Towards a Trustworthy Thin Terminal for Securing Enterprise Networks

by

Evan J. Frenn

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2013

APPROVED:

_____

Dr. Craig A. Shue, Major Advisor

_____

Dr. Krishna K. Venkatasubramanian, Reader

_____

Dr. Craig E. Wills, Head of Department

Abstract

Organizations have many employees that lack the technical knowledge to securely operate their machines. These users may open malicious email attachments/links or install unverified software such as P2P programs. These actions introduce significant risk to an organizations network since they allow attackers to exploit the trust and access given to a client machine. However, system administrators currently lack the control of client machines needed to prevent these security risks.

A possible solution to address this issue lies in attestation. With respect to computer science, attestation is the ability of a machine to prove its current state. This capability can be used by client machines to remotely attest to their state, which can be used by other machines in the network when making trust decisions. Previous research in this area has focused on the use of a static root of trust (RoT), requiring the use of a chain of trust over the entire software stack. We would argue this approach is limited in feasibility, because it requires an understanding and evaluation of all the previous states of a machine. With the use of late launch, a dynamic root of trust introduced in the Trusted Platform Module (TPM) v1.2 specification, the required chain of trust is drastically shortened, minimizing the previous states of a machine that must be evaluated. This reduced chain of trust may allow a dynamic RoT to address the limitations of a static RoT.

We are implementing a client terminal service that utilizes late launch to attest to its execution. Further, the minimal functional requirements of the service facilitate strong software verification. The goal in designing this service is not to increase the security of the network, but rather to push the functionality, and therefore the security risks and responsibilities, of client machines to the networks servers. In doing so, we create a platform that can more easily be administered by those individuals best equipped to do so with the expectation that this will lead to better security practices.

Through the use of late launch and remote attestation in our terminal service, the system administrators have a strong guarantee the clients connecting to their system are secure and can therefore focus their efforts on securing the server architecture. This effectively addresses our motivating problem as it forces user actions to occur under the control of system administrators.

Acknowledgements

I would sincerely like to thank my research advisor, Professor Craig Shue, for his support and guidance throughout my education at WPI. His advice and assistance on my research, coursework, and general education were greatly appreciated. Specifically, he provided insightful and supportive criticism on my thesis and I am truly thankful for the time and effort he has given to supporting my education.

I wish to thank all my teachers for their help and support. In particular, I would like to thank Professor Krishna Venkatasubramanian, my thesis reader, for his time and valuable effort. I would like to thank Dr. Nathanael Paul for his enthusiasm and introduction into the field covered in my thesis. I would also like to thank my fellow students in the ALAS lab for their intellectually stimulating conversations and thoroughly enjoyed working with all of them. Lastly, I would like to thank my family and friends for their support in accomplishing this thesis. Without the help of all those mentioned, this work would not be possible.

Contents

Glossary

AC module	Authenticated Code module.
ACPI	Advanced Configuration and Power-management Interface.
AIK	Attestation Identity Key.
APIC	Advanced Programmable Interrupt Controller.
BSP	bootstrap processor.
BYOD	Bring Your Own Device.
CoT	Chain of Trust.
CPL	current privilege level.
DMA	Direct Memory Access.
DMAR	DMA Remapping.
DRTM	Dynamic Root of Trust for Measurement.
EK	Endorsement Key.
FCS	Frame Check Sequence.
GDT	Global Descriptor Table.
I/O APIC	IO APIC.
IDT	Interrupt Descriptor Table.
ILP	Initiating Logical Processor.
IM	Itermediate Driver.
ISR	Interrupt Service Routine.
KBC	keyboard controller.
KBE	keyboard encoder.
LOC	lines of code.
Local APIC	Local APIC.
LPC bus	Low Pin Count bus.

MLE	Measured Launch Environment.
MSR	model-specific register.
MTM	Mobile Trusted Module.
NDIS	Network Driver Interface Specification.
NIC	Network Interface Card.
NV memory	Non-Volatile Memory.
NW	Normal World.
OIC	Other Interrupt Controller.
OS	Operating System.
PAL	Piece of Application Logic.
PALs	Pieces of Application Logic.
PCH	Platform Controller Hub.
PCIe	PCI Express.
PCR	Platform Configuration Register.
PIC	Programmable Interrupt Controller.
PKI	Public Key Encryption.
PMR	Protected Memory Region.
RA	Remote Attestation.
RCBA	Root Complex Base Address.
RLP	Responding Logical Processor.
RoT	Root of Trust.
RTM	Root of Trust for Measurement.
RTR	Root of Trust for Reporting.
RTS	Root of Trust for Storage.
SMI	System Management Interrupt.
SMM	System Management Mode.
SMP	Symmetric Multiprocessing.
SMRAM	System Management RAM.
SMT	Symmetric Multi-Threading.

SRK	Storage Root Key.
SRTM	Static Root of Trust for Measurement.
STM	SMM Transfer Monitor.
SW	Secure World.
TC	Trusted Computing.
TCB	Trusted Computing Base.
TCG	Trusted Computing Group.
TEE	Trusted Execution Environment.
TLR	Trusted Language Runtime.
TOCTOU	Time Of Check, Time Of Use.
TP	Trusted Path.
TPM	Trusted Platform Module.
TTT	Trusted Thin Terminal.
TXT	Trusted Execution Technology.
VGA	Video Graphics Array.
VM	virtual machine.
VMM	Virtual Machine Monitor.
VT-x	Intel Virtualization Technology.
vTPM	virtual TPM.

1 INTRODUCTION

A fundamental problem with respect to computer security lies in the disparity between the trust and trustworthiness of today's systems. Currently available commodity computers are designed to use Operating Systems (OSs) that require trust from any application running on the system. This trust is inherent to the privilege hierarchy of x86 systems, allowing any software executing in ring 0, namely the kernel, unrestricted access to any application running in ring 3, namely user applications. We formally define trust in the context of computer security as the dependence of A on B, such that the actions of B can influence the security goals of A. As a brief example, the confidentiality of any password input for an application, such as an email account or VPN client, is implicitly a security goal. Any application requiring password input from the user is required to place trust in the operating system as it provides access to user input and can directly affect the confidentiality of the password.

While trust in the OS is intrinsic to the design of today's prominent systems, these systems are generally lacking in trustworthiness. We define trustworthiness as the ability of the previously mentioned entity B to ensure the security goals desired by A. There are several design characteristics common to today's prominent operating systems that inhibit their trustworthiness. One of the leading factors to OS's low assurance is their large code bases. For example, Linux 3.6 has roughly 16 million lines of code (LOC) [1] and while Microsoft has not released the size of Windows Vista, 7, or 8, the last released size of Windows XP was roughly 45 million LOC [2]. The extremely large code base of both operating systems is generally caused by over privileging many of their provided features and make it impractical to formally verify the system with regards to security. As an example of improper use of privilege domains commonly employed, device drivers generally do not need the full capabilities that come with running in kernel mode including complete access to system memory. They are, however, usually run within ring 0, needlessly increasing the code base that must be trusted, termed the Trusted Computing Base (TCB).

A desirable feature currently lacking from commodity systems is the ability of software to authenticate itself to remote parties. This manifests in a lack of trustworthiness that an application can place in data received from a remote party, either inducing input validation and sanitization requirements or creating possible vulnerabilities. As an example taken from [3], game servers are not able to identify whether network input is coming from a valid version of client software or a version corrupted by a cheating user and must therefore either take precautions to validate client input or allow the user to cheat.

Further complicating the lack of trustworthiness of popular operating systems is the ease

with which they can be misconfigured by untrained users. A simple example of misconfuration by a large percentage of users is the use of weak or default passwords. In a study performed by Weir *et al.* [4], a dataset of more than 32 million leaked passwords was analyzed. Utilizing a freely available password cracking tool and a commonly used dictionary, the researchers were able to crack more than a third of the passwords. A more recent study, published anonymously since the methods used were not legal, performed a network mapping of the entire IPv4 address space and found roughly 1.2 million devices with default or no password required for authentication [5]. As a followup example, anti-virus producer McAfee found that nearly a fifth of PCs do not have anti-virus software installed [6]. Anti-virus software provides easy-to-use protection from known malware with reputable products available for free such as AVG [7], Avast [8], and Malwarebytes [9] and yet a significant portion of devices are left unprotected. We believe the failure of the general public to properly secure their devices is in fact a failure of the design of the system with regards to security and not a fault of the owner. This belief guides our design of an authentication scheme outlined in the Future Works section.

In a corporate setting, the inability of users to properly secure devices indicate this responsibility should fall on IT professionals. However, the increasing trend of businesses to allow Bring Your Own Device (BYOD) policies, in which an employee utilizes a personal device for work-related matters, runs in opposition to IT control of device security. A recent study performed by Cisco of roughly 600 U.S. business found that 95% of employers allowed employees to utilized their personal devices for work related matters [10]. A separate poll found that 31% of employees utilized public Wi-Fi access points when connecting to their company's network [11]. While ideally there is a vetting process prior to allowing personal devices to access corporate resources, IT professionals do not have full control of these devices and may have minimal or no access to them.

We aim to address the security issues created by acceptance of BYOD policies by corporate organizations. Specifically, we intend to create the Trusted Thin Terminal (TTT), a tool designed to allow a BYOD policy while concurrently meeting the following goals:

1. **IT Control:** Push security responsibilities to skilled professionals
2. **Increase Trustworthiness:** Create a *trustworthy* thin client, mitigating the previously outlined trust issues related to today's operating systems
3. **Authentication:** Utilize available authentication techniques to allow the trustworthy thin client to authenticate itself

Our first goal of IT control logically follows from the inability of users to properly secure devices as outlined above. The structure of a thin terminal design, in which the processing

of user actions fall on the server, effectively addresses our first goal. As an example, if the user is requesting a corporate document from a remotely connected enterprise server, our TTT will be tasked with simply forwarding the request to the server. Processing of the file request will fall on the server which is controlled by IT, thereby allowing them to correctly configure the server with respect to the confidentiality goals of corporate documents, among other security goals of the corporation.

The motivation for our second goal of increasing trustworthiness stems from several different sources. As was described above, the use case for a thin terminal application pushes many of it's security requirements to the remotely connected server, which provides the ability to address our first goal. It is also motivated by our desire to extract the operating system from the TCB of the TTT. By removing reliance on the OS, we require the TTT to be self sufficient. While we are removing hardware abstraction and support of the OS, we would still like our application to provide similar functionality and a thin terminal service lends itself well to this purpose since it is generally required only to forward user I/O to the server and display server responses. It is important to note that we only intend to remove reliance on the OS on the client side; we do not address corporate server reliance on an operating system. However, we note the server will be run by IT professionals with the knowledge and ability to address this problem.

While we are removing reliance on the operating system, we also intend to provide functionality that was previously lacking as exemplified by our goal of authentication. The ability of the TTT to authenticate itself, and more importantly, provide proof to a remote system that its integrity is sound and has not been compromised by malware provides a strong assurance of its trustworthiness. In order to provide the necessary authentication measure, we rely on the Trusted Platform Module (TPM) and a process termed Remote Attestation (RA), both of which are described in depth in the ensuing background section.

We envision a use case for the TTT in which an employee requests access to the corporate network using her personal device. IT would then be able to provide the TTT as a normal application to the employee, who would use it to connect to the enterprise network. Using a trustworthy authentication method and removing reliance on the OS, the TTT provides the intended server high assurance of the trustworthiness of the remote client and allows it to then make decisions based on its trust in the TTT.

The ensuing section (Section 2) provides an introduction to Trusted Computing (TC) including the TPMs and a complimentary mobile approach termed ARM TrustZone security extension to give the reader an understanding of the underlying concepts of this work. Section 3 provides an in depth review of related work as well as a comparison to the TTT. Section 4 presents an overview of the TTT design and provides an adversary model, as well

known limitations of the underlying architecture. Section 5 describes the TTT architecture, partitioned by the provided functionality. Section 6 present possible avenues of future work and Section 7 concludes.

2 BACKGROUND

This section provides the reader with necessary background information to better understand the concepts applied in this work.

2.1 Trusted Platform Module

The Trusted Platform Module (TPM) is a specification and implementation of a secure tamper proof integrated circuit created by the Trusted Computing Group (TCG), a consortium of roughly 110 technology related companies including AMD, Cisco, IBM, Intel, and Microsoft [12]. The TPM is designed to provide generation of cryptographic keys, secure storage, remote attestation, and a pseudo random number generator. The current TPM specification is version 1.2b, implementations of which are generally designed for x86 machines in which the TPM is located on the Low Pin Count bus (LPC bus), a low-bandwidth bus connected to the southbridge on most motherboards. In this work, we make an assumption of the ubiquity of the TPM, which can be found on a large number of devices including Dell, HP, Acer, Lenovo, and Toshiba machines [13]. The TPM is also required on all machines purchased by both the U.S. Army and Department of Defense. [14, 15]

2.1.1 Remote Attestation

The TPM's ability to provide a platform for Remote Attestation (RA) is its most intriguing feature and one which has garnered significant research into the TPM's security implications. **Attestation**, as a security primitive, is the ability of a machine to make a claim to an appraiser about the properties of a target by supplying evidence that supports that claim [16]. Remote Attestation, as the name implies, is the ability to make a claim to a *remote party.* RA is accomplished through the designation of a TPM as multiple Roots of Trust with each root trusted with a specific responsibility. A **Root of Trust (RoT)**, as defined by the Trusted Computing Group (TCG), is a component which must behave in the expected manner, because its misbehavior cannot be detected. Effectively, a Root of Trust (RoT) is an entity in which trust must be placed and which allows the ability to extend the trustworthiness of the RoT to other entities.

2.1.2 Root of Trust for Measurement

The TPM acts as a Root of Trust for Measurement (RTM), allowing metrics to be collected in regards to a machine's software integrity. The TPM implements a Root of Trust for Measurement (RTM) by allowing for *measurements* to be stored in special registers located directly on the TPM, termed Platform Configuration Registers (PCRs). A **measurement**, with respect to a TPM, is a SHA-1 cryptographic hash of an intended target, usually an executable binary on the system. The measurement hash is *extended* into one of the TPM's PCRs by taking the SHA-1 of the measurement concatenated with the previous value of the PCR i.e. $PCR_i \leftarrow$SHA-1$(m|PCR_i^{old})$. PCR values may only be extended in this manner and the static PCRs,designated PCRs 0-16, may only be reset to 0 on system restart. The tightly controlled interface in which PCR values can be changed provides a strong platform for extending the trustworthiness of the TPM as an RTM to software executing on the central processor of the system. Previous versions of the TPM specification required the TPM to contain at least 16 PCRs, while v1.2 requires a minimum of 24 PCRs with PCRs 17-23 utilized as dynamic PRCs in the late launch process.

2.1.3 Root of Trust for Storage

The TPM acts as a Root of Trust for Storage (RTS), an RoT whose behavior is well defined for the specific task of providing secure storage. The Root of Trust for Storage (RTS) accomplishes this task the the use of a glssrk, whose private key is protected by its restriction to Non-Volatile Memory (NV memory) memory located directly on the IC. The Storage Root Key (SRK), at a minimum, is a 2048 bit RSA key generated when TPM ownership is initially configured. The SRK can be used to enable secure key storage by encrypting the private portion of any storage or identity key generated by the TPM. Unlike the SRK, these keys will be partially encrypted and outputted as a key blob whose storage responsibilities will be external to the TPM. A key blob can be loaded onto the TPM, in which its private key will be decrypted using the SRK as it's root parent and stored in volatile memory for subsequent use. A more generic Root of Trust for Storage (RTS) is the TPM's ability to *seal* data, in which the SRK or descendant storage key can be used to encrypt data. Similar to secure key storage, storage responsibilities of the encrypted data blob are external to the TPM. Decryption of sealed data is tied to platform metrics including designed PCR values, effectively limiting decryption of the data to a known system state. The connection to platform metrics can be disabled to allow for simple encryption and decryption of data using keys secured by the TPM. Due to the performance overhead of Public Key Encryption (PKI), it is a well accepted practice to only encrypt or seal a symmetric key and instead

utilize a symmetric cipher for secure data storage. This practice effectively extends the trustworthiness of the RTS to the symmetric key.

2.1.4 Root of Trust for Reporting

The Root of Trust for Reporting (RTR) is the ability of a TPM to securely report to a remote party the data protected by the TPM's RTS. Primarily this consists of reporting PCR values whose integrity relies on the RTM. The ability of a TPM to provide authenticated reporting relies on the creation and storage of an **Endorsement Key (EK)**, an immutable RSA key generated by the TPM manufacturer. The EK is utilized to identify and authenticate information from an individual TPM. This ability is extended to the Attestation Identity Key (AIK), which is subsequently used in any signing command issued to the TPM. The Root of Trust for Reporting (RTR) in concert with the RTM, both of which rely on the RTM provide the basis for previously mentioned capability of remote attestation.

2.1.5 Late Launch

Late launch, a recent addition to the TPM as of v1.2, allows for the creation of a Dynamic Root of Trust for Measurement (DRTM). The previous technique of a Static Root of Trust for Measurement (SRTM) ties the known state of a PCR to system boot in which PCRs 0-16 are reset to zero, effectively binding system integrity to its history since reboot. The Dynamic Root of Trust for Measurement (DRTM) significantly reduces complexity by allowing PCRs 17-23 to be reset to a known state following the issuance of a late launch command. As an overview of the DRTM, the dynamic PCRs are first reset to a known state and a code module is provided to the late launch procedure. This code module is subsequently measured and extended into PCR 17 and provided an environment for secure execution. The ability of late launch to dynamically reset specific PCRs to a known state and follow a hardware enforced procedure for loading, measuring, and executing a code segment effectively removes analysis and processing complexities of a system's history since reboot found in the Static Root of Trust for Measurement (SRTM).

2.1.6 Intel Trusted Execution Technology

Both AMD and Intel CPUs provide hardware support for the setup and execution of a late launch environment. While there are a few design differences between AMD's Secure Virtual Machine (SVM) Technology and Intel's Trusted Execution Technology (TXT), the focus of this paper will be on the later as the variations are inconsequential to our intended work. Intel Trusted Execution Technology (TXT) relies on two hardware extensions to execute a code

module in a late launch environment, a procedure it terms Measured Launch Environment (MLE). TXT utilizes Safer Mode Extensions (SMX), which introduced a specialized leafed instruction called *GETSEC[SENTER]* to launch an Measured Launch Environment (MLE). It subsequently relies on Intel Virtualization Technology (VT-x) to protect the MLE from corruption.

As an overview of the MLE launching procedure, the *SENTER* instruction will first reset all dynamic PCRs to zero. It will then issue a synchronizing operation to the Responding Logical Processor (RLP). The RLPs are those processors which are not designated as the Initiating Logical Processor (ILP), the processor in which the SENTER instruction originated. The Responding Logical Processors (RLPs) will initialize the processor to a known safe state and wait to rejoin the MLE. The Initiating Logical Processor (ILP) will then measure and load an Authenticated Code module (AC module), which has been digitally signed by the chipset vendor and will only be allowed to execute once it's authenticity is verified. The measurement for the Authenticated Code module (AC module) will be extended into PCR 17. The AC module is tasked with configuring the system into a known secure state as well as measuring and extending the MLE into PCR 18. The MLE code is then executed in a secure environment in which it can remotely attest to its execution and integrity utilizing the DRTM and RTR. It is important to note that the integrity of the late launch procedure is protected throughout its execution by disabling interrupts before the SENTER instruction is executed and protecting the AC module and MLE memory regions from corruption using Intel Virtualization Technology (VT-x). Once the MLE receives execution of the machine, it can re-enable both interrupts and the previously waiting RLPs.

2.2 ARM TrustZone

This section is dedicated to providing a brief introduction to ARM TrustZone Security extension. We provide this introduction to give conceptual background information to another viable approach in comparison to TPMs that may provide an avenue for future work. TrustZone is a trusted platform integrated into specific ARM processors for mobile devices. It consists of hardware capabilities providing a foundation for software functionality. The following sections will outline the hardware and software capabilities of TrustZone individually.

2.2.1 TrustZone Hardware

The TrustZone hardware capabilities are built upon the creation of a duplicate of the ring 0 and ring 3 modes of a traditional ARM processor. This effectively creates two hardware partitions on the processor, each consisting of a kernel and user mode. Following ARM

nomenclature, the newly created partitions are referred to as the Normal World (NW) and Secure World (SW). The NW is tasked with handling all legacy and non-sensitive code while the SW is provided for security sensitive applications. In a similar manner to the relation between ring 0 and ring 3 of each individual world, the NW is unable to preempt the SW. TrustZone extends the security state of the processor to system peripherals by adding an extra bit to the system bus, termed the Non-Secure or NS bit. The current value of the NS bit, hardware enforced by the processor to indicate its current world, is utilized by system peripherals to protect SW resources from NW access. Peripherals that provide SW protection include a secure display, user input, and system memory and storage.

2.2.2 TrustZone Software

The TrustZone specific software is referred to as the Secure Middleware and is tasked with providing similar functionality to a hypervisor in that it regulates system configurations for transitions to and from the SW user space to the NW. There is a standard available through GlobalPlatform for a Trusted Execution Environment (TEE), which relies on underlying hardware to provide execution protection between a secure and non-secure environment. TrustZone hardware is commonly used to provide the underlying hardware requirement of the TEE in which the Secure Middleware is an implementation of the TEE APIs standardized by GlobalPlatform. Available implementations of the Secure Middleware that conforms to the TEE API standard include Texas Instruments' M-Shield Mobile Security Technology solution [17], Giesecke & Devrient's Mobicore [18], and Sierraware's SierraVisor [19].

3 RELATED WORK

In this section, previous works related to attestation, isolation, client-based trust, and security-enhanced remote applications are given. As previously mentioned, the focus on available platforms for attestation is targeted at addressing our goal of IT control of increasing the trustworthiness of our thin client. Isolation techniques to increase the trustworthiness of a system are outlined as well as a comparison to attestation based approaches. Techniques targeted at increasing a user's trust in a local machine are discussed for possible inclusion into the TTT. Finally, previous research into a secure remote access client are outlined as a comparison to our overarching approach.

3.1 Attestation

As previously mentioned, attestation is the ability a machine to provide proof of it current configuration. We categorize attestation into two subcategories of hardware-based and software-based attestation. Hardware-based attestation, as the name implies, relies on hardware specifically designed to provide some mechanism for attestation. Software-based attestation techniques are generally targeted towards general purpose platforms that do not necessarily have hardware designed to provide attestation.

3.1.1 Hardware-based Attestation

One of the earlier hardware-based approaches targeted at providing RA was IBM's 4758 [20], a predecessor to IBM's 4765 processor currently available. The 4758 was a tamper-responding secure coprocessor providing a secure booting process, secure storage, and cyprographic functions including DES encryption and a secure hashing function. The secure boot process allows up to four layers of software to be loaded on the machine, with the first two being firmware required to boot the 4758 processor. Each layer is required to verify the integrity of the following layer prior to allowing it to execute and if the integrity of the succeeding layer is not valid, the processor falls into a failsafe mode. The secure boot procedure requires initial setup to link the separate software layer and is similar to a TPM based secure boot procedure outlined below. The 4785 processor is targeted at providing a general purpose computing platform and therefore the designers integrated the CP/Q operating system to provide a runtime environment for applications running in layer three. While one of the original goals of 4758 was to provide RA, the processor is only setup to provide a chain of software changes. Specifics are not provided as to how the individual layers or changes to them are stored, transmitted, and later identified by remote parties.

A more simplified approach whose goal is aimed at only to provided RA and x86 platform is Copilot [21], a coprocessor designed to specifically allow remote access to memory for the purpose of verifying kernel integrity. Copilot is implemented as a PCI express addon card with Direct Memory Access (DMA). The card is designed to allow remote verifier with an external connection to the Copilot addon card separate from all other machine components, to request an MD5 hash of critical sections of a kernel's memory to asses their integrity. This approach, however, requires the verifier to have a strong understanding of the kernel design and only allows a single verifier to access Copilot. It is also vulnerable to IOMMU tricks in which Copilot is only given a partial view of kernel memory and it cannot attest to the CPU state.

Trusted boot, sometimes referred to as measured boot, relies on a Core Root of Trust

for Measurement (CRTM), which is essentially the first piece of code that executes on a platform. The CRTM is tasked with computing and reporting a measurement of the BIOS software to the TPM prior to allowing the BIOS software to execute. The TPM will store this value by extending one of its PCRs. The BIOS is then tasked with repeating this processing of measuring and reporting on the bootloader, prior to allowing its execution and the process is continued until the operating system has been loaded. This process is generally referred to as creating a chain of trust (CoT) as the integrity of each piece of software is reliant on the software history prior to its execution, hence its trust is "chained" to the previous software units. Once a piece of software has been measured, a verifier is then able to make a judgment on its trustworthiness to continue measuring subsequent software. If malware is present in the chain of trust, again aside from the CRTM, it is unable to hide its presence and can only discontinue the CoT after it has received control of the processor. This is because before the malware receives control, it would have already been measured and reported to the TPM by its predecessor. The control structure of PCRs, only allowing for extensions to the register value means the malware is unable to erase its measurement.

Once a trusted boot sequence has completed, the TPM interface allows the operating system to request a quote for remote attestation. A quote is a PCR value cryptographically signed with the TPMs AIK. The quote created by the TPM can be provided to a remote machine that, after verifying the signature of the quote, can check that the PCR value representing the measurement of every element of the Chain of Trust (CoT) equates to a known trusted boot sequence. Verifying the PCR value requires the remote verifier to have precomputed the same CoT sequence , requiring the remote verifier to have a similar platform to the target and have previously measured each application present in the CoT. [22]

The trusted boot architecture can be extended to perform a secure boot, in which the discovery of malware or unknown software during the CoT process causes the machine to fall into a failsafe mode, usually causing the system to halt. Secure boot is performed by loading a TPMs nonvolatile memory with a Low Pin Count bus (LPC bus). This policy contains the correct measurements for each stage of the CoT process that are verified against as the system boots and utilizes Intel TXT to bring the system to a fail safe state if an invalid measurement is reported. [23]

The trusted boot platform was further extended by IBM to encompass the entire software history of a device since boot [23]. The Integrity Measurement Architecture (IMA) modified the Linux kernel to allow for measuring and reporting applications to the TPM after the operating system has loaded. The measurement points of IMA included all user-level executables, dynamically loaded libraries, dynamically loaded kernel modules, and scripts. In addition, IMA added a measurement list of the software history of the device and each

applications associated measurement. The measurement list provides an easier method for verifying the system in comparison to using a TPM quote, which can in turn be used to validate the list. This architecture suffers from several shortcomings that have limited its adoption. The use of configuration files for many applications, which have the ability to alter the behavior of the corresponding software, is not addressed by the architecture. A second limiting factor of IMA, also affecting both trusted and secure boot, is the semantics of the reported measurements. The architecture provide load time attestation, but do not give any indication of the current runtime state of the applications. Since being loaded, an application could be exploited by low integrity input e.g. unverified network data or application data of unknown provenance. The injected malicious code or altered execution state of the application would not be reflected in the corresponding TPM quote. Therefore, the meaning of a measurement as it relates to the system integrity is ambiguous. As reported by the authors, IMA accumulates about 500-600 measurements as part of the CoT over normal execution of a deskotp machine. Each measurement must be evaluated for its security implications and incurs overhead that is not feasible without a trust authority for reference.

Policy Reduced Integrity Measurement Architecture (PRIMA) [24], a continuation of the IMA platform, attempts to overcome the limitations of load time attestation while reducing the required chain of trust. It accomplishes this task by applying a Clark-Wilson model to the system, in which trusted applications are required to filter low integrity input before processing the data. This allows attestation of a target application to only require measurements of those applications which must be trusted in order to maintain the targets integrity. An example of a trusted application on most commodity systems would be the operating system as it has access to both code and data sections of any attested application. Requiring trusted applications to filter or sanitize low integrity input allows inference of the run time integrity of an application, while at the same time reducing the number of applications that must be trusted and therefor attested to in order to verify the integrity of a target application. A significant limitation of this work is the requirement for a strict Mandatory Access Control (MAC) policy. The MAC policy allows for information flow analysis to determine which applications must be trusted and must therefore filter low integrity data. However, correct creation of a MAC policy represents a responsibility we do not believe is valid for normal users. PRIMA is also hampered by the requirement of the operating system as a trusted entity, as this requires extensive modifications to the OS.

Flicker, a platform out of Carnegie Mellon University, represents a significant divergence from the previous works outlined. It is a secure platform for code execution built around the late launch functionality of a TPM [25] allowing for a dynamic, as opposed to static, RTM. Flicker is designed to allow small pieces of security sensitive code termed Pieces of

Application Logic (PALs), e.g. password input and verification, to execute in the secure late launch environment. The intended goal of Flicker is to allow short security sensitive operations to be performed with frequent and rapid transitions to and from the legacy system. While the architecture provides its intended functionality, the frequency of late launchs use coupled with the extensive overhead incurred in setting it up limit Flickers application.

3.1.2 Software-based Attestation

Pioneer, an entirely software-based attestation platform, represents a significant shift from the previously outlined hardware reliant approaches [26]. Software-based in general clearly do not require specialized hardware to execute and therefore are compatible with a larger percentage of commodity systems. They also benefit from the ability to update easily in case an error is found in the implementation or security vulnerabilities are found in the cryptographic primitives used. They also generally require strict timing constraints that limit their application, as seen in the works outline below.

Pioneer, specifically, is an early approach based on creating a software-based verification function. The verification function consists of three essential components: checksum code, a send function, and a verification function. The essential primitive of software-based attestation lies in the functionality of the checksum function, which is tasked with computing a checksum of the entire verification function. The underlying idea is to optimize the checksum function in such a was as to incur a time penalty if any portion of the function is spoofed by a malicious attacker. To create an optimized checksum function, resistant to spoofing attempts, Pioneer is designed to be a strongly ordered checksum that includes specifically selected CPU registers in order to defend against possible attacks. To attest to a desired piece of software, verifier first sends a nonce value to the attester which is then used as a basis for the checksum function. The checksum function computes a checksum of the entire verification function, requiring 2.5 million iterations to multiply any timing penality incured by an attacker to a scale large enough to overcome network jitter. The time delay between a reqeust and subsequent response by the attester is used by the verifier to indicate whether the rest of the verification function can be trusted. The verification function will the compute a hash of the attested software to run which is sent to the verifier who, based on the previous checksum results, can then make a judgement on the trustworthiness of the hash value and whether the attested software was correctly invoked following the verification function. This process effectively creates a CoT from the checksum function, the the hashing function, and finally the send and invocation function.

As an early attempt at timing-based attestation, Pioneer is limited by its applicable assumptions in that it does not address Symmetric Multi-Threading (SMT) or DMA attacks.

It also requires non-maskable interrupt handles to be replaced by a simple return instruction, meaning possibly time-sensitive interrupts may not be correctly handled. Subsequent work by the authors, titled PioneerNG [27]

A more recent approach from researchers at the MITRE Corporation [28] focused on relaxing the assumptions made by the authors of Pioneer and provided specific results as it relates to network characteristics. They also implemented a variant of their software-based approach with TPM assistance for more accurate timing measurements and included an in depth analysis of Time Of Check, Time Of Use (TOCTOU) attacks with regards to attestation in general. Their timing-based approach was implemented on a Windows system as a Network Driver Interface Specification (NDIS) Itermediate Driver (IM). This design allowed their self-checksum function to intercept network traffic at the lowest layer possible without being hardware specific. The self-checksum function differs from Pioneer in functionality by allowing interrupts to remain enabled. It accomplishes this functionality by placing the checksum below the esp register and including the EFLAGS and DR7 registers in the checksum functionality. Their self-checksum function is also able to take as input arbitrary memory addresses to be included in the checksum, removing the requirement for the hashing function found in Pioneer.

While timing-based attestation techniques are utilized because they have the ability to be hardware agnostic, the researchers at MITRE also looked into the use of TPM-based timing measurements to avoid errors induced by network latency. While they found that an initialization phase is required to learn the timing characteristics of individual TPMs, it provides a strong middle ground between exclusively hardware or software based platforms. However, their proposed technique, like Pioneeer, is limited in that it does not account for DMA or SMT. The authors relate both DMA access and SMT to the general attestation problem of TOCTOU. TOCTOU attacks occur when a malicious actor has access to attested memory following a subsequent measurement, indicating the memory at the "time of check" does not match the subsequent memory at the "time of use". The authors provide an outline of the requirements that must be met for a TOCTOU attack to be possible.

3.1.3 Time Of Check, Time Of Use

In a continuation of the outlined work by researchers at MITRE, we include a summary of their analysis of TOCTOU attacks as it has a direct relation to our work and to the best of our knowledge is the most thorough examination to date. TOCTOU attacks occur when a malicious actor has access to attested memory following a subsequent measurement, indicating the memory at the "time of check" does not match the subsequent memory at the "time of use". The authors postulate there are three requirements that all must be met for

a TOCTOU attack to be possible:

1. The attacker must know when the measurement is about to start
2. The attacker must have some unmeasured location to hide in for the duration of the measurement
3. The attacker must be able to re-install as soon as possible after the measurement has finished

3.2 Isolation

Approaches to isolating security sensitive applications from untrusted software generally consist of two categories. The first approach is a privilege isolation similar to kernel-user space access restrictions. This approach generally consists of relying on a hypervisor to correctly enforce protection mechanisms with the assumption that the limited responsibilities of said hypervisor induce far fewer vulnerabilities than commodity operating systems. The second approach is to partition resources including storage and process power between trusted and untrusted applications. While late launch could be categorized as a partition-based platform, we will focus on work that relies on partitioning its main security primitive.

Terra, a "trusted Virtual Machine Monitor", is a hypervisor based approach that is loaded onto a machine using trusted boot [3]. Terra provides isolation between virtual machines (VMs) with different security requirements including data confidentiality and integrity, utilizing its attested state by the trusted boot process and trust in Terra's correct implementation as assurance of the security mechanisms integrity. Terra also provides a software-based remote attestation service to guest VMs. It is noteworthy that unlike previous work, Terra does not provide a single hash value as a measurement as it assumes this is impractical. Instead, Terra provides individual hash values of 4 kB blocks, leading to a possible measurement size of roughly 20 MB. While the authors argue this does create a significant overhead for disk storage, it could possibly become an issue during transmission to a remote party. It is also unclear whether their block-based measurements provide greater functionality and could in fact require further analysis by a remote verifier.

TrustVisor, derived from the Flicker platform, is also a hypervisor-based approach, but differs in that it uses a DRTM as opposed to the SRTM utilized by Terra. [29]. TrustVisor overcomes the overhead incurred by the Flicker architecture through a single invocation of late launch, in which it the TrustVisor hypervisor is loaded and measured. The legacy operating system is then loaded on top of the hypervisor and allowed to run unrestricted. The hypervisors sole virtualization capability is to allow legacy applications to register Piece

of Application Logics (PALs), which will be executed in a secure environment separated from the legacy system. The secure environment is isolated from the legacy system through the use of virtualization techniques including nested page tables for memory protection and an IOMMU for DMA protection. Each registered PAL, prior to being executed, will be measured and reported to a corresponding software-based virtual TPM (vTPM). The trust placed in each vTPM is extended from the hardware-based late launch functionality and supported virtualization capabilities of the platform and the minimalist design of TrustVisor. While TrustVisor effectively overcomes the overhead incurred by the Flicker design, it still requires software developers to partition components of each application based on their security requirements.

Vasudevan *et al.* [30] implemented Lockdown, a partition-based platform for security applications on commodity systems. While lockdown is in fact a hypervisor, it is designed to fully partition an untrusted "red" environment from a trusted "green" system for security sensitive applications. Lockdown, like TrustVisor, is loaded into a MLE and its measurement is stored in a PCR on the TPM. Lockdown leverages the Advanced Configuration and Power-management Interface (ACPI) to save and switch between the "red" and "green" environments. The normal use of ACPI is to transfer a system into one of four sleep modes for power conservation. Instead, Lockdown employs ACPI to save the memory and system configurations of one environment and then wake, i.e. restore, the other environment. The hypervisor implementation is then tasked with intercepting and routing storage device access from the environments to partition available storage between the two environments. Lockdown provides code integrity an local attestation by first measuring applications prior to allowing them to load in the secure "green" environment. The measurement is checked against a list of pre-approved applications and if the application passes this process, its code is loaded into memory and Lockdown subsequently disables write access to the code to prevent interference from other "green" applications. Drawbacks to this approach, as stated by the authors, are the tradeoffs between security and usability. Lockdown provides greater assurance of security as compared to previous hypervisor approaches by requiring minimal functionality of the hypervisor, allowing Lockdown to only require 10k LOC. However, usability suffers when switching between environments, which can take 13-31 seconds to perform.

In parallel with the development of Lockdown, Vasudevan *et al.* implemented an external Lockdown Verifier to create a Trusted Path (TP). A **Trusted Path**, as defined by Zhou *et. al* [31], is a protected channel that assures the secrecy and authenticity of data transfers between a user's I/O devices and a program trusted by that user. Zhou *et al.* argue for the necessity of a trusted path as security primitive, since the lack of one negates any security

measures in place on the machine as the ability to forge and intercept user I/O prevents user verification of the systems integrity. Utilizing the previously mentioned TrustVisor architecture, McCune *et al.* created a platform in which trusted paths between a specific application and user I/O could be created on commodity X86 machines. This architecture is limited in that it does not allow for the use of device drivers and requires applications utilizing a trusted path to support the native hardware of the I/O devices. The authors admit their architecture does not address the unsolved problem seen in McCune *et. al* [32], in which a compromised system removes the ability to notify the user of the compromise. The Lockdown Verifier, building upon the proposed theoretical iTurtle device in McCune *et. al* [32], in a USB device designed to utilize DMA to communicate on the LPC bus to request quotes from the TPM. In this manner, the Lockdown verifier is able to indicate to the user using an LED and speaker as to the correct measurement of the Lockdown hypervisor and therefore provide assurance to the user the integrity of the current environment. While this work provides greater assurance of the integrity of a trusted path, it does not completely solve the previously mentioned problem with TrustVisor illustrated by the lack of a verifier to authenticate the integrity of the Lockdown Verifier. Expanding on the phrase taken from [32], it is "turtles all the way down" implying it will always be necessary to verify the last known verifier.

3.2.1 Mobile-based Approaches

Mobile-based attestation approaches are briefly outlined to provide a basis for discussion of future work. The following works build upon the TrustZone architecture to provide attestation to mobile devices.

The Trusted Language Runtime (TLR) [33] is a secure platform for code execution with many comparable characteristics to the previously mentioned Flicker architecture. TLR allows for the invocation of "trustlets", which are generally synonymous with Flicker's PALs. The trustlets run inside the TrustZone SW, providing hardware protection from the legacy OS and applications. The use of TrustZone as a hardware RoT allows it to provide code integrity protection and the ability to seal data to specific trustlets. In contrast to minimal TCB design of Flicker, TLR offers a rich runtime environment by including the .NET MicroFramework. This design choice highlights the intention of TLR to provide realistic usability, while admittedly only providing adequate trust properties best exemplified through the lack of remote attestation.

Other notable research on secure software execution utilizing the TrustZone hardware extension includes applying the TCGs Mobile Trusted Module (MTM) specification [34] to mobile platforms [35]. The MTM is essentially a mobile version of the TPM which provides

remote attestation while accounting for the possibly conflicting interests of parties involved in the platforms lifetime including manufacturer, service provider, and device owner.

3.3 Related Applications

Shulz and Sadeghi [36, 37] have extended and hardened the IPsec protocol to allow for attestation data to continuously be transmitted as part of the underlying VPN network. The extensions is designed to be agnostic of the underlying attestation protocol and relies on the L4/Fiasco microkernel to provide isolation and protection to the "sVPN" service running as an extension to the hypervisor. While "sVPN" service was successfully integrated into the L4/Fiasco microkernel and provided to guest VMs as a special hardware adapter, we are not aware of an implementation that integrates an attestation protocol to provide assurance of the hypervisor integrity.

4 Trusted Thin Terminal Design Overview

We develop the TTT, a remote access tool utilizing the late launch architecture, to provide both secure execution and remote attestation. The reasoning behind the design of the TTT is twofold. The first motivation behind its design is the use of the late launch architecture to effectively address our initially outlined goals of increasing trustworthiness and providing authentication. The ability of the TTT to remotely attest to the use of late launch, and therefore its execution, unmodified from external and possibly malicious sources, acts as an authentication measure to the remotely connected party such as a corporate network. To setup the MLE for the Trusted Thin Terminal, we utilize the previously mentioned Flicker architecture, launching the TTT as PAL. It is important to note that we utilize Flicker in a manner opposite its original design intention. Instead of running platform independent code whose execution time is minimal, the TTT will make use of multiple platform-dependent devices and will execute for similar durations to normal user applications.

The second motive for the design of the TTT lies in the inherent client server architecture of a remote access tool and its general purpose capabilities. The client server architecture allows a large portion of the processing requirements of the main functionality of the TTT to be pushed to the server. It also allows the functionality pushed to the server to be analyzed prior to execution, allowing IT to properly configure the servers to provide the desired functionality.

We envision a design of the TTT that provides similar functionality to Secure Shell (SSHv2). The current implementation is a proof-of-concept encompassing the required func-

tionality for the remote access tool including user I/O and network access. It is important to note that the SSH functionality we strive for is not inherent to our design and we envision the ability to provide other remote access techniques in the future including a fully featured remote desktop as outline in our Future Work section.

We believe the TTT has many benefits in comparison to previous work that make it more adept at the issue of securing unmanaged devices. The use of late launch provides many benefits in comparison to other previously outline techniques. Timing-based attestation techniques [28, 26] are limited to LAN deployment on account of their timing requirements, which are vulnerable to network jitter variations. While attestation techniques based on a static RoT such as IBMs IMA and PRIMA architectures [23, 24] overcome the previously mentioned timing issues and provide full functionality of a users machine, the required modifications to all or a significant portion of user applications and the operating system are overly cumbersome and still leave the system vulnerable to exploitation. Our approach relies on the minimal CoT of the late launch architecture to require users to only evaluate the trustworthiness of the TTT.

In comparison to the original intention of the Flicker platform, and to a certain extent the design of the TrustVisor hypervisor, we argue there are tradeoffs in both usability and security that allow our approach to better handle the issues created by BYOD policies. While the fine-grained security modules approach utilized by Flicker provides greater security in comparison to pushing the security responsibilities to IT professionals, we would argue the requirement of modifying each security-sensitive application used by employees is too great a barrier for practical use. In addition, we are able to mitigate the performance overhead associated with the MLE setup noted by the creators of Flicker in a similar manner to TrustVisor, that is by only requiring a single use of late launch over the execution of a single instance of the TTT.

Utilizing the late launch architecture allows the TTT to provide strong assurance against TOCTOU attacks, by mitigating the previously outlined requirement of an attacker being able to re-install as quickly as possible following a measurement. The isolation provided in an MLE by disabling Symmetric Multiprocessing (SMP), DMA[1], and interrupts allows the TTT to tightly control code executed on the system. We do concede, however, that the network capability of the TTT provides an interface for an attacker to regain system control during the MLE execution although we argue the minimal design of the TTT allows for thorough vulnerability analysis.

Comparing the TTT to previous works that have focused on securing remote access tools,

[1]DMA is not actually disabled, but rather the MLE is placed in a memory region protected from DMA access as part of Intel TXT

we would argue our work is the first to root trust in hardware. Previous research into securing the X Window system [38] utilizes SELinux, but does not use any hardware-based security mechanisms. While SELinux provides greater fine-grained access control in comparison to commonly deployed discretionary access control methods, it is unable to provide assurances of its integrity to remote parties and therefore cannot satisfy our authentication goal. There has also been recent work focusing on extending the IPsec protocol to include remote attestation abilities [37]. To the best of our knowledge, it does not appear as if this work has progressed to the point in which a remote attestation protocol has been integrated with their modified version of IPsec. It is also important to note that the authors implementation is reliant on the L4 microkernel, which runs in opposition to our intended goal of removing reliance on the operating system even when a microkernel design is used.

4.1 Adversary Model and Limitations

At the software level, we consider an attacker who has full control to run any software in kernel mode (Ring 0) with the exception of those attacks explicitly outline below. Because the attacker can exist at Ring 0, he has the ability to snoop on all network traffic, interface with the TPM AC module module of the machine, and arbitrarily read and write any code or data store in conventional memory. The attacker can also invoke a late launch environment through *GETSEC[SENTER]* with arbitrarily choosen parameters. However, peripheral firmware attacks in which a firmware vulnerability is exploited or a firmware update is invoked in order to introduce malicious code are not addressed in this work and are therefore prohibited. However, there has been previous research into applying timing-based attestation techniques to verify the integrity of peripheral software [39] that could be applied in future work to address this attack vector. We also do not address DoS attacks as the attacker is assumed to have full control of the legacy system and therefore it would be trivial for the attacker to simply shutdown the machine.

There are several limitations of the late launch architecture which we do not claim to address and therefore exclude from our adversary model. The first such limitation lies in the hashing algorithm utilized by the TPM. There was a significant collision attack shown against SHA-1 in 2005 [40]. While this attack does not compromise the second-preimage resistance of SHA-1, Nist recommended to phase out SHA-1 by 2010 in favor of SHA-2 [41]. Further, the TPM's hardware implementation of the hashing algorithm limit its ability to update currently deployed implementation which therefore could prove vulnerable in the future.

A more applicable attack against late launch lies in Intel TXT's inability to correctly

handle interrupts for System Management Mode (SMM), referred to as System Management Interrupts (SMIs). SMM is a mode for x86 processors that has the ability to preempt both kernel code and a Virtual Machine Monitor (VMM) and is generally utilized for handling low level hardware functionality or faults. However, as shown in [42], it is possible to overwrite the SMI handler with malicious code. As an attacker, this ability highly desired as SMM code cannot cannot be preempted and runs without any memory protection giving the attacker full access to system memory. While there are techniques available to prevent modification to SMM, their utilization is reliant on the BIOS and therefore are not necessarily implemented. Intel TXT separately provides two further protections against SMM exploitation. The first is the ability to make SMI maskable. This will, however, cause many systems that rely on SMM to handle low level tasks to freeze. Intel TXT also provides a dual-monitor technique, providing a second VMM alongside Intel VT-x called SMM Transfer Monitor (STM) which has the sole responsibility of virtualizing SMM code, effectively separating it from the rest of the system. There is currently no known implementation of the STM available and therefore the attacker is prohibited from modifying System Management RAM (SMRAM) or BIOS code tasked with loading SMRAM.

Yet another attack against the late launch architecture, specifically the Intel implementation, lies in known vulnerabilities of the AC module first loaded by the *GETSEC[SENTER]* [43]. A buffer overflow, found in the AC modules loading of the ACPI DMA Remapping (DMAR) table, enumerates the vulnerability of the late launch architecture to AC module exploitation. Exploitation of the AC module allows for arbitrary code execution prior to loading the MLE, effectively breaking the chain of trust and allowing a spoofed TPM measurement. Further compounding this issue, *GETSEC[SENTER]* disables BIOS protection of SMRAM, allowing the AC module to compromise SMI handlers as previously outlined. As described by the authors of the attack, the only reasonable mitigation is the use of an STM, which is not currently available, and therefore we do not address nor allow exploitation of the AC module by an attack. Without mitigating corrupt SMI handlers through the use of a STM, even correctly patched AC modules are vulnerable to attacks in which the attacker would be able to fully compromise a MLE and still authenticate as the original uncompromised MLE. We concede that this is a severe limitation of the late launch architecture that currently remains vulnerable to attack.

The attacker is prohibited from launching hardware attacks. This decision is motivated by previous attacks against the tamper-resistant TPM in which the RTS was compromised [44]. This includes snooping of system buses as the TPM is located on the LPC bus bus whose low bandwidth allows for possibly data interception or modification.

5 Trusted Thin Terminal Architecture

The TTT has two distinct layers providing different functionality. The lower layer is concerned with I/O including network capabilities as part of the TCP/IP stack, keyboard input from the user, and display. The upper layer includes higher level functionality of the SSH client.

We implemented the TTT on a Dell Optiplex 990 desktop computer with an Intel Core i5-200 processor and Intel Q67 Express chipset. The NIC was an integrated Intel 82579LM Ethernet LAN and the display adapter utilized was the onboard Intel HD Graphics 2000. We used Flicker v0.7 [45] and the AC module available directly from Intel termed "2nd-gen-i5-i7 [46]. **Note:** While the TTT generally consisted of an implementation of a PAL running on top of the Flicker architecture, some changes needed to be made that will be explicitly defined in the following description. Unless explicitly stated, the described functionality is contained within the PAL. The BIOS was updated to version A17 and the TPM, Intel TXT, and VT-x were enabled[2]. The TPM used was a ST Microelectornics ST19NP18-TPM.

5.1 Trusted Thin Terminal Display

The trade-offs between usability and security that influence the design of the TTT display provide a strong example of a theme that runs throughout the TTT framework. To provide a fully function display, we could simply have used or replicated available drivers for the Intel HD Graphics 2000 display adapter. However, this runs in opposition to our goal of creating a trustworthy thin client as it would add significantly to the TCB to provide support for only a subset of Intel display adapters. Instead, we opted to use a Video Graphics Array (VGA) standard text mode that provide a generic, albeit outdated, display supported by a large subset of commercially available machines. The VGA text mode was traditionally utilized by the MS-DOS operating system and is still used by subsections of some BIOS versions. While its prevalence in today's systems is diminishing, its inclusion as a VGA standard and the general popularity of VGA capable GPUs provide a strong guarantee of text mode support for the near future. The computational requirements needed to transition a machine to VGA text mode and display output to the user represent the minimal possible addition to the TCB. The use of a display based on text simplifies the TTT's design while meeting its requirements as we only ever need to display ASCII text.

[2]Note: it is highly recommended to update the BIOS to the latest version prior to utilizing Intel TXT. If the machine is improperly shutdown during an MLE execution, the system will expect the BIOS to initiate a *GETSEC[ENTERACCS]* command to launch an SCLEAN AC module, tasked with securing wiping memory. The system will prevent access to DRAM until the SCLEAN module has executed. Therefore, it is possible to completely disable a system with an incorrect BIOS version [43]

In order to transition a machine to 80 x 25 VGA text mode, the TTT must correctly configure 61 single byte VGA registers controlling aspects such as how video RAM is memory mapped and the current cursor location. Usually the transition to text mode is performed by the BIOS by executing the video BIOS option ROM located on the video card. This process can be triggered by the bootloader prior to transitioning the CPU to protected mode by executing a software interrupt. However, using this method would require the BIOS to be included in the TCB of the TTT, which would leave the TTT vulnerable to any malware running in kernel mode on the legacy system. Therefore, the TTT must manually configure the VGA registers to transition to text mode after being loaded into the MLE. The VGA registers are located in port-mapped I/O address space and can therefore be accessed using the *IN/OUT* instructions when the current privilege level (CPL) is less than or equal to the value stored in the *IOPL* register. The Flicker architecture is already configured to set the *IOPL* register to ring 3, allowing complete access to the I/O address space. The VGA registers that must be written are a mixture of both banked and unbanked registers. Unbanked registers may simply accessed placing their address in the *DX* register when executing an *IN* or *OUT* instruction. Banked registers map several VGA registers to a single set of Index and Data I/O addresses. In order to write to a banked register, the Index address must first be written to indicate the VGA register accessed during a subsequent read or write to the corresponding Data address. In total, 125 registers must be written to switch to 80 x 25 text mode. A good reference providing descriptions of each VGA register can be found at the Free VGA website [47] and a listing of the standardized values to use for 80 x 25 text mode can be found on the OSDev wiki [48].

Once the display has been transitioned to VGA text mode, the font data must then be loaded. Each of the 256 characters available in 80 x 25 text mode is represented in memory as an 8 x 16 bit field. Each bit represents whether the foreground (present) color or background (zero) color will be displayed in that section of the selected character map. In order to understand how font data is loaded, an understanding of VGA memory is first required. VGA memory consists of four planes of 64kBs of memory, totalling 256kBs of memory that can be memory-mapped to the address space *0x000A0000-0x000BFFFF* depending on the configuration of the Graphics and Sequencer registers. Plane one controls contains the characters index to be displayed and plane two is written to to set the color attributes of both the foreground and background. Plane three contains the font data and is used when an index is written to plane one to retrieve the character map for that index that should be displayed and plane four is not used by text mode. Under normal conditions for the 80 x 25 text mode, plane one and two are interlaced to the address space *0x000B8000-0x000BFFFF* with the character index being written to even memory addresses and the color attributes

for corresponding character contained in the ensuing odd memory address. Plane three is not memory-mapped under normal conditions and must therefore replace plane one and two prior to loading the font data. Once the font data is loaded to plane 3, the VGA display must be returned to normal operating conditions. Once the font data is loaded, the text mode is correctly setup and is used by the TTT by writing the plane one memory.

5.2 User Input

User input is entirely accessed through the keyboard. To make the TTT compatible with a larger portion of machines, it is desired that it can process input from both USB and PS/2 keyboards. While the functionality to hand PS/2 keyboards is relatively trivial, the two available methods for handling USB keyboards require are complex and require large additions to the TTT's TCB. The first available method to handle USB keyboards is provided by the BIOS through USB Legacy Keyboard support. Through this method, the BIOS configures the Platform Controller Hub (PCH) to trigger SMI interrupts when access to the PS/2 keyboard controller (KBC) occurs. It also loads SMRAM to transfer the KBC access to the USB keyboard device. There are, however, significant drawbacks to using this method. The first is the inclusion of the BIOS in the TTT's and that many current BIOS versions no longer support USB Legacy Keyboards. Further restricting use of this method, once operating systems receive control of the machine, they generally disable this functionality as SMM runs in a mode similar to real mode and the loaded BIOS code must correctly handle access to the rest of memory. The second available method consists of providing minimal USB support to handle USB keyboards alongside support for PS/2 keyboards. However, the design of the USB bus requires significant complexity illustrated by the corresponding drivers available in the Linux kernel, which cannot simply be ported to the TTT as they are reliant on other functionality of the Linux kernel. Considerable time would be required to port an available open source USB driver to the TTT and therefore this approach was not pursued for this research. While we admit it is not ideal to limit keyboard support to PS/2 devices, the complexity of providing USB support and the implications on the TCB are barriers for which we currently do not have a solution.

To correctly handle input from a PS/2 keyboard, the TTT must be able to properly handle peripheral hardware interrupts. The traditional manner in which peripheral interrupts were processed was first by the Programmable Interrupt Controller (PIC) included as part of the device's glspch. The PIC would receive an interrupt on one of its input pins from a peripheral device and then alert the CPU. The convention has changed slightly with the acceptance of multicore processors caused by the PIC's inability to route between different

cores. While the PIC functionality is still included in the PCH and the current implementation of the TTT only utilizes a single core, we designed the TTT to be able to use multicore interrupts with the assumption future capabilities may require it. The capability to route interrupts correctly in a multicore system is handled by the Advanced Programmable Interrupt Controller (APIC), also included as part of the PCH. The APIC is split between a single IO APIC, which directly receives hardware interrupts and routes them to the correct Local APIC, of which each core has an instance. Once the MLE is setup, the TTT must correctly configure the IO APIC, Local APIC, and the Interrupt Descriptor Table (IDT) of the CPU which routes interrupt vectors to the correct Interrupt Service Routine (ISR).

The Flicker architecture provides the PAL with a custom configuration of the Global Descriptor Table (GDT), a table used by the CPU to provide access control to memory along with conversion from a segmented address to a linear address. Flicker will also replace the GDT and IDT of the MLE with the legacy operating systems original tables once the PAL exits the MLE. It does not, however, provide the functionality to configure the IDT when the MLE environment is first configured. The IDT consists of 256 8-byte descriptor entries, in which the interrupt vector is used as an index to the IDT to select the correct descriptor that contains a segmented address to the corresponding ISR. The size of the IDT means the CPU is capable of handling up to 256 distinct interrupts. We modified the Flicker architecture to load the IDT using the *lidt* instruction with a table that consists of disabled ISRs, indicating the CPU should not process the received interrupt. Since it was already setup to load the legacy system's IDT upon MLE exit, this functionality did not need to be modified.

Once the PAL portion of the TTT is entered, the machine must then be correctly configured to an interrupt whenever a key is pressed by the user. Upon entering the PAL code, the TTT verifies interrupt handling by the CPU is currently disabled by executing a *CLI* instruction. It then proceeds to overwrite the current IO APIC (I/O APIC) configuration to disable all external maskable interrupts except IRQ1, which corresponds to a keyboard interrupt. However, there is initial setup required by the I/O APIC being dynamically memory-mapped. The location of the I/O APIC in memory is located in the Other Interrupt Controller (OIC) register located on the PCH. The OIC register is part of the chipset configuration registers of the PCH and is dynamically memory-mapped as well. The base address of the OIC register is located in the Root Complex Base Address (RCBA) register, located on the LPC Interface bridge which can be accessed using the PCI Express (PCIe) bus. Legacy access to the first 256 configuration registers of each device located on the PCIe bus is I/O memory mapped to ports *0x0CF8* and *0x0CFC* as index and data registers respectively. Since we have no need to access registers mapped beyond the first

256 to provide keyboard functionality, the legacy access method is sufficient and simplifies the PCIe code. Once the RCBA register has been read from the LPC Interface device, the OIC can correctly be accessed by reading from memory to identify the memory region that is mapped to the I/O APIC. The I/O APIC has two registers that are memory mapped, the I/O Register Select Register and the I/O Window Register, that act as index and data registers respectively. The I/O APIC contains 24 Redirection Table Registers corresponding to 24 individual maskable hardware interrupts. The TTT masks all interrupts except IRQ1 corresponding to the second redirection table, which it routes to the bootstrap processor (BSP)'s Local APIC since the PAL currently executes only on the BSP.

Once the I/O APIC is correctly configured, the TTT configures the Local APIC of the BSP to route interrupt messages received from the I/O APIC to the CPU. The TTT first checks that the Local APIC is enabled by checking the IA32_APIC_BASE model-specific register (MSR). If the Local APIC is not enabled, it unfortunately cannot be re-enabled without a system reset. It would be possible to fall back to single core PIC functionality to handle the case where the Local APIC is disabled. However, this functionality is currently not included in the TTT. To be able to get access to the CPU's MSRs, the TTT must be able to execute a *rdmsr* instruction. However, this instruction can only be executed when the CPL is zero and the TTT executes in ring 3. This again required modification to the Flicker architecture to read in the IA32_APIC_BASE MSR prior to switching to ring 3 and then pass the value read the PAL code. Once the TTT has verified the presence of a Local APIC and that it is enabled, it verifies the current task priority stored in a Local APIC memory-mapped register is setup to block interrupts. The Local APIC ID, which is also memory mapped, is utilized in setting up the I/O APIC when specifying the Local APIC used in handling keyboard interrupt (IRQ1).

After the APIC devices have been correctly configured, the IDT must be fixed to point to the correct keyboard ISR. Once this is completed, the KBC must be correctly configured to raise an interrupt on IRQ1 and to use the correct scan code. A scan code is simply a byte or several byte representation of a key. As an example, the scan code 2 representation for the letter "a" is 1C in hex and the scan code for the pause key being pressed is the chain of bytes E1, 14, 77, E1, F0, 14, F0, 77 in hex. The KBC controls the keyboard encoder (KBE), which is an IC located on the keyboard that processes user input and converts it to the correct value based on the scan code. Both the KBC and KBE are mapped to the I/O address space and configured by the TTT through the *IN* and *OUT* instructions. Once the KBC has been properly configured, the TTT re-enables interrupts by executing a *STI* instruction. The keyboard ISR is invoked whenever a key is pressed and is required to convert the scan code from the KBE to the corresponding ASCII value for display.

5.3 Network Interface Card Support

We describe the underlying functionality required to support the TCP/IP stack described in the ensuing section. It is important to note that unlike the display and user input, the underlying Network Interface Card (NIC) code is entirely platform specific. While there are general similarities between the Intel 825** GbE NICs, as seen by the e1000e Linux driver that provides driver support for a large subset the Intel NICs, this support incurs significant complexity exemplified by the driver being roughly 23k source LOC as counted by CLOC [49]. Providing abstract driver support in a secure and manageable manner is currently an open research problem discussed further in the Future Work section. Described below is the support provided for the Intel 82579 GbE PHY NIC. While we make a concerted effort to verify the NIC configuration properly, we admit there are likely configuration changes that currently could be made by the legacy OS to compromise the NIC functionality in an unforeseen manner. As an example, we currently do not protect the TTT from malicious configurations of the NIC's debug registers, which could be used to gain access to a small number of packets transmitted prior to the TTT being terminated. However, we argue it would certainly be possible to configure the NIC correctly for a complete product and do not believe this was required to show a proof-of-concept.

The main interface through the NIC occurs through the use of two circular tables, a receive descriptor table and a transmit descriptor table. Because the functionality of the two tables is similar, we will explain the use of the tables in the context of transmit descriptor table. The transmit descriptor table is located in main memory and the NIC has DMA access to the region. Similar to the OIC register used to program the I/O APIC, the base address of the NIC configuration registers including the register holding the address of the descriptor tables are dynamically memory-mapped and their base address is stored in a register located on the PCH. This address is fetched using the PCIe bus and is subsequently used to access the NIC's configuration registers. The transmit descriptor table, as the name implies, contains descriptors, each of which has configuration options and a memory reference to a buffer containing a full Ethernet frame to be transmitted. To coordinate between already transmitted and currently pending frames, the NIC contains both a descriptor table head and tail pointer. The head pointer is controlled by the NIC to indicate the descriptors that have been processed by the NIC. To transmit a frame, the TTT allocates a memory buffer and copies the frame to that buffer. It then configures the desciptor pointed to by the tail pointer to reference the newly allocated frame buffer and moves the tail pointer to the next available descriptor in the table to indicate to the NIC there is new data to be transmitted. The NIC will subsequently process and transfer pending frames using DMA to a FIFO buffer located directly on the NIC for transmission and will update the head pointer

once this transfer is complete. The process is very similar for packet reception. However, the TTT must allocate empty buffers which are presented to the NIC by writing a desciptor to the receive tail pointer and then update the tail pointer. The NIC, in a similar fashion, writes to empty buffers referenced in descriptors between the head and tail pointers and updates the head pointer to indicate frame reception.

The TTT also makes available the ability to offload checksum functionality to the NIC, including the IP Header checksum, the TCP/UDP checksum, and the Ethernet Frame Check Sequence (FCS). This functionality is enabled by writing a specialized descriptor, termed a TCP/IP Context Transmit descriptor, to the transmit descriptor table. The context descriptor does not reference data to send, but rather configures whether each checksum should be offloaded to the NIC, the byte location each checksum should be placed in the Ethernet frame, and the start and end location each checksum should be computed over. A single context descriptor can be written to the transmit descriptor table to process all subsequent frames as long as the configuration it contained does not change. The TTT also makes use of the NIC's ability to verify the checksum fields of a received frame, prior to writing a good frame to the receive descriptor table.

5.4 TCP/IP stack

The functionality currently encapsulated in the TTT TCP/IP stack include Ethernet at the link layer, IP at the network layer, and UDP at the transport layer. At the transport layer, the TTT provides DNS and DHCP traffic functionality, along with functionality similar to a UDP datagram socket.

5.4.1 Link Layer

Ethernet is the single link layer protocol used by the TTT. While wireless communication could be supported, this capability was limited by our lack of access to devices with the correct hardware. We also considered the ability to provide network access, regardless of the underlying protocol used, to be sufficient in providing a proof-of-concept. Much of the Ethernet functionality is handled by the NIC and therefore is not discussed.

5.4.2 Network Layer

The network layer provides IPv4 functionality to the TTT. It provides the basic functionality of the IP protocol and does not currently provide fragmentation. Packets received with the More Fragment bit set or an offset are currently dropped during packet reception. The reasoning for this was based solely on providing a proof-of-concept and was not motivated

by any known barrier. It also does not use or process the options fields of any packets transmitted or received. However, it does provide a utility function for converting an IPv4 address from dotted-decimal ASCII characters to network format to allow the user to specify IPv4 addressed.

5.4.3 Transport Layer

The stack currently includes a UDP implementation and does not provide TCP support. The reasoning for this is the same as described for a lack of IP fragmentation reassembly; This functionality was not needed for a proof-of-concept and there exist no known barriers to providing this functionality. We do note a timing mechanism must be provided to measure packet RTTs and handle TCP timeout and retransmission accordingly. The UDP checksum computation, in a similar manner to the IP header checksum, is offloaded to the NIC as has been previously described.

5.4.4 Application Layer

When the TTT first initializes, it makes a request for a new DHCP lease. This is to remove reliance on the operating system as to the system's current IP address and DNS server listed in the DHCP offer. The IP address and DNS server are utilized for subsequent requests and to provide the ability to resolve domain names.

5.5 Design and Implementation Issues

It is interesting to note that some protection methods provided by the late launch environment inhibit the software design of the TTT. One such example is the DMA protection provided by the MLE. The TTT is required to be placed in a Protected Memory Region (PMR) prior to executing the *GETSEC[SENTER]* instruction or else the AC module will fail to launch the TTT. The placement of the TTT in a PMR has a significant effect on memory allocation. The provided memory allocation of the Flicker architecture allocates memory regions within the PMR. This hinders allocation of transmit and receive descriptors for the NIC, which must be DMA accessible for the NIC to function correctly. Referencing a memory region within the PMR will actually cause the NIC to crash and subsequently bring down the Ethernet bus connected to the device. Memory allocation must therefore be placed in a DMA accessible region for the NIC to funciton properly.

Once the TTT has been launched, it is possible to disable the PMR, however, this is reliant on the ACPI table configured by the BIOS, which could be maliciously configured to compromise the TTT. Therefore, memory allocation must be configured to be placed

outside the PMR. The memory allocation framework provided by the Flicker architecture was modified to place memory allocation outside the PMR. However, this memory allocation is currently not coordinated with the legacy kernel and it is possible the TTT will overwrite legacy kernel memory, compromising it's integrity. To fix this issue, the kernel must provide the TTT with an allocated memory region, which the TTT will use to allocate memory if the provided memory region is valid. Otherwise, it will simply fall back to the current method. However, the memory setup by the kernel prior to launching the TTT must be physically sequential as the Flicker architecture executes with paging disabled. This desired functionality is currently not implemented in the TTT.

Another significant performance limitation of the TTT is incured by a hardware bug found in the NIC. The intended design of the NIC's receive ring is to have the tail pointer, controlled by software, incremented to indicate the memory region between it and the head pointer as valid descriptors to be used by the NIC when receiving packets. The head pointer is controlled by the hardware to indicate the descriptors which have been used by the NIC in storing received packets. When the head pointer is incremented to the point where it is equal to the tail pointer, it is our understanding with documentation as support that the NIC is in a state where it does not have memory allocated and should therefore drop any received packets until space is allocated. However, we found that the NIC did not correctly read the tail pointer and would increment the head pointer past the intended limit. This would subsequently cause the NIC to fault as it used descriptors that had not been properly allocated.

The described bug found in the NIC was compounded by the low performance found of the Flicker memory allocation. The initial design of the TTT would dynamically allocate memory regions for packet reception whenever the minimum threshold of receive space was met. However, because it could not keep pace with reception, the described bug would frequently occur. The provided fix the for the TTT is to allocate all receive descriptors prior to enabling reception for the TTT. However, this creates significant overhead for TTT setup, as described in the Results section.

6 Results

As a proof-of-concept, the intention of this work is to show that the capabilities required to implement the TTT are possible. The required capabilities of the TTT include user input, display output, and network capabilities. The current implementation of the TTT provides a text console to the user, in which it initially retrieves the IP address of the machine using the DHCP protocol and then allows the user to make TPM quote requests and DNS

queries. While the functionality of the TTT is proof of it's ability to provide the required functionality, there are limitations to the current approach as outlined below.

Currently, the display output relies on the VGA standard, which is largely compatible with most standard GPUs currently available. The TTT is currently reliant on the legacy OS to initialize the GPU to VGA mode and then the TTT transitions the VGA mode to the standards compliant 80x25 text mode. While we rely on the operating system to correctly initialize the GPU, malicious or accidental misconfiguration will not compromise the security of the MLE, but rather create a denial of service. Our reliance on pre-configuration of the GPU is similar to assumptions made by the Flicker architecture, which is reliant on the correct configuration of the ACPI to load the PAL into a DMA protected memory region. If the ACPI is mis-configured, Flicker will fail to launch, essentially creating the same denial of service possibly created by a mis-configuration of the GPU.

As outlined in the User Input section, input from the user is currently limited to PS/2 keyboards. This limitation stems from the complexity required to provide USB drivers and requires further investigation into the minimal required functionality to support USB devices. While this does not limit the security or functionality of the TTT, it could severely limit the adoption of our approach based on the limited use of PS/2 keyboards, which are being phased out in favor of USB keyboards. While this is currently a limitation of the TTT, we note that it is only a limitation of the implementation and not the overall design.

The current network capabilities of the TTT represent the greatest impediment to hardware agnostic trustworthy thin terminal. While the TTT is capable of providing the required network capabilities, the current implementation is entirely reliant on the provided NIC being an Intel 82579LM Ethernet LAN as there is limited interoperability between NIC devices. As outlined in the Future Work section, providing secure device drivers is currently an open problem and one that must be addressed to make the TTT a reality for widespread use.

While this work servers as a proof-of-concept, performance is still important with regard to feasibility. The main hindrance with regards to performance of the TTT is based on the overhead of allocating memory for the NIC to address the previously mentioned hardware bug. The flicker architecture takes roughly 2.7 seconds to launch with a standard deviation of 0.13 seconds. The overhead incurred in allocating memory for the NIC pushes the launch overhead to 10.8 seconds with a standard deviation of roughly 1.2 seconds. While this is roughly a threefold increase in performance, we would argue this represents a normal launch delay for large applications and would not adversely affect usability of the TTT.

7 Future work

While the described work provides a proof-of-concept of the intended TTT and gave a good understanding of the hardware functionality required to create a minimal client, there are several features that could be used to greatly increase the functionality of the TTT. Outlined below is a brief introduction into possible areas of future research related to the TTT.

7.1 Driver Support

The first priority as an addition to the TTT is the addition of USB support. Not only does this allow the TTT to interface with USB keyboards, but it can also be used to provide a mechanism for enforcing access control policies to the remote server. Since one of the founding ideas of the TTT is to provide a trustworthy remote client, remotely connected servers can provide access to confidential data and trust that an access control policy they attach to that data is correctly implemented by the TTT. Allowing for local storage on a USB device of remotely accessed data provides a strong proof-of-concept into how an access control policy could be applied. The ability to control access to data through the TTT is not only limited to storage and provides an avenue for further research.

As previously outlined, a similar addition to the TTT includes the ability to provide a graphical display to greatly increase its functionality and usability. While the previously outlined text display is applicable to all graphics card with VGA support, it is unclear if there is a similarly standardized graphical display. While VGA does provide a standard for graphical displays with high resolution, it is not known whether this would be completely hardware agnostic. A common thread through the two discussed additional features is the ability to provide driver support, which we believe provides the greatest potential for research and future work related to extending the TTT. Providing driver support for the primary hardware devices of a system including GPU and USB is not regulated to simply porting available drivers to the TTT. An empirical study of the Linux kernel over period of almost 2 years found that device drivers are three to seven times more likely to contain bugs than the rest of the kernel [50]. Research questions would focus on how to protect the trustworthy components of the TTT from malicious or accidental corruption by device drivers and how to verify the drivers are behaving as expected. Previous approaches [51, 31] have looked at providing hypervisor support to protect the kernel from the device drivers. Whether these approaches or possibly new techniques could be applied to the TTT is of interest and if a solution is found, it would provide a framework for considerable extension to the current TTT implementation.

7.2 Authentication

Another significant area of research related to the TTT is the use of access methods. The previously outlined issues with password use as an authentication measure on account their lack of entropy provides a strong argument for the use of public key authentication mechanisms such as the one provided by SSH. This method allows a machine to authenticate itself in relation to a public key through a challenge-response protocol in which it proves ownership of the corresponding private key. The integral part of this scheme relies on the confidentiality of the private key, as with other public key schemes. The TPM is well positioned to provide a strong mechanism for private key protection through its ability to provide a Root of Trust for Storage. While we are not aware of currently available VPN or SSH software that makes use of the TPM for secure key storage[3], it has been used by encryption software such as Microsoft Bitlocker for this purpose [52].

We envision usage of a TTT authentication key being tied to the measurements of the AC module and TTT in the TPM's PCR 17 and PCR 18, respectively. This provides a strong access control mechanism, protecting the proposed TTT authentication key's confidentiality in the event of the legacy systems compromise. We also envision the possibility of the authentication key being tied to a user identity, rather than a specific machine, and therefore the ability to migrate the authentication key to another platform would be desirable. While the TPM provides a mechanism for key migration to and from TPMs, this protocol is reliant on a trusted third party with similar responsibilities to those originally envisioned of a TPM Certificate Authority, who would authenticate a given identity in relation to an associated TPM Endorsement Key. Both trusted third-party infrastructures never came to fruition and therefore the current key migration scheme must be adapted. Adaptation of the key migration protocol or creation of a new protocol would require research into this area and modification to the Flicker architecture as it currently does not provide any functionality related to migratable keys.

8 Conclusion

The goal of this work was to provide a Trusted Thin Terminal that addresses the security issues created when a Bring Your Own Device policy is allowed in a corporate setting. These issues included an asymmetrical imbalance between the trustworthiness and required trust

[3]While we could not find any reference of current remote access software making use of the RTS of a TPM for key storage, it would be surprising if in fact this method has not been used. We are not arguing our outlined approach would be novel, but rather that it would provide stronger key protection than legacy software reliant on the operating system

in today's commodity operating systems, along with the inability of the general public to correctly configure their devices to meet organizational security goals. We subsequently outlined 3 goals in relation to the BYOD problem: shift security responsibilities to IT control, provide a trustworthy thin client for access to corporate networks utilizing isolation techniques for protection, and provide a basis for authentication of the thin client system.

To accomplish the intended goals, we built upon previous research using a Trusted Platform Module, a secure coprocessor providing a Dynamic Root of Trust for Measurement in designing the Trusted Thin Terminal. This allowed us to develop a proof-of-concept proving user I/O and network capabilities, both of which are essential in developing a remote access program addressing our goal of IT control of security responsibilities. We relied on the Flicker architecture, with noted modifications, to provide a gateway into a secure Measured Launch Environment, in which our TTT executes in an isolated state unhampered by legacy software, providing a framework for increased trustworthiness. The use of the late launch architecture allows the TTT to provide trustworthy authentication to a remote server, effectively addressing our goal of authentication.

While our proof-of-concept provides the framework for creating a usable text-based remote client, we admit there are several limitations to our approach we discovered in developing the TTT. We found there are significantly complex hardware configurations that must be addressed to provide even minimal functionality, exemplified by the inclusion of at least 23K source LOC to provide USB keyboard support. The variety of GPU configurations limits our ability to provide a graphical display without recreating the very operating system we are attempting to replace. While this limited the functionality of the TTT and currently makes it hardware specific, it provides a strong platform for future research into authentication methods and device driver security. The TTT proof of concept provides a first step towards creating a truly trustworthy thin client capable of addressing the security issues of user owned devices in corporate settings.

References

[1] H. M. U. Ltd. What's new in linux 3.6. [Online]. Available: http://www.h-online.com/open/features/What-s-new-in-Linux-3-6-1714690.html?page=3

[2] Microsoft. (2013) A history of Windows. [Online]. Available: http://windows.microsoft.com/en-US/windows/history

[3] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 193–206.

[4] M. Weir, S. Aggarwal, M. Collins, and H. Stern, "Testing metrics for password creation policies by attacking large sets of revealed passwords," in *Proceedings of the 17th ACM conference on Computer and Communications Security*. ACM, 2010, pp. 162–175.

[5] Anonymous. (2012) Internet census 2012. [Online]. Available: http://internetcensus2012.bitbucket.org/paper.html

[6] C. Scott. Nearly a fifth of U.S. PCs have no antivirus protection, McAfee finds. [Online]. Available: http://www.pcworld.com/article/256493/nearly_a_fifth_of_us_pcs_have_no_virus_protection_mcafee_finds.html

[7] AVG Technologies. Avg antivirus. [Online]. Available: http://free.avg.com/us-en/homepage

[8] AVAST Software a.s. Avast antivirus. [Online]. Available: http://www.avast.com/index

[9] Malwarebytes Corporation. Malwarebytes antivirus. [Online]. Available: http://www.malwarebytes.org/

[10] Cisco. Cisco study: IT saying yes to BYOD. [Online]. Available: http://newsroom.cisco.com/release/854754/Cisco-Study-IT-Saying-Yes-To-BYOD

[11] C. Camp. The BYOD security challenge: How scary is the iPad, tablet, smartphone surge? [Online]. Available: http://www.welivesecurity.com/2012/02/28/sizing-up-the-byod-security-challenge/

[12] T. C. Group. (2013) TCG members. [Online]. Available: http://www.trustedcomputinggroup.org/about_tcg/tcg_members

[13] ——. (2009) How to use the TPM: A guide to hardware-based endpoint security. [Online]. Available: http://www.trustedcomputinggroup.org/resources/how_to_use_the_tpm_a_guide_to_hardwarebased_endpoint_security/

[14] R. Lemos. (2006) U.S. army requires trusted computing. [Online]. Available: http://www.securityfocus.com/brief/265

[15] T. C. Group. (2008) Enterprise security: Putting the TPM to work. [Online]. Available: http://www.trustedcomputinggroup.org/resources/enterprise_security_putting_the_tpm_to_work/

[16] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," 2011.

[17] Texas Instruments. OMAP mobile processors : Security. [Online]. Available: http://www.ti.com/general/docs/wtbu/wtbugencontent.tsp?templateId=6123&navigationId=12316&contentId=4629&DCMP=WTBU&HQS=Other+EM+m-shield

[18] Giesecke and Devrient. Tee. [Online]. Available: http://www.gi-de.com/en/trends_and_insights/mobicore/tee/TEE.jsp

[19] Sierraware. Open Virtualization. [Online]. Available: http://openvirtualization.org/

[20] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. Van Doorn, and S. W. Smith, "Building the IBM 4758 secure coprocessor," *Computer*, vol. 34, no. 10, pp. 57–66, 2001.

[21] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, vol. 6, no. 3. San Diego, CA, USA: USENIX Press, 2004, pp. 6–4.

[22] J. McCune, B. Parno, A. Perrig, M. Reiter, and A. Seshadri, "How low can you go?: recommendations for hardware-supported minimal TCB code execution," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1. ACM, 2008, pp. 14–25.

[23] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th conference on USENIX Security Symposium*, vol. 13, 2004, pp. 16–16.

[24] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *Proceedings of the eleventh ACM symposium on Access control models and technologies*. ACM, 2006, pp. 19–28.

[25] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," *SIGOPS Operating Systems Review*, vol. 42, no. 4, pp. 315–328, 2008.

[26] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 1–16.

[27] A. Seshadri, A. Adviser-Perrig, and P. Adviser-Khosla, "A software primitive for externally-verifiable untampered execution and its applications to securing computing systems," 2009.

[28] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 239–253.

[29] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[30] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a safe and practical architecture for security applications on commodity platforms," in *Trust and Trustworthy Computing*. Springer, 2012, pp. 34–54.

[31] Z. Zhou, V. Gligor, J. Newsome, and J. McCune, "Building verifiable trusted path on commodity x86 computers," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 616–630.

[32] J. McCune, A. Perrig, A. Seshadri, and L. van Doorn, "Turtles all the way down: Research challenges in user-based attestation," DTIC Document, Tech. Rep., 2007.

[33] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Trusted language runtime (TLR): enabling trusted applications on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 2011, pp. 21–26.

[34] T. C. Group. (2008) TCG mobile reference architecture. [Online]. Available: http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_reference_architecture

[35] J. Winter, "Trusted computing building blocks for embedded linux-based ARM trustzone platforms," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*. ACM, 2008, pp. 21–30.

[36] A.-R. Sadeghi and S. Schulz, "Extending IPsec for efficient remote attestation," in *Financial Cryptography and Data Security*. Springer, 2010, pp. 150–165.

[37] S. Schulz and A.-R. Sadeghi, "Secure VPNs for trusted computing environments," in *Trusted Computing*. Springer, 2009, pp. 197–216.

[38] D. Kilpatrick, W. Salamon, and C. Vance, "Securing the X Window system with SELinux," Technical Report 03-006, NAI Labs, Tech. Rep., 2003.

[39] Y. Li, J. M. McCune, and A. Perrig, "Viper: verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 3–16.

[40] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full SHA-1," in *Advances in Cryptology–CRYPTO 2005*. Springer, 2005, pp. 17–36.

[41] NIST. NIST comments on cryptanalytic attacks on SHA-1. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/statement.html

[42] R. Wojtczuk and J. Rutkowska, "Attacking Intel trusted execution technology," *Black Hat DC*, 2009.

[43] ——. (2011) Attacking Intel TXT via SINIT code execution hijacking. [Online]. Available: http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf

[44] C. Tarnovsky. (2010) Deconstructing a 'secure' processor. [Online]. Available: https://media.blackhat.com/bh-dc-10/video/Tarnovsky_Chris/BlackHat-DC-2010-Tarnovsky-DeconstructProcessor-video.m4v

[45] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: Minimal TCB code execution. [Online]. Available: http://sourceforge.net/projects/flickertcb/files/

[46] S. A. Sehra. Intel trusted execution technology. [Online]. Available: http://software.intel.com/en-us/articles/intel-trusted-execution-technology

[47] J. Neal. Hardware level VGA and SVGA video programming information page. [Online]. Available: http://www.osdever.net/FreeVGA/vga/vga.htm

[48] OSDev. Vga hardware. [Online]. Available: http://wiki.osdev.org/VGA_Hardware

[49] Cloc. [Online]. Available: http://cloc.sourceforge.net/

[50] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, *An empirical study of operating systems errors*. ACM, 2001, vol. 35, no. 5.

[51] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 207–222, 2003.

[52] Bitlocker drive encryption overview. [Online]. Available: http://windows.microsoft.com/en-us/windows-vista/bitlocker-drive-encryption-overview