cloud Computing Security Workshop*, 2011.

[Vie01] John Viega. Protecting sensitive data in memory. `http://www.ibm.com/developerworks/library/s-data.html?n-s-311`, 2001.

[VPQP09] Amit Vasudevan, Bryan Parno, Ning Qu, and Adrian Perrig. Lockdown: A Safe and Practical Environment for Security Applications. Technical Report CMU-CyLab-09-011, CMU, 2009.

[wav10] Trusted Computing Whitepaper. Wave Systems Corporation, 2010. `http://www.wave.com/collateral/Trusted_Computing_White_Paper.pdf`.

[WGSS11] Michael Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, 2011.

[Whe01] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2001.

[WRT09] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology. Invisible Things Lab, 2009.

[YMC07] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[YSE+07] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *CCS*, 2007.

[YY04] Adam Young and Moti Yung. Malicious Cryptography: Exposing Cryptovirology. Wiley, 2004.

[ZBS98] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.

[ZDH+13] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *CCS*, 2013.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[ZW09] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security*, 2009.

Vita

Alan Mark Dunn graduated from Montgomery Blair High School in Silver Spring, Maryland in 2001. He graduated from the Massachusetts Institute of Technology in 2005 with B.S. degrees in Physics and Mathematics. He entered a graduate program in Physics at Duke University, graduating with an M.A. in 2008 before realizing that Computer Science is the one true way and entering the doctoral program in the Department of Computer Science at the University of Texas at Austin in 2009. He received an M.S. in Computer Science in 2012.

Contact email: `adunn@cs.utexas.edu`

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.