Towards Leveraging Late-Launch to Create Trustworthy Thin-Terminal Clients

Evan J. Frenn and Craig A. Shue
Computer Science Department
Worcester Polytechnic Institute, Worcester, MA, USA

Abstract

In computer security, there is often a disconnect between the trust placed in a device to meet a security goal and the actual ability of the device to meet these goals. In organizational environments, this disconnect may become larger as users increasingly use their personally-owned computing devices for work purposes. These users often lack IT backgrounds and do not properly secure their devices, creating greater security risks. In this work, we propose using a Trusted Platform Module (TPM) to enter a "late-launch" environment, where it will exclusively execute trusted, organization-provided code to create a thin-terminal on user devices. This thin-terminal will interact with centralized IT servers, providing useful functionality to the user while ensuring the device itself will pose no risk to the organization's security goals. We have implemented a proof-of-concept version of this environment and showed how simple text-based interactions can be performed with a trustworthy client. In doing so, we highlight the challenges and tradeoffs inherent in such an approach.

Keywords: Trusted computing; dynamic root of trust; trusted platform module.

1 Introduction

A fundamental problem with respect to computer security lies in the disparity between the trust and trustworthiness of today's systems. A system, B, is considered trusted by another system, A, if the actions of B can influence the security goals of A. However, that same system, B, is only considered trustworthy if it will ensure that A's security goals are met. While conceptually straightforward, many systems are implicitly trusted by organizations and users without any evidence that the systems are actually trustworthy.

Software defects are a significant barrier to a system's trustworthiness, since a single defect may be sufficient for an adversary to exploit. Defects are common in software, with software validators considering code to be high-quality when it has only a single defect (or less) per thousand lines of code [1]. However, modern operating systems are large: Linux 3.6 has roughly 16 million lines of code [2] while Microsoft Windows XP had 45 million lines of code [3][1]. Accordingly, a modern operating system may have tens of thousands of defects, even if they have less than the average 1.0 defect per thousand lines of code.

While the inherent size of modern operating system may undermine their trustworthiness, the management of these systems may also be a cause for concern. A recent Cisco study of roughly 600 U.S. business found that 95% of employers allowed employees to use their personally-owned devices for work-related matters [4]. These employees, who may not have IT backgrounds, are responsible for managing these devices. Untrained users often practice poor password management [5], [6] and device security practices (such as 20% of systems not using anti-virus [7]). With poor system administration by users without IT skills, organizations must consider many employee-owned devices to be untrustworthy.

In this work, we aim to address the security issues created by acceptance of Bring Your Own Device (BYOD) policies by organizations. Specifically, we intend to create the Trusted Thin Terminal (TTT), a tool designed to support BYOD policies while concurrently meeting the following goals:

1. **Support Centralized IT Administration:** Rather than use the possibly compromised functionality of an employee-owned device, we will keep functionality on organizational servers which are maintained by skilled IT professionals. While users may have the convenience of mobile device interfaces, the actual organizational assets will remain on IT infrastructure.

2. **Make Client Systems Trustworthy:** We will ensure client systems can be considered fully trustworthy by an organization. In particular, an organization will be able to provide an execution environment to a client and have assurances that the provided code, and only that code, is executing on the client while accessing organizational assets. These clients will function as thin-terminals that allow their users to interact with remote organizational assets.

3. **Provide Evidence of Authenticity:** The client systems will have an ability to attest to their running state and prove they are running code provided by the organization.

[1] Microsoft has not disclosed the number of lines of code in Windows Vista, 7, or 8.

To create the TTT, we will use features of the Trusted Platform Module (TPM), a hardware component that is present in many computing systems. In particular, we use a recent addition in the TPM in version 1.2, called "Late Launch," that allows the creation of a Dynamic Root of Trust for Measurement (DRTM). When done correctly, this feature allows a system to enter a hardware-protected mode where a specific segment of code will execute without interference from the traditional operating system or the applications running on it. Even if the employee's device is compromised with malware, the software running in the late launch environment will be properly isolated and can be trusted by the organization to execute properly.

We envision a deployment scenario where each employee-owned device is registered with the IT staff. As part of this process, the IT staff will install an application on the device that can trigger entry into the late launch environment, along with the code required to implement the TTT. To enter this mode, a user would simply launch the organization application, which would transition the device into a thin-terminal mode and connect to an organization server. The device would then attest to its running state, providing the organization with assurances that it is executing safely.

To demonstrate the feasibility of the approach, we have created a proof-of-concept implementation of the TTT. In only 6,294 lines of code, our implementation supports user I/O and network communication, allowing the client to interact with a server in a trustworthy manner. This implementation can be further extended to provide full graphical interaction with the remote server while minimizing the amount of code on the client.

The remainder of this work is organized as follows. Section 2 provides background and a discussion of related work. Section 3 describes the design of the TTT, including the adversary model. Section 4 describes the TTT architecture and implementation. We provide discussion and future work in Section 5 and conclude in Section 6.

2 Background and Related Work

The trusted computing field is broad, with many areas of related work. Hardware implementations such as IBM's 4758 [8] and Copilot [9] created a base for trusted computing work. Other hardware, such as ARM's TrustZone platform can be used as a base for mobile devices [10]. However, in this work, we focus on the Trusted Platform Module and how it has been previously used by the community since this serves as the base for our own work.

The Trusted Computing Group (TCG), a consortium of roughly 110 technology-related companies including AMD, Cisco, IBM, Intel, and Microsoft [11], created the specification and implementation of the Trusted Platform Module (TPM). The TPM is a secure, tamper-proof integrated circuit that is designed to generate cryptographic keys and provide secure storage, remote attestation, and a pseudo-random number generator. TPMs can be found on a large number of devices including those manufactured by Dell, HP, Acer, Lenovo, and Toshiba [12]. The U.S. Army and Department of Defense require all purchased machines to include a TPM [13, 14].

2.1 TPM Attestation

TPMs have several functions to support secure operations. One of the most well-known TPM features is its ability to create an *attestation*. An attestation allows a machine to make a claim to a remote party, called the appraiser, with evidence that supports the claim [15]. The TPM serves as the root of trust for system measurements, sealed storage, and reporting.

Many attestations include a measurement. In TPMs, a measurement 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 Platform Configuration Registers (PCRs) by taking the SHA-1 of the measurement concatenated with the previous value of the PCR. This can be denoted as $PCR_i \leftarrow$SHA-$1(m|PCR_i^{old})$. These PCRs are controlled by the TPM and can only be altered in this manner and during a system boot (at which point they are initialized to zero). This controlled interface provides rich functionality while minimizing TPM complexity.

Trusted boot, sometimes called "measured boot," is an example of using TPM measurements. In trusted boot, the TPM computes a measurement of the BIOS software prior to executing the BIOS software. The TPM will store this value by extending one of its PCRs. The BIOS is then tasked with repeating this process 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, 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. This measurement can then be signed by the TPM's Attestation Identity Key (AIK) and verified by a remote system. However, the verifier must have a similar platform to the target and have previously measured each application present in the Chain of Trust (CoT) [16]. A variant of trusted boot, called "secure boot," proceeds similarly, but halts the system if an invalid measurement is reported. [17]

The trusted boot platform was further extended by IBM to encompass the entire software history of a device since boot [17]. 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 application's associated measurement. In Policy Reduced Integrity Measurement Architecture (PRIMA) [18], the IMA process was enhanced 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 [19] 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 target's integrity. However, even the PRIMA approach requires a strict Mandatory Access Control (MAC) policy to manage information flow and trust. PRIMA is also hampered by the required inclusion of the operating system as a trusted entity, as this requires extensive modifications to the OS.

Attestation is affected by Time Of Check, Time Of Use (TOCTOU) attacks. These 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." Kovah *et al.* [20] 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, and 3) the attacker must be able to re-install as soon as possible after the measurement has finished. Kovah *et al.* indicate that both Direct Memory Access (DMA) access and Symmetric Multi-Threading (SMT) are manifestations of the general attestation problem of TOCTOU.

Each of these attestation approaches face a TOCTOU concern due to timing and possibly malicious behavior on the client. In our approach, we will use an isolated environment that is known to the remote system. Accordingly, the remote system can verify the environment, confirm the attestation came from that environment, and knows how long that environment will continue to run. This disrupts the TOCTOU attack requirements and allows our system to behave in a more trustworthy manner.

2.2 TPM 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. In 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.

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 variations are inconsequential for this work and we focus on the latter for simplicity. 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. When loading the MLE, the system will measure and extend the MLE into PCR 18. While executing, interrupts are disabled and all other processing cores on the system are idled until the MLE exits or manually reactivates these features.

McCune *et al.* [21] created a tool, called Flicker, that is built around the late launch functionality of a TPM. Flicker is designed to allow small pieces of security sensitive code, termed Pieces of Application Logic (PALs), that execute in the secure late launch environment. These code segments, which can perform operations such as password verification, can be performed in a trustworthy manner, even in an otherwise untrusted system. 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 launch's use, coupled with the extensive overhead incurred in setting it up, limit Flicker's application.

In our approach, we will use the late launch environment and use the Flicker tool to create a PAL that allows us to implement the TTT. However, we will avoid the frequent transitions that hinder Flicker's utility.

2.3 TPMs for Isolation

Other work extends late launch to provide isolation between the secure environment and the general computing base. In Terra, Garfinkel *et al.* [22] create a "trusted Virtual Machine Monitor" hypervisor-based approach that is loaded via trusted boot. Terra provides isolation between virtual machines (VMs) with different security requirements, including data confidentiality and integrity, by utilizing its attested state by the trusted boot process and trust in Terra's correct implementation as assurance of the security mechanisms integrity.

McCune *et al.* [23] extended their work on Flicker to create TrustVisor, which like Terra, is a hypervisor-based VM approach. However, it uses a DRTM, as opposed to the Static Root of Trust for Measurement (SRTM) utilized by Terra. 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 hypervisor's sole virtualization capability is to allow legacy applications to register 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.* [24] 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 and 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 lines of code (LOC). However, usability suffers when switching between environments, which can take 13-31 seconds to perform.

While each of these approaches create a trustworthy hypervisor to allow a simultaneous "trusted" and "untrusted" environment, we simply omit the "untrusted" environment. We place the system into a minimalist OS mode where all operations occur in the trusted state, allowing the system to act as a simple interface to a fully-functional remote server.

3 Trusted Thin Terminal Design

Our Trusted Thin Terminal design is specifically aimed at minimal functionality on the client. The client gains all its functionality from a central IT server. This model is similar to remote execution through the X11 windowing protocol [25] and the Secure Shell (SSH) [26] protocols. We explicitly put the functionality and difficulty in securing those environments in the hands of IT professionals rather than attempt to secure the devices owned by users.

We use the late launch TPM functionality as a base for our approach. Accordingly, our approach has some similarities with Flicker and TrustVisor. However, unlike Flicker, which invokes late launch for short periodic bursts, we enter the late launch mode for a prolonged period. TrustVisor makes a similar design decision to create a trusted hypervisor that provides a secure execution environment for Flicker-like invocations. However, programmers must alter their code to comply with TrustVisor to perform these functions.

In our approach, we enter the late launch environment to create a micro-OS for the user. Once in the late launch environment, we perform simple polling of input and network devices. Upon obtaining new network input from the remote server, we update our video display. When we receive new user input, we transmit it over the network to the remote server. This functionality is sufficient for many interactive processes and replicates the functionality of VNC's Remote Framebuffer Protocol [27] clients. However, the minimal code base allows comprehensive evaluation to reduce the errors associated with the program.

We now describe our threat model and then describe the process of invoking the TTT.

3.1 Adversary Model and TPM Limitations

In our design, we assume that the adversary has full access to the general computing environment, including privileged access to the operating system and all applications. The adversary may have full control of the network communication between the client and the remote server. The adversary may alter the TTT software on the client arbitrarily. However, such alterations have the same result of a denial-of-service (DoS) attack, which the adversary can already launch in a more straightforward manner (such as deleting the executables entirely). Essentially, if the executable is corrupted, it will either not run or it will fail to properly attest to the remote server and thus the attack is equivalent to a DoS.

We include both the remote server, the system's CPU, and the TPM inside our trusted computing base (TCB). CPUs and IT servers are often already in an organization's TCB and TPMs are designed to be inside the TCB. We assume the attacker has not physically altered the system.

The remaining assumptions for the adversary model are simply to address inherent limitations of the TPM and its implementation. These limitations are common across all systems that make use of the late launch functionality and are being investigated by manufactures and the commu-

nity. In particular, attacks using peripheral software [28], weaknesses in the TPM hashing algorithms [29], System Management Interrupts (SMIs) [30], the `GETSEC[SENTER]` modules [31], or TPM hardware attacks [32] are excluded from our model. As the community addresses these vulnerabilities, our assumptions may be relaxed. Until then, these assumptions make the problem tractable.

3.2 Invoking Late Launch

To implement the TTT, we use the Flicker architecture to create a Piece of Application Logic (PAL) that implements the TTT functionality. This PAL is then provided as a MLE for the late launch call. We group all this functionality into a logical unit that we label the "Trusted Thin Terminal Launcher." The launcher is executed by the user as a regular application on the device. It then performs the `GETSEC[SENTER]` call on the TTT executable, disabling interrupts and suspending other computation cores while causing the TPM to measure the executable, hash the executable, and extend the hash value into the PCR 18 register. Once this process completes, the CPU begins executing the TTT program. The TTT can then request a measurement and quote of the TPM, reflecting the new value of PCR 18, as evidence that it can then supply to the remote server to prove that it is executing in a known good state. We depict this process in Figure 1.

Once in the TTT program, the system begins a looping process where the user inputs are relayed to the remote server and the server inputs are used to update the user's screen.

Importantly, our design and use of the late launch approach avoids the concerns with TOCTOU attacks. In particular, since the measurements of the relevant PCRs can only be made in the late launch environment, the remote server knows that the legitimate code was executing at the time of the measurement. By coupling the measurements with keying information, the remote server can ensure messages will only be processed by a client in the trusted environment. This provides protection against information leakage were the client to prematurely leave the trusted environment.

4 Trusted Thin Terminal Architecture

The TTT architecture is focused on handling user input and display and communication with the remote server. The TTT functions as a micro-OS by polling each of these inputs and taking the appropriate actions. In developing each of these functions, we are constrained by our design decision to keep our TCB small. The inclusion of graphics libraries, USB device drivers, or network card device drivers would dramatically simplify the implementation process, but would do so at the expense of drastically increasing the size of the trusted computing base. Instead, we aimed for minimalism to create a small code base (amounting to $6,294$ lines of code) that could be verified, while still providing essential functionality for the user.

We now describe the system we used for our client and then describe the approach to add a user display, keyboard input, and network communication.

4.1 Client Machine

For our client machine, we used a Dell Optiplex 990 desktop computer with an Intel Core i5-200 processor and Intel Q67 Express chipset. We used the integrated Intel 82579LM Ethernet LAN NIC and the integrated Intel HD Graphics 2000 display adapter. We used Flicker v0.7 [33] and the Authenticated Code module (AC module) available directly from Intel termed "2nd-gen-i5-i7" [34]. We updated the BIOS to version A17 and enabled the TPM, Intel TXT, and VT-x in the BIOS. We note that a BIOS upgrade is strongly advised since older BIOS versions may not properly wipe and unlock the system's memory during an improper shutdown, rendering the system nonfunctional.

4.2 Creating the Trusted Thin Terminal Display

The most straightforward approach to implementing a video display as part of a MLE is to simply use or replicate the available display device drivers. Unfortunately, such as design decision would run contrary to our goals of a minimal TCB since the include device drivers constitute a significant segment of programming code. Further, the device drivers are hardware-specific. Even if we included the device driver for our Intel HD Graphics 2000 video card, it would only work for a subset of Intel video cards.

To address this issue, and to make our task tractable, we focused on a minimalistic display. We used the Video Graphics Array (VGA) standard text mode that provides a generic display that is supported by a large range of commercially available machines. This text mode is part of the VGA standard and is often used by some system BIOS versions to provide low-level control. While this mode constrains us to ASCII text, it allows us to demonstrate a successful remote-login system.

While the system BIOS has a special option that can place the video card into VGA mode, the process can also be invoked manually, allowing us to exclude the BIO from the TCB. To do so, we manually configure the 61 single-byte VGA registers that control the video card configuration and memory mapping. Finally, to switch to a 80 column by 25 row text display, we modify a total of 125 registers, according to available resources [35],[36].

Once in VGA text mode, each of the 256 characters can be represented by at 8x16 bit field, with each bit indicating whether a bit is present or absent. These bits are set using a set of three VGA memory planes, where the first plane represent the index of the character to be displayed, the second plane contains foreground and background color attributes, and the third plane contains the font data used to render each character. When initializing the TTT video environment, we populate the latter two planes. Then, as data is received from the remote server or user, we populate the first plane to reflect the appropriate text characters.

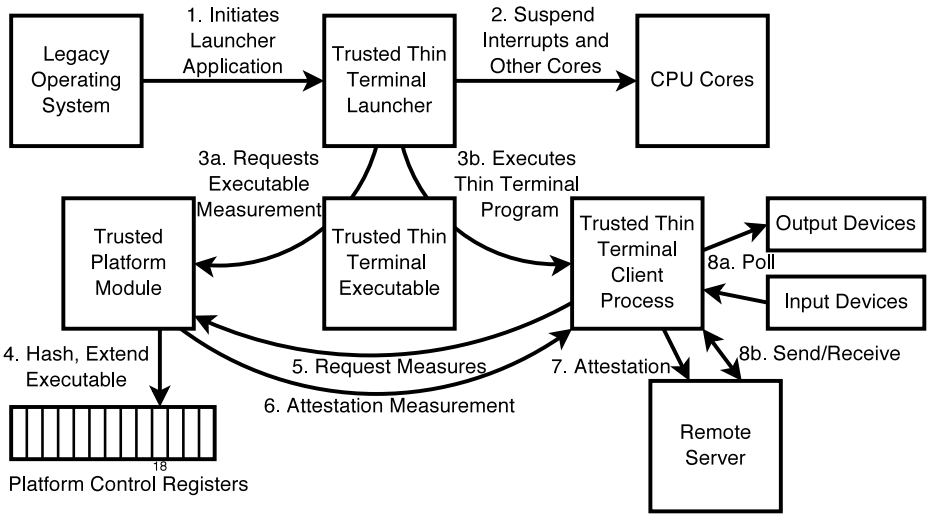

Figure 1: Process to Launch the Trusted Thin Terminal Client

4.3 Accepting Trusted Thin Terminal User Input

Given our constrained video environment, we focus on user input from the keyboard. Two keyboard types are popular in modern computers: PS/2 keyboards and USB keyboards. As with the video display environment, we must make pragmatic design decisions to support the functionality required for a thin terminal while minimizing the size of the TCB.

While USB keyboard interfaces are gaining in popularity, it is challenging to support a USB keyboard in a simple way with little additional code in the TCB. The most straightforward option is to use the BIOS USB Legacy Keyboard support interface. In this approach, the BIOS sends an interrupt when the keyboard keys are pressed. However, this method requires the inclusion of the BIOS in the TCB, which we want to avoid, while only functioning on the dwindling legacy USB keyboard support in BIOS versions. The second option is to include USB device drivers in the TTT that supports these devices directly. This option may be viable, but the complexity of the USB interface would make such implementation time-consuming. For now, we simply defer such support and focus on the PS/2 interface.

To support a PS/2 keyboard, the TTT must properly handle peripheral hardware interrupts. In single core systems, peripheral interrupts were processed by the Programmable Interrupt Controller (PIC) included as part of the device's Platform Controller Hub (PCH). When a key is pressed, the PIC receives an interrupt on one of its input pins and then alerts the CPU. To accommodate multicore processors, PIC was upgraded to the Advanced Programmable Interrupt Controller (APIC), which is now a standard component in the PCH. In APIC, each core has a Local APIC instance and a central IO APIC receives the hardware interrupts and directs it to the appropriate Local APIC instance. Accordingly, our MLE must configure each of these components, along with the CPU's Interrupt Descriptor Table (IDT), upon initialization to ensure the appropriate Interrupt Service Routine (ISR) is invoked when the CPU receives an interrupt. While the Flicker architecture provides a foundation for many of these operations, it does not provide the appropriate configuration functionality for the IDT when initializing a MLE. Accordingly, we modified the Flicker architecture to load the IDT upon initializing the MLE.

Once the TTT PAL begins execution, it disables all interrupts except for the interrupts from the keyboard. It then configures the Local APIC of the single executing processor to interrupt the CPU each time it receives a keyboard interrupt. It then updates the IDT to reflect the correct keyboard ISR inside the TTT. This ISR then obtains a scan code, which is a multi-byte representation of the keyboard key that has been pressed. The ISR must then convert these scan code notations from the keyboard into the appropriate ASCII value for display and transmission to the remote server.

4.4 Network Interface Card Support in the Trusted Thin Terminal

To communicate with the remote server, our TTT must be able to control the Network Interface Card (NIC) that is used to send communication. While we were able to create generic drivers for the video display and keyboard input, control of a NIC is inherently platform-specific. While other similar NICs may be controlled with only slight variations to the driver code, an abstract, minimalistic network card driver is an open topic.

In our implementation, we focus on supporting the Intel 82579 GbE PHY NIC present in our own system. We note that we must reuse data structures that were initially populated by the legacy OS. While we have made a concerted effort to avoid information passing between these environments, we note that the complexity of this interface is worthy of future study.

Most communication with the NIC is through the use of two circular tables, a receive descriptor table and a transmit descriptor table. Both tables are stored in the system's main memory and the NIC accesses the tables using DMA to the memory region. While the addresses of both tables are dynamic, we obtain the memory location of these tables

by accessing the PCH. The functionality of the two tables is similar, so we focus on describing the transmission table for brevity. The transmit descriptor table contains a set of structures that have configuration options and a memory reference to a buffer containing a full Ethernet frame to be transmitted.

To coordinate between previously 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 descriptor 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, to receive a packet, the TTT must allocate an empty buffer and present it to the NIC by writing a descriptor to the location indicated by the receive tail pointer and then advance 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. For convenience, the TTT also can activate checksum offloading in the NIC, causing the network hardwired to compute the checksum values for the TCP, UDP, IP, and Ethernet headers.

Once the TTT has initialized and configured the NIC, it must determine the data to be transmitted. In particular, it must construct the network layer headers and the headers and payload for all layers above the network layer in the ISO stack. In our proof-of-concept, we create IPv4 packets using UDP. We implement simplistic DHCP and DNS implementations and provide a primitive UDP datagram socket. Upon initializing, we issue a DHCP lease request, allowing us to learn our IP address and routing information, while not relying upon the legacy operating system.

The current TTT is capable of using the TPM to generate quotes that can be transmitted to a remote server to show that the proof-of-concept system functions.

4.5 Performance Discussion

In reusing the Flicker environment, we obtain performance that is similar to the Flicker late launch environment's `SK_INIT` invocation time, which was reported as around 15ms [21]. However, unlike Flicker, the late launch overheads are amortized over the length of a thin-terminal session. Accordingly, the overhead time required to transition into a thin terminal mode will be insignificant in practice.

While in the late launch environment, we essentially operate a micro-OS using a single CPU processor. Since all of our operations consist of polling for inputs and sending outputs, this lack of parallelism is not a concern. While unnecessary for a thin-terminal, future work could explore enhancements to securely re-enable additional cores to operate in the late launch environment.

5 Discussion and Future Work

We have created a basic foundation for creating a trustworthy thin terminal system within the confines of a general processing system. While the general computing system may have rich functionality to support its users, the thin-client allows the device to transform itself into a conduit for a remote login session with a trusted remote system. In doing so, the system provides strong isolation assurances, allowing the system to engage in protected communication, even if the legacy system is compromised by an adversary. Such a system can be used to support organizational "Bring Your Own Device" policies while still granting organizational IT departments strong control over the organization's assets. In particular, virtual desktop interfaces can be used to provide these devices with access to organizational resources, even on untrusted devices.

While this work represents a proof-of-concept implementation of a trustworthy thin terminal, there are opportunities to extend the work and improve its usability. In this section, we describe some future work opportunities before concluding the work.

5.1 Future Work

Enhancements to the input and output device drivers can significantly improve the usability of the approach. While our approach supports a keyboard and VGA interface for the user, these are insufficient for modern computing environments. We must support higher resolution displays with arbitrary pixel mapping support. In particular, the VGA standard does describe a higher resolution, but it is unclear if this approach would be hardware agnostic. Any implementation that would allow the TTT to simply accept a (possibly compressed) pixel value map from the remote server would likely suffice, since a similar approach has been successfully used in VNC systems.

Driver support for USB devices is also an important consideration. Support for a USB mouse and keyboard would significantly increase the coverage of modern system components. While other input devices exist, such as touch interfaces in mobile devices, the driver for each of these devices must be kept minimal to be verifiable by deployers. We note that increased driver support is a tradeoff: they enable basic functionality, but come with a security risk. As an example, a study of two years of the Linux kernel found that device drivers had a three to seven times greater likelihood of containing software defects than the rest of the kernel [37].

Finally, in future work, we can construct an authentication protocol to allow the trustworthy client to authenticate to a remote server at the system granularity. This would allow the remote server to have assurances about the operating state of the client before attempting to authenticate the user associated with the client. This would essentially allow the system to act as a second factor of authentication for the session.

6 Conclusion

The goal of this work was to provide a Trusted Thin Terminal that addresses the security issues created when a BYOD policy is allowed in a corporate setting. These issues include an asymmetrical imbalance between the trustworthiness and required trust in today's commodity operating systems, along with the challenges the general public must overcome to correctly manage the security of their devices. We proposed three design goals for the BYOD problem: 1) shift security responsibilities to IT control, 2) provide a trustworthy thin client for access to organizational resources, and 3) provide evidence to attest to the thin client's state.

To accomplish these goals, we built upon previous research using a Trusted Platform Module, a secure coprocessor providing a Dynamic Root of Trust for Measurement, to design the Trusted Thin Terminal. This allowed us to develop a proof of concept proving a user interface and network capabilities. These capabilities are essential components in developing a remote access program that interfaces with IT-controlled servers. We leveraged the Flicker architecture, with noted modifications, to provide a gateway into a secure Measured Launch Environment, in which our TTT executes. This environment allows our client software to operate in isolation from the other software on the device. The use of this late launch architecture allows the TTT to provide trustworthy authentication to a remote server, effectively addressing our goal of authentication.

Our efforts provide a proof-of-concept text-based remote client. In doing so, we describe the inherent challenges of creating a minimalistic micro-OS capable of supporting a thin terminal client. We highlight several opportunities for future work that can dramatically improve the user experience at the cost of a larger trusted code base. While the effort may be challenging, we note that the construction of well-vetted, minimalistic device drivers could empower organizations to create trustworthy thin-clients for BYOD devices.

References

[1] Coverity, "Coverity scan: 2011 open source integrity report," http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf, 2011.

[2] T. Leemhuis, "What's new in Linux 3.6," http://www.h-online.com/open/features/What-s-new-in-Linux-3-6-1714690.html?page=3, October 2012.

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

[4] Cisco, "Cisco study: IT saying yes to BYOD," http://newsroom.cisco.com/release/854754/Cisco-Study-IT-Saying-Yes-To-BYOD, 2012.

[5] 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.

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

[7] C. Scott, "Nearly a fifth of u.s. pcs have no antivirus protection, mcafee finds," http://www.pcworld.com/article/256493/nearly_a_fifth_of_us_pcs_have_no_virus_protection_mcafee_finds.html, 2012.

[8] 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.

[9] 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.

[10] J. Winter, "Trusted computing building blocks for embedded Linux-based ARM trustzone platforms," in *ACM Workshop on Scalable Trusted Computing*. ACM, 2008, pp. 21–30.

[11] Trusted Computing Group. (2013) Members. [Online]. Available: http://www.trustedcomputinggroup.org/about_tcg/tcg_members

[12] ——. (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/

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

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

[15] 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," *Internationl Journal of Information Security*, 2011.

[16] 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.

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

[18] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *ACM Symposium on Access Control Models and Technologies*. ACM, 2006, pp. 19–28.

[19] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *IEEE Symposium on Security and Privacy*, vol. 184. Oakland, CA, 1987, p. 194.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] B. Scheifler, "FYI on the X window system," IETF RFC 1198, January 1991.

[26] T. Ylonen and C. Lonvick, "The secure shell (SSH) protocol architecture," IETF RFC 4251, January 2006.

[27] T. Richardson and J. Levine, "The remote framebuffer protocol," IETF RFC 6143, March 2011.

[28] Y. Li, J. M. McCune, and A. Perrig, "VIPER: verifying the integrity of peripherals' firmware," in *ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 3–16.

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

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

[31] ——, "Attacking Intel TXT via SINIT code execution hijacking," http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, November 2011.

[32] 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

[33] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: Minimal TCB code execution," http://sourceforge.net/projects/flickertcb/files/, 2013.

[34] S. A. Sehra, "Intel trusted execution technology," http://software.intel.com/en-us/articles/intel-trusted-execution-technology, 2012.

[35] J. Neal, "Hardware level VGA and SVGA video programming information page," http://www.osdever.net/FreeVGA/vga/vga.htm, 1998.

[36] OSDev, "VGA hardware," http://wiki.osdev.org/VGA_Hardware, 2012.

[37] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM Symposium on Operating Systems Principles*, 2001, pp. 73 – 88.