Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems

Lorenzo Martignoni,* Pongsin Poosankam,*† Matei Zaharia,* Jun Han,† Stephen McCamant,*
Dawn Song,* Vern Paxson,* Adrian Perrig,† Scott Shenker,* and Ion Stoica*
*University of California, Berkeley and †Carnegie Mellon University

Abstract

Current PC- and web-based applications provide insufficient security for the information they access, because vulnerabilities anywhere in a large client software stack can compromise confidentiality and integrity. We propose a new architecture for secure applications, Cloud Terminal, in which the only software running on the end host is a lightweight *secure thin terminal*, and most application logic is in a remote *cloud rendering engine*. The secure thin terminal has a very small TCB (23 KLOC) and no dependence on the untrusted OS, so it can be easily checked and remotely attested to. The terminal is also general-purpose: it simply supplies a secure display and input path to remote software. The cloud rendering engine runs an off-the-shelf application in a restricted VM hosted by the provider, but resource sharing between VMs lets one server support hundreds of users. We implement a secure thin terminal that runs on standard PC hardware and provides a responsive interface to applications like banking, email, and document editing. We also show that our cloud rendering engine can provide secure online banking for 5–10 cents per user per month.

1 Introduction

The ultimate motivation for much of computer security is protecting access to information: preventing unauthorized users from learning or altering sensitive data. However, current systems do a poor job of end-to-end information protection: applications are divided across multiple tiers, and each tier has many points of potential security vulnerability. One of the most vulnerable tiers is the software stack on end-user personal computers. To support a wide array of applications, operating systems and libraries contain millions of lines of code and evolve constantly. This complexity inevitably leads to vulnerabilities, and bugs anywhere in the stack can open the door for malware. For instance, if you use your PC for online banking, malware in your browser, your OS, or your drivers might log your keystrokes, steal your account information, or make transactions on your behalf. Common security mechanisms that run inside the OS can only offer limited protection, as the OS itself has a large attack surface. To provide the level of security needed by sensitive applications, we need to take the user-administered desktop OS out of the trusted computing base.

One frequently proposed approach to protect sensitive applications is to run them in their own virtual machines [12, 5]. This is a good first step, because virtualization offers strong isolation, but it is too heavyweight a solution to be secure and easy-to-use on end-user systems. Multiplexing hardware between several general-purpose OS VMs requires either a virtual machine monitor that itself runs on a general OS (e.g., VMware Workstation [1]), or a hypervisor that is almost as complex as a general OS (e.g., Xen [3]), so the client-side trusted computing base (TCB) is still too large to reliably secure. Managing multiple VMs also puts an administrative burden on users: installation tasks are multiplied, users may not know which VM to use for each task, and users must roll back any VMs that get compromised. Furthermore, it is difficult to introduce a general hypervisor, such as Xen, below an existing installation of a commodity OS. Security and usability could both be improved by keeping VM-style isolation, but changing how an application is decomposed between the client and server, so that the client can be both small and general-purpose.

Thus we propose a new architecture, which we call Cloud Terminal, for protecting applications from client-side security risks. We are motivated by the observation that the client side of many security-sensitive applications is predominantly concerned with providing an interface to information, rather than performing intensive computation or high-framerate animation. Thus we propose to move application-specific computations away from the hard-to-defend end-user platform. Instead, when accessing a sensitive application, we propose to use the end-user PC simply as a secure I/O path to access application logic in the cloud.

This architecture allows for a radically simpler client-side platform: the client-side *secure thin terminal* (STT) software only needs to render graphical data from a remote application and forward keyboard and mouse events to it. The STT isolates itself from the user's untrusted OS through a small hypervisor-like layer that we call a "microvisor." Because its scope is limited to running the STT, the microvisor is substantially simpler than a hypervisor: it does not need logic for time-sharing across multiple VMs, or even drivers for network and storage hardware (instead, it tunnels encrypted data through the untrusted OS). The simplicity of the STT makes it easier to assess the correctness of the software, and facilitates remote attestation. Leveraging a hardware root of trust, the client can prove to the server that it is running unmodified STT software, directly on the real CPU. The combination of strong isolation and attestation allows our STT to be installed and to securely function on any running system, even one infected with malware.

To complement the reduced client, we move application and rendering logic into a *cloud rendering engine* (CRE), which goes beyond cloud applications like web-based office suites. The cloud rendering engine executes an application all the way to producing a bitmap image to appear on the user's screen. It then sends that bitmap to the STT over an encrypted protocol. To provide strong isolation, the cloud rendering engine for each STT runs in a separate VM, but we show that because VMs can share state, one server can support hundreds of concurrent users. Hosting the application logic in a central location also allows providers to more easily manage software updates and protect information.

We argue that Cloud Terminal has the *minimal* client-side TCB needed for accessing remote applications from an untrusted system. Any system with this goal would need code for isolating itself from the OS, capturing user input, and displaying bitmaps on the screen, but these functions make up the *majority* of the code in our STT. Thus, Cloud Terminal achieves a unique and previously unmet sweet-spot between security and generality.

To summarize, our contributions in this paper are two-fold. First, we introduce the Cloud Terminal architecture, including the secure thin terminal and cloud rendering engine abstractions, as an effective new tool for building applications with strong information security. Second, we evaluate this architecture with realistic applications. We implement a secure thin terminal that runs on standard PC hardware, providing a small 23 KLOC TCB and TPM-based remote attestation. We build a cloud rendering engine from off-the-shelf software, and show that it supports hundreds of concurrent users per server. We then evaluate this infrastructure for applications including online banking, secure document editing, and email.

2 Overview

This section gives an overview of our architecture, including use cases, our threat model, a comparison with existing systems, and an overview of the design.

2.1 Use Cases

Cloud Terminal is designed for public and corporate applications that require high information security but not intensive computation or rendering. Many use cases satisfy these properties, including financial applications such as online banking, and communication applications that let corporate users access work data from their personal devices, such as a corporate email client.

In the public service scenario, users would install a single secure thin terminal that lets them access multiple services, such as banks and financial organizations, each hosting its own cloud rendering engine in its own datacenter. Financial services are a natural use case because they are a major target of fraud, end-users and institutions both have incentives to prevent attacks, and UI requirements are simple.

In the corporate scenario, employees would install a secure thin terminal distributed by their organizations to securely access applications like email, document viewing, and document editing from their personal computers. Accessing documents via Cloud Terminal, instead of downloading them to a personal device, significantly reduces the attack surface for data theft and malware.

2.2 Goals and Threat Model

Our aim is to design a solution that significantly improves the security of sensitive applications but requires minimal effort to adopt and use. Specifically, we seek to meet the following goals:

1. The solution should be installable on existing PCs alongside a potentially compromised commodity OS, without requiring the user to re-image her system.
2. The solution should not require trust in the host OS.
3. The solution should be able to attest its presence to both users and application providers, to protect against spoofing and phishing.
4. The system should support a wide range of sensitive applications.
5. The TCB of the system should be small.

We assume an adversary that controls the OS on the user's PC and can intercept all its network traffic, but does not have physical access to the machine to mount hardware attacks like cold-boot memory recovery [13]. We also assume that the attacker cannot reliably infer the user's input from sensors on the machine (e.g., by listening for keystroke timings on the microphone). Our goal is to prevent the attacker from viewing and modifying in-

Property	Red/Green VMs [18]	Per-app VMs [12, 29]	Browser OS (e.g. Chrome OS [7])	Virtual Desktops & Thin Clients	Flicker [20]	Cloud Terminal
Installable below an existing system	✘	✘	✘	✔	✔	✔
Remote attestation	✘	✘	✘	✘	✔	✔
Generic applications	✔	✔	✘	✔	✘	✔
Fine-grained isolation	✘	✔	✔	✘	✔	✔
No trust in host OS	✔	✔	✘	✘	✔	✔
User interface	arbitrary	arbitrary	browser only	arbitrary	✘	arbitrary
Management effort	medium	high	low	low	low	low
TCB size	>1MLOC	>1MLOC	>1MLOC	>1MLOC	250 LOC + app logic	23 KLOC

Table 1: Properties of Cloud Terminal compared to other security architectures.

teractions between the user and a secure application, and from logging into it as the user.

Cloud Terminal also protects against some social engineering attacks, such as users being tricked to run a false client, through remote attestation. In addition, it provides two defenses against phishing: a shared secret between the user and the secure thin terminal, in the form of a graphical theme for the terminal's UI, and the ability to use the user's TPM as a second, un-phishable authentication factor and detect logins from a new device. These mechanisms are similar to common mechanisms in web applications (e.g., SiteKey [32] and cookies for detecting logins from a new machine), with the important distinction that the secret image and the TPM private key *cannot* be retrieved by malware or by man-in-the-middle attacks. Nonetheless, we recognize that there are open problems in protecting users against social engineering and we do not aim to innovate on this front, as the problems are orthogonal to our focus on designing a well-isolated client.

Finally, because Cloud Terminal must coexist with the untrusted OS, it is not designed to prevent denial-of-service attacks from the untrusted OS: for instance, the untrusted OS could prevent the installation of the client in the first place, or block its network traffic.

2.3 Existing Approaches and Comparison

Although many architectures to improve the security of end-user systems have been proposed, it is challenging to simultaneously meet all the goals identified in the previous section. In this section, we compare several existing proposals (Table 1) and explain the elements of our approach that let it meet the goals.

One frequently proposed approach is to isolate sensitive applications using virtual machines, either by having a "red" VM for untrusted applications and a "green" VM for trusted ones [18], or one VM for each sensitive application [12, 29]. While VM-based solutions provide strong isolation, any hypervisor aiming for wide deployability needs a TCB comparable in size to an OS kernel, including drivers for all the the network and storage devices common on consumer PCs. Attacks targeting these components in popular hypervisors have already been demonstrated [17, 23, 41]. In addition, we need to include the applications inside the trusted VM (e.g., a web browser) in the TCB. Finally, VM-based systems are not readily installable below an existing commodity OS without requiring the user to reimage his machine, and they increase the administrative burden on users by requiring users to manage each VM separately.

In contrast, browser OSes like Chrome OS [7] limit their attack surface by disallowing binary applications and provide strong isolation between web applications. However, this means that they cannot run the user's existing legacy software or access non-web applications.

The approach closest to ours is virtual desktop infrastructure (VDI), where users access centrally managed virtual desktop VMs through thin client software [37]. While thin client systems allow organizations to centrally manage their desktops and remove infections, they still suffer from a fundamental limitation compared to Cloud Terminal, in that all of the user's applications run in the *same* VM and are not protected from each other. For example, a user's VM might become infected through a drive-by download from personal web browsing, which would then put all other applications at risk. Another limitation of current VDI systems is that the thin client software runs within the untrusted OS on the user's PC and is unprotected from malware on that machine.

Finally, remote attestation is challenging in VM-based approaches, browser OSes and VDI because of the wide range of software that can run on the client. For example, even if a system could prove that it is running a particular OS kernel and only a trusted set of binaries, the system could still be executing malware in the form of a shell script or a malicious browser extension. Thus, these approaches miss an opportunity to provide powerful security guarantees via TPM attestation. One example of a system that *does* support attestation is Flicker [20], which allows applications to run small pieces of application logic (PALs) in an isolated, attestable environment,

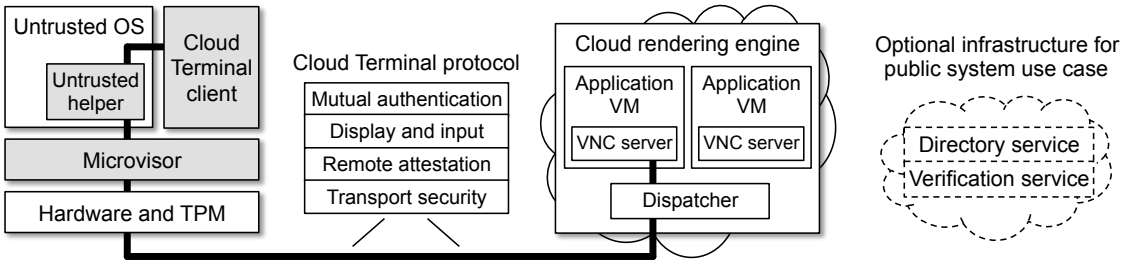

Figure 1: Cloud Terminal architecture, showing the secure thin terminal (left) and cloud rendering engine (right). The shaded areas make up the secure thin terminal.

for purposes such as key signing and password handling. However, executing a client for a remote application as a Flicker PAL would be difficult, because a PAL cannot interact with the user through the GUI, as Flicker suspends the untrusted OS during PAL execution.

Cloud Terminal achieves deployability on existing systems, support for general applications, remote attestation, and a small TCB through two design elements:

- **Small, general client:** Cloud Terminal accesses all sensitive applications through the same, simple component: a secure thin terminal capable of displaying arbitrary remote UIs. Thus, the user does not need to manage multiple VMs, and service providers can run their applications in their own datacenters under tight control. Unlike in virtual desktop systems, the sensitive applications are also isolated from each other (rather than running in the same VM), and the thin terminal is protected from the untrusted host OS.
- **Microvisor:** The secure thin terminal isolates itself from the OS through a hypervisor-like layer, but this "microvisor" is substantially smaller than a full hypervisor because it is not designed to run multiple VMs. For example, the microvisor accesses the network and storage devices through the untrusted OS (but it encrypts its data), leveraging the OS's existing drivers without having to trust them. Likewise, it does not need code for managing multiple VMs, or even for booting the untrusted OS; it can install itself below a running instance of the OS, and only needs to protect an area of memory from the OS.

This design lets Cloud Terminal achieve a sweet-spot between security, trusted code size, and generality: it can access a wide range of applications through a small, well-isolated, and remotely verifiable client.

2.4 Architecture

Figure 1 shows the overall Cloud Terminal architecture. Our approach centers on two abstractions: the *secure thin terminal* on the client and the *cloud rendering engine* on the server. We now describe these abstractions and show how they interact with other system components.

Secure Thin Terminal (STT). The STT is software that runs on a user's computer and provides secure access to a remote application, without requiring trust in any other software on the device. The STT temporarily takes over a general-purpose system, and turns it into a more limited but trustworthy device for accessing generic remote applications. The STT has the following features:

- The STT provides a common graphical terminal functionality that can be used by many applications.
- The STT isolates itself so that the untrusted system cannot access its data.
- The STT implementation is lightweight, making it easier to check its correctness.
- Using a hardware root of trust, the STT can remotely attest that it is running unmodified on real hardware.

The security of the STT comes from its simplicity: it focuses solely on providing an interface to applications running elsewhere. It provides this interface simply by relaying input events and remotely-rendered bitmaps. The STT co-exists with a pre-existing untrusted OS, but does not rely on the untrusted system for any security-critical functionality. Using hardware virtualization, the STT isolates itself from the untrusted OS: the OS never has unencrypted access to the STT's data, and cannot read input events or access the video memory when the STT is active. A hardware root of trust allows the STT to prove to remote parties that it has complete control of the machine, namely that its code is unmodified and that it has direct access to a real (non-emulated) CPU.

The STT consists of the *microvisor*, which provides isolation from the OS; the *Cloud Terminal client*, which communicates with the remote application and renders its display; and an untrusted user-space *helper* that tunnels encrypted data through the untrusted OS.

Cloud Rendering Engine (CRE). The CRE is STT's server-side counterpart. It has the following attributes:

- The CRE contains almost all the application functionality, to the point of producing bitmaps to display.
- The CRE runs an isolated instance of the application for each STT, in a separate virtual machine.

- CRE VMs run a minimal software stack needed to render the remote application, rather than allowing the user to install her own software.
- CREs are managed centrally by the application provider, facilitating administration and protection.

From some uses, such as document viewing, the CRE can be almost completely self-contained. For other uses, the CRE can provide access to a service on other machines, such as an existing web application, by running standard rendering software such as a web browser.

Our CRE implementation executes the application instance for each user session in a separate VM to provide strong isolation. These VMs run a stripped-down commodity desktop environment, connected to an unmodified VNC server that we proxy through a *dispatcher* for access by the appropriate STT. The main challenge in implementing the CRE is scalability: the system needs to be able to support a large enough number of users per server to be cost-effective. We employ aggressive sharing of disk and memory pages between application VMs to support several hundred concurrent users per CRE server.

Finally, although the CRE hosts a "cloud" of VMs, we expect it to run in a *private* provider-owned datacenter.

Cloud Terminal Protocol. The secure thin terminal and cloud rendering engine communicate over the network using the *Cloud Terminal protocol*. The Cloud Terminal protocol extends an existing framebuffer-level remote desktop protocol (VNC) by adding additional levels of security. Specifically, the Cloud Terminal protocol uses end-to-end encryption between the Cloud Terminal client in the STT and the CRE, performs remote attestation of the client, and provides mutual authentication between the user and application.

Public Infrastructure Services. If Cloud Terminal is deployed as a public service for accessing multiple applications via a single STT (e.g., financial websites that enroll in the system), rather than as a private service within a corporation, it must also host two infrastructure services. The *directory service* provides the client with a list of CREs to connect to (e.g., the CREs for various online banks). The *verification service* lets users check that they installed a genuine STT. Nonetheless, even in this use case, each application provider still hosts its own CRE on its own hardware. We describe this usage scenario in more detail below.

3 Secure Thin Terminal

The secure thin terminal has three components. The first is the *microvisor*, a small hypervisor-like layer providing isolation from the untrusted system and simple APIs for the following functionalities: keyboard and mouse input, video output, sealed storage, and networking. The microvisor can also attest (i.e., cryptographically prove)

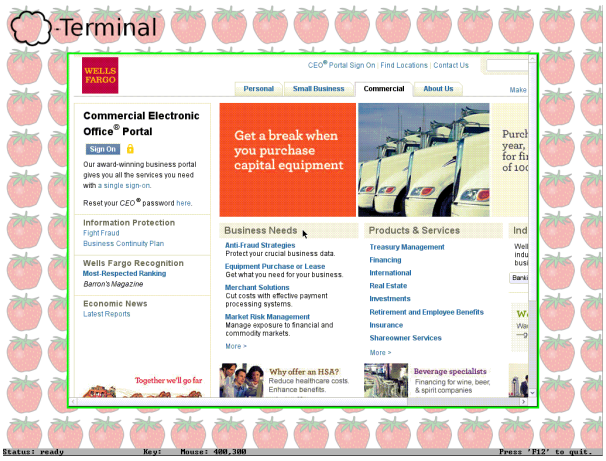

Figure 2: Screenshot of the STT. The textured border around the browser (the strawberries) is a secret image configured by the user that will only be shown correctly by the genuine STT.

the integrity of its code to a remote third party through a *measurement* (a signed hash) from the TPM. The second component is the Cloud Terminal client, which runs within the microvisor. The third component is an untrusted user-space helper to which we delegate the handling of networking and storage API calls. We refer to this subset of the API calls as *untrusted API calls*. Through the helper, we can exploit the drivers present in the untrusted OS, and thus use any type of network controller (including WiFi and 3G cards), without having to take over and configure the network hardware.

The secure thin terminal can be installed on a system at any time, even if malware is present. Once the STT has been installed, the user can bring up the Cloud Terminal client by pressing a designated secure attention key (e.g., Ctrl-F12). While the client is running, the untrusted OS and its applications continue to run, but blindly: we prevent them from seeing the user's inputs and the contents of video memory. Figure 2 shows the STT's UI.

3.1 Microvisor

The simplicity and small code size of the microvisor come from its limited scope (e.g., no network and storage drivers) and from leveraging hardware virtualization support (Intel VT [22, 14]). For attestation, we employ Intel's trusted execution extension, TXT [15]. The combination of hardware-supported virtualization and trusted execution allows us to establish a tamper-proof, measured execution environment.

When the Cloud Terminal client is not running, the microvisor is responsible only for isolating itself from the untrusted OS and applications. Specifically, the microvisor makes its address space inaccessible by preventing the untrusted system and the I/O devices from accessing those ranges of physical memory. Additionally, the

microvisor intercepts keystrokes to detect when the user requests to bring up the Cloud Terminal client. It does so by trapping reads made by the untrusted system to the PS/2 port, emulating the read, and updating the state of the untrusted system accordingly, as if the read had been performed directly. As we shall describe, we use a similar approach to capture the user's input securely when the Cloud Terminal client is running. For video output, the microvisor maps the video memory into its virtual address space and then renders images on the screen by directly writing to the memory.[1]

In our current implementation, the microvisor is installed after the untrusted OS has started running; nevertheless, it assumes complete control of the system, in a similar manner to malicious hypervisors [9, 28]. This approach does not require any prior setup of the system.

To securely install the microvisor, we use code from the Flicker secure execution infrastructure [20], which in turn uses Intel TXT [15]. This primitive allows us to establish a dynamic root of trust for measurement and attestation: it ensures that the code for performing the installation of the microvisor and the code of the microvisor cannot be tampered with (the code is executed atomically and it is stored in a region of memory that is completely isolated, even from hardware devices) and stores a measurement (hash) of this code in the TPM. The Flicker-attested code also generates a key pair whose private component is kept only by the microvisor, linking future communications to the attested execution. The private key is kept in volatile RAM whose contents will be lost if the untrusted OS forces a reboot, so no other code can masquerade as the microvisor.

We start the installation by saving the current state of the untrusted system so that we can later resume it as a "guest" of the microvisor. Then, we invoke the `senter` instruction and run the code performing the necessary steps to enable the microvisor. Finally, we resume the execution of the untrusted system from where it was interrupted. From this point on, the microvisor has full control of the system and it is responsible for isolating itself from the untrusted OS. As described in Section 5, the measurement stored in the TPM is used to prove to the cloud rendering engine that the client is genuine.

3.2 Cloud Terminal Client

The Cloud Terminal client is essentially a process that runs in the context of the microvisor and interacts only through the microvisor's API. The client starts by making a backup of the contents of video memory and ends with restoring these contents and redrawing the screen. The rest of the execution implements the Cloud Terminal protocol. Untrusted API calls and API calls to get user input block the client and resume the execution of the untrusted OS (without access to the video memory, as we describe later). The client is then resumed when a blocking call returns. To ensure that no sensitive data can be accessed outside the Cloud Terminal client, before and after an untrusted API call the client respectively encrypts the input arguments and decrypts the output arguments. Data transmitted over the network are encrypted using the shared session key established during the initial stage of the protocol. Data stored on disk are encrypted with a symmetric key that we store persistently in the TPM using *sealed storage* [20], ensuring that the key can only be retrieved by a genuine STT.

3.3 Securing the Execution of the Client

During the execution of the client, the microvisor transparently dispatches untrusted API calls to the untrusted helper. Since the untrusted system must continue to run while the client is running, we have to ensure that an attacker cannot see the video output of the secure thin terminal and that he cannot sniff inputs coming from the keyboard and the mouse.

To read keystrokes and mouse events from the client, we intercept and emulate reads from the PS/2 port, but do not deliver sensitive events to the untrusted system. To prevent an attacker from seeing what is being rendered on the screen, we hijack the virtual mapping of the video memory in the untrusted system. Specifically, we configure the MMU to redirect accesses to the memory region mapping the video memory to the region that we use as a backup. Then, we ensure that the untrusted system cannot map the real video memory. When the Cloud Terminal client is terminated, we restore the original mapping, content, and permissions of the video memory.

3.4 Untrusted User-Space Helper

The helper program runs in user-space in the untrusted system and is very simple: it provides basic networking and storage capabilities but is aware of the data it manages. Since sensitive data do not leave the microvisor unencrypted, a compromised helper cannot violate data confidentiality or integrity.

The helper and microvisor share a memory region. The microvisor makes requests by writing to the region, and the helper signals completion using a hypercall (the `vmcall` instruction, similar to a system call). The helper using polling, with a frequency set by the microvisor, to avoid the need to modify the untrusted OS.

4 Cloud Rendering Engine

The cloud rendering engine (CRE) runs instances of an application to render bitmaps displayed by the STT. It

[1] We currently do not support USB keyboards and mice, though we plan to do so in the future. We also require the host OS to be configured with hardware graphics acceleration disabled so that we can correctly manipulate its graphics state.

consists of a *dispatcher* that accepts connections from STTs and one *application VM* for each session. The software to run in each VM is chosen by the service provider, but may include a web browser to render an existing web application, a word processor for secure document editing, or in-house enterprise applications.[2]

When a client connects to the CRE, the dispatcher assigns a VM cloned from a base image to run its application instance. We run a remote desktop server, such as a VNC server, inside each VM to render the application's UI to bitmaps, and send these to the client.

4.1 CRE Scalability

The main challenge in designing the CRE is scalability: for cost efficiency, we wish to support hundreds of application VMs per CRE server. We leverage two properties of the CRE workload to achieve this. First, interactive applications spend most of their time waiting for input, allowing for statistical multiplexing of CPU and network resources. Second, because the VMs all run the same software, they can share a high fraction of memory [38, 40, 39]. Although the specific sharing mechanisms we use are not new, our contribution is the observation that these techniques work especially well for a CRE workload, providing far higher savings than in traditional server consolidation, and thus making CREs cost-effective.

By leveraging these features of the workload, we can support several hundred concurrent users running rich desktop applications, such as Firefox, on a single commodity server at a cost of few cents per user-hour (see Section 6.3). We describe the key optimizations below.

Memory sharing. We coalesce identical memory pages from the application VMs into a single page with copy-on-write behavior. We found that this can reduce the footprint of each guest VM by 38% to 61%.

Disk sharing. Each VM uses a copy-on-write disk image based on a single master image. This minimizes the number of extra blocks needed per VM and keeps the base image in the server's buffer cache for fast access.

Stripped-down OS. The guest VMs run only the software needed to support the desired application.

Reduced timer interrupts. We lowered the CPU usage of our guests substantially by configuring them to run a "tickless" Linux kernel, which uses ACPI to set timers instead of needing a periodic 100 Hz interrupt [31]. This change reduced the CPU usage of each idle guest from 10% of a core to less than 1%.

[2] We used VMs for isolation instead of lighter-weight containers to show that high scalability can be achieved even with strong isolation.

4.2 CRE Security

The CRE provides significantly more protection to the application than an operating system on the user's machine because it accepts only sessions from attested STTs, receives only keyboard and mouse input from these clients, runs up-to-date software chosen only by the service provider, and does not expose guest VMs to the Internet. Nonetheless, there is a risk that users running sessions on the same CRE server could interfere with each other. We mitigate this risk in three ways:

Network isolation. The client VMs run on separate virtual networks behind a NAT, so that they cannot send traffic to each other. We also restrict their access to the external network using a firewall, so that they can only communicate with servers necessary for their applications (e.g., a web server for the banking use case).

Resource isolation. We use standard VM isolation features to cap guests' memory, disk, network and CPU use.

Restricted user environment. The applications in each guest VM run on a Linux user account with minimal privileges, inside a desktop environment that does not let the user launch any other applications.

In the end it may still be possible for a malicious user to subvert the software running inside a CRE VM, such as a web browser or the Linux kernel, but the CRE should have no more authority that the user it is assigned to. Even if the user gained full control of her VM, she would not be able to access other users' VMs over the network. Cross-VM information leakage [27] may be a concern, but we believe that the information gain from such leakage is limited, especially because of the high number of VMs running on each CRE server.

5 Setup and Session Protocols

We now discuss how users install Cloud Terminal (Section 5.1) and how the STT and the CRE communicate (Section 5.2). We focus on the first use case in Section 2.1, where the user installs a single STT to access multiple public applications, such as banking sites. The private deployment case is similar. We include some extensions to the protocol, such as multi-factor authentication and copy-and-paste support, in an extended version [11].

5.1 Cloud Terminal Installation

The STT installation process seeks to achieve two goals: (1) certifying to the user that she is installing a genuine STT and (2) establishing a shared secret between the STT and the user, so that the user can check whether she is accessing the real STT in the future.

For the first goal, we include a *verification service* as part of the public Cloud Terminal infrastructure that communicates with users through a secondary channel (a phone) to let them verify their STT. When the STT

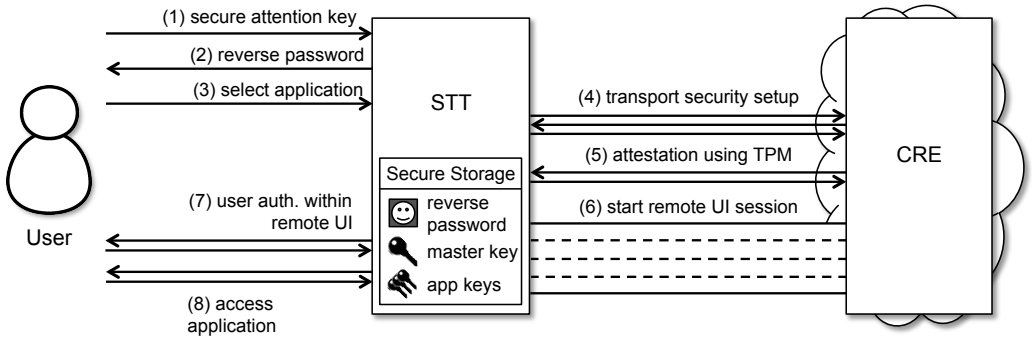

Figure 3: Process for a Cloud Terminal session. The user presses a key (1) to open an STT UI authenticated by the reverse password (2). She then picks an application (3). We use an application key to set up a secure transport session (4), attest the validity of the STT using the TPM (5), then set up a remote UI session (6). The user logs in through this UI (7) to access the application (8).

starts for the first time, it connects to the verification service, attests its presence on the machine using the TPM, and receives a random nonce value from the verification service which it displays to the user. The user then calls the verification service at a well-known number, enters the nonce, and is told whether the nonce corresponds to a correctly attested STT. Due to space constraints, we discuss this process in more detail in [11].[3]

For the second goal, we have the user select a background image to be shown on the STT's UI, which we call a *reverse password* because its role is to authenticate the software to the user rather than the user to the software [32]. The STT keeps this image in sealed storage, so that malware cannot obtain it and phishers cannot guess it. However, because users have been shown prone to social engineering attacks against similar authentication schemes in web applications [30], the reverse password is *not* our only defense against phishing; as we will discuss in Section 5.2, applications can also use the TPM's private key can as a second authentication factor to detect logins that are not from the user's device.

5.2 Session Protocol

Once the user has installed the STT, she can launch its UI at any time by pressing a special key intercepted by the microvisor. This UI displays a list of available applications that the STT obtains from a *directory service* managed by the Cloud Terminal provider. We expect providers to require substantial verification before adding applications to this directory (e.g., similar to Extended Validation SSL certificates). The STT connects to the directory service securely using a *master public key* for the service stored in the STT binary, and gets back an *application public key* for each application.

Once the user selects an application, the STT opens a session to it through the Cloud Terminal protocol. This is a simplified TLS-like protocol that also uses the TPM to attest that the user is running a valid STT, and shows a remote UI through a subset of VNC. Through the remote UI, the user can log into the service as usual (e.g., with a password). Overall, session setup consists of the following steps, also shown in Figure 3:

1. The user presses a special key to bring up the STT.
2. The user checks that this screen displays the correct reverse password image to authenticate the STT.
3. The user selects the application to access.
4. The STT connects to a CRE for the application using its known DNS name, by tunneling network access through the helper in the OS. It uses the application's public key to authenticate the server and sets up an encrypted connection through a subset of TLS [10].[4]
5. The CRE verifies that it is talking to a valid STT through TPM attestation: it sends a nonce to the STT and gets back a combination of the nonce and a hash of the executing code, signed by the TPM. The STT also sends the CRE a hash of the key of the application it connected to, also signed by the TPM, so that a malicious CRE cannot proxy a session to another CRE by forwarding its nonce.
6. The CRE renders the application's UI through a framebuffer protocol similar to VNC.[5]

[3] One challenge with this process is knowing that the STT that contacted the verification service is running on the *user's* machine. Parno observed that any verification process with current TPMs is vulnerable to a "cuckoo attack" where an adversary gives the user a trojan horse to install but runs a real TPM on a different machine and relays its UI to the user [24]. He proposes several solutions, the most attractive of which is to have PC vendors engrave a hash of the TPM's public key on each device. The user could then confirm this hash to the verifier by sending a verified-selected subset in an SMS message. Another measure that reduces the viability of this attack is to only accept TPM keys signed by a hardware manufacturer and to only accept each key once, so that the adversary must acquire a new TPM chip for each attack.

[4] We used RSA for key exchange, 128-bit AES CBC for symmetric encryption, and HMAC-SHA256 for the MAC.

[5] We chose a subset of the RFB protocol [26] used by VNC. RFB

7. Within this remote UI, the application displays an interface of its choice to authenticate the user.

In addition to letting the CRE verify that it is communicating with an unmodified STT running on the physical CPU, the TPM attestation step can also be used by applications to defend against phishing. The key idea is that even if an attacker obtains the user's account name and password, she cannot obtain the private key of the TPM on the user's device: even the user herself does not know it! Thus, the application can remember the TPM public key it saw for each user and treat sessions that employ a different TPM as suspicious, asking the user to confirm that they are using a new device through a secondary channel like SMS. Many current web applications similarly detect logins from a new device using browser cookies, but the TPM private key has the advantage that it cannot be read by malware, unlike cookies.

6 Implementation and Evaluation

To evaluate the Cloud Terminal architecture, we have implemented a secure thin terminal that runs on standard PCs and a cloud rendering engine based on off-the-shelf software. This section describes our implementation (§6.1) and four applications: banking, secure document viewing, document editing, and email (§6.2). We then evaluate the system's performance (§6.3) and estimate the cost for a provider to run Cloud Terminal (§6.4).

6.1 Implementation

Secure Thin Terminal. We have implemented all the client components of our architecture. All components are available for Linux, but the attestation in Linux is currently only partially secured (i.e., not all the code of the system is measured). For simplicity, the address space of the Cloud Terminal client is currently not isolated from the address space of the microvisor, though this could be achieved by running the client in an unprivileged ring. We have not yet implemented support to protect the STT from malicious device DMAs (using VT-d) or from malicious SMI handlers. Also for simplicity, our implementation of the RFB protocol supports only raw image encoding and does not perform any compression.

Our trusted computing base consists of 20.5K lines of C and 1.4K lines of assembly, distributed as follows. The microvisor consists of 7K lines of C and 0.7K lines of assembly. The client consists of 3K lines of C (excluding two large array initializers for a bitmap font and the system logo). Our cryptographic routines are from the PolarSSL library [25] (we borrowed 5.5K lines of C) and our TPM based attestation code is based on Flicker (5K lines of C and 0.7K lines of assembly). (Measurements are all non-comment-non-blank physical lines.)

explicitly allows clients to support a subset of protocol options, which allows our cloud rendering engines to use an off-the-shelf VNC server.

Operation	Time (msec)
PCR Extend of Flicker (PCR-18)	3.43
Key Generation	24.25

Table 2: Time in milliseconds required for some of the operations in the attestation protocol of the Thin Client over averaged over 100 trials. All of the operations occur within the TCB.

We tested the Linux implementation of the secure thin terminal on a Lenovo W510 laptop. This laptop has a PS/2 keyboard and mouse, supports virtualization technology and trusted execution, and includes an STMicroeletronics TPM.

Table 2 depicts some of the main operations that are necessary in performing the remote attestation. The Thin Client is responsible for performing these operations within its TCB.

Cloud Rendering Engine. We implemented the CRE on Linux, using KVM [16] as our virtual machine manager. We leverage Linux's Kernel Samepage Merging (KSM) daemon [2] to share identical memory pages. We keep a pool of fresh VMs for new sessions to hide startup times.

For our guest OS, we used a Debian GNU/Linux 6 installation with minimal packages to run the desired application (e.g., Firefox), a simple window manager (XFWM) and a VNC server. Our experiments used a resolution of 800x600 pixels and 8 bits-per-pixel of color depth, which we found sufficient for a single application (see Figure 2). We also configured the window manager to prevent the user from launching other programs.

6.2 Applications

We built four applications over Cloud Terminal: online banking, document viewing, document editing, and secure email. Very little effort was required to implement these applications: two give access to existing web applications, while the other two use existing Linux software.

Online banking. We built a CRE for the Wells Fargo website, which displays the existing site using Firefox. We run Firefox in kiosk mode: the page fills the entire desktop and the user cannot execute other applications. We block off-site resources and links with an HTTP proxy. The bank can easily configure a whitelist for this proxy.

Document viewing. As an example of a corporate application for Cloud Terminal, we built a CRE for viewing sensitive PDF documents via Evince, a Linux PDF viewer. We currently show a local document in each VM, though it would be easy to give the VMs access to the company's network file system after users authenticate.

Document editing. In addition to viewing documents, we let users edit them through a CRE that runs AbiWord, a word processor compatible with Microsoft Word.

Application	Activity	Baseline (ms)	STT (ms) with # of concurrent clients =						Network Usage (bytes)	
			1	50	100	150	200	300	inbound	outbound
Doc. editing	Launch app.	2,844	1,732	2,000	2,033	2,208	2,441	2,553	487,047	3,888
	Type a key	30	50	52	51	53	50	54	1,607	346
	Move the mouse	32	47	48	53	49	59	51	480	138
Doc. viewing	Launch app.	1,699	1,817	1,947	2,066	2,093	2,147	2,493	483,219	2,040
	Scroll a page	114	1,175	1,208	1,281	1,270	1,380	1,704	352,358	5,497
Online banking	Launch app.	6,911	1,581	2,047	2,232	2,319	2,563	____	490,149	4,680
	Browse a new page	1,183	2,302	2,516	2,573	2,610	2,661		415,732	10,939
Secure email	Launch app.	6,936	1,865	1,992	2,054	2,254	____	____	488,367	3,954
	Display an e-mail	992	1,976	2,160	2,106	2,254			318,300	8,416

Table 3: End-to-end UI latency and network traffic at the STT for various actions. Measurements are averaged over 30 runs.

Secure email. As an example of a richer web application, we also used Firefox to provide access to Gmail. For example, an organization might implement this service to let employees access sensitive email from their home PCs. Although Gmail is more CPU-intensive than our other applications, we could still support more than a hundred concurrent sessions on one server.

6.3 Performance Evaluation

We sought to answer two questions with our performance evaluation. First, how responsive is the STT as a means for accessing remote applications? And second, how far can a CRE scale while providing a good user experience? We omit an evaluation of time to launch the STT and overhead imposed by the microvisor on the user's OS, because these overheads are negligible and have been measured in detail for similar primitives [19].

To answer these questions, we ran a CRE on a 16-core server with 2.0 GHz Opteron processors and 64 GB RAM. We then connected up to 300 emulated clients to it to generate load. These clients replayed packet traces from our STT implementation to loop a 3–5 minute recorded UI session.[6] Finally, we connected a "probe" STT instance running on a laptop that we interacted with manually to measure UI responsiveness. This client experienced a 23 ms network latency from the Berkeley campus network to our CRE hosting provider in Seattle.

In summary, our results show that each of our applications scaled to 150–300 simultaneous sessions on a single server while providing a responsive experience. Furthermore, the network bandwidth per session was well within the means of current ISPs. The main limiting resource on the CRE server was the CPU. We stopped scaling each application when its CPU load reached 90%.

6.3.1 Qualitative Usability

Most importantly from a usability perspective, the system felt usable qualitatively. We were able to type paragraphs of text unhindered by the refresh speed (typing latencies were similar to SSH), and to comfortably navigate through the applications. The slowest action was scrolling the page, which we have not yet optimized.

6.3.2 Client-side Metrics

To quantitatively demonstrate the usability of STT, we measure the end-to-end UI latency at the client for various user activities, such as typing a key, navigating to a new page, and loading an application. End-to-end UI latency for a user activity is the amount of time taken from when the user begins an action and until the system finishes rendering the result of that action. Table 3 presents the average UI latency on an STT client when running with different levels of CRE server load, as well as the amount of bandwidth used. We compare these measurements against a baseline where we run the application locally. For startup time, this baseline is after a reboot, so it includes the time to load the program from disk.

The results in Table 3 show that the latency introduced by the STT is low for most activities. For keystrokes, Cloud Terminal adds up to 24 ms of latency even when the CRE is serving 300 concurrent users, bringing the total latency up to 54 ms. This is substantially lower than the average inter-keystroke timing of 100 ms [33]. For activities that refresh a large part of the screen as opposed to displaying one new character, like navigating to a new page, we see between 1.0 and 1.6 seconds of extra latency. Much of this is due to unoptimized rendering and image compression that we believe can be improved without substantial effort (e.g., we can speed up scrolling by sending a command to translate part of the image).

Interestingly, launching an application for the first time was often *faster* via Cloud Terminal than locally, because the application was pre-loaded in a CRE VM.

6.3.3 Server-side Metrics

We report the load on the CRE server running each application with varying numbers of clients in Table 4. We see several interesting trends. First, CPU was always the

[6] The sessions were: opening an account on Wells Fargo, reading a PDF, editing a Word document, and reading/writing emails.

Application	Document editing			Document viewing			Online banking			Secure email		
and # clients	100	200	300	100	200	300	100	150	200	50	100	150
CPU load	27%	54%	84%	34%	66%	96%	37%	60%	81%	28%	57%	84%
Mem. GB	8.4	15.1	22.1	8.4	15.7	25.2	25.5	38.3	47.6	15.1	26.4	38.1
KB/s in	367	708	1067	286	553	769	1059	1607	1931	251	392	591
KB/s out	2054	4360	8951	2993	5562	7709	2568	3527	4395	1802	3468	4888

Table 4: CRE load for each application with various numbers of concurrent clients.

limiting resource for our server configuration. This includes both the cost of encrypting traffic and of running the application itself. Second, both CPU and memory usage were 2–4× higher for the browser-based applications (Wells Fargo and Gmail) than the standalone Linux applications. Gmail was especially CPU-intensive due to its heavy use of JavaScript. Nonetheless, even these applications were able to scale to 150 and 200 sessions on a single CRE server, while running in Firefox 3. Finally, bandwidth usage stayed below 9 MB/s in total, or, on average, at most 32 KB/s per session, which is within the capabilities of both home ISPs and hosting providers.

6.4 Cost Analysis

We estimate the per-user cost of running a Cloud Terminal service based on our measurements and on hosting prices at one provider (CariNet [6]). CariNet offers a 12-core server with 40 GB RAM and unmetered 100 Mbps connectivity for $1010/month. Assuming that this server can run 3/4 of our 16-core load, it can host 82,000 to 164,000 user-hours per month. This makes the overall cost between 1.2 and 2.5 cents per user-hour. For an online banking service, the average user is unlikely to log in for more than 2 hours per month (based on an informal poll of our group), making the cost up to 5 cents per user per month. For a corporate application, the cost is at most $3 per employee per month even if the employees use the service 8 hours per day.

7 Related Work

Section 2.3 has already compared Cloud Terminal to several previous approaches for secure access to applications, including systems that isolate applications using VMs [18, 12, 29], browser OSes [7], thin clients [37], and Flicker [20]. Cloud Terminal distinguishes itself from these systems by simultaneously providing five properties: installability under an existing (potentially compromised) OS, remote attestation, support for general applications, isolation across applications, and a small enough TCB to make verification feasible (23 KLOC). The main insight allowing this is the choice of a much simpler, but still general client: a thin terminal. This client is simple enough to remotely attest yet capable of displaying arbitrary UIs.

Several other related projects include Tahoma [8], a browser OS that isolates web applications using virtualization, and IBOS [35], a microkernel-based browser OS. Both approaches reduce the trusted code base on the client (although, unlike Cloud Terminal, they must include drivers for network and storage devices in the TCB), but they are limited to protecting web applications and they do not provide remote attestation. Proxos [34] partitions a system call interface so that a "private application" has certain requests serviced a VMM-isolated "private OS" and others by a commodity OS, similarly to how the STT interacts with some devices directly and others via the untrusted helper. One application of Proxos is to protect a web browser, but for this application the TCB includes both Xen and an X server.

Remote attestation has been explored by several projects, including Tboot, which can perform a measured and verified bootstrap of an OS or of a hypervisor [36], and TrustVisor, a hypervisor that relies on hardware attestation to ensure code integrity and secrecy for selected portions of an application [19]. In contrast to these systems, Cloud Terminal can run an entire, interactive UI session in an attestable execution environment.

Several projects have looked at small-TCB approaches to secure sensitive inputs to online services, such as credit card numbers and passwords, in the presence of malware. Bumpy [21] allows users to type a secure attention sequence to encrypt input for a web service, but takes a hardware approach (using an encrypting keyboard) in contrast to our software approach. The Trusted Input Proxy (TIP) [4] uses a hypervisor and a separate VM to pop up dialog boxes for sensitive input whose responses are injected in a TLS stream. However, TIP does not provide confidentiality for the whole UI session—for example, malware can still see the account statements sent back by a bank or the contents of a document being edited. In contrast, the STT hides an entire UI session.

In concurrent work, Zhou et al. [42] describe how to protect trusted I/O paths from device-level attacks such as overlapping memory-mapped I/O and spoofed interrupts. The STT relies on such trusted paths to the keyboard and display, and could benefit from many of the techniques they propose.

Finally, the cloud rendering engine can be seen as an extreme form of Software as a Service (SaaS). By running almost all of the application logic centrally—everything except the I/O path—the CRE offers two advantages to application providers: it makes it easier to

update the application (without requiring users to download a binary client in order to benefit from new features or security fixes to most of the code), and it allows for a simpler, more secure, and remotely verifiable client.

8 Conclusion

We presented Cloud Terminal, a new architecture for secure applications built around two primitives: a small *secure thin terminal* (STT) on the client and a *cloud rendering engine* (CRE) that contains almost all the application logic. The STT can be installed under a running operating system on standard PC hardware, even on compromised machines, and can be verified remotely. It achieves a sweet-spot between security, trusted code size, and generality by implementing a remote display protocol that can render arbitrary applications. The CRE runs applications in a provider-managed cloud and can scale to hundreds of sessions per machine. We have shown that Cloud Terminal is implementable on standard hardware and can provide secure access to a variety of applications at a low cost of 1.2 to 2.5 cents per user-hour.

Acknowledgments

Our shepherd Jon Howell suggested a change to the verification protocol to reduce assumptions about the phone system. The work described here has been supported by the NSF under awards CCF-0424422, 0842695, 0831501, and CNS-0831535, the AFOSR under MURI awards FA9550-08-1-0352 and FA9550-09-1-0539, Intel through the ISTC for Secure Computing, a Google PhD fellowship, and the NSERC (Canada). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

References

[1] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44:3–18, Dec. 2010.

[2] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *Linux Symposium*, pages 19–28, July 2009.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[4] K. Borders and A. Prakash. Securing network input via a trusted input proxy. In *HotSec*, 2007.

[5] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security Symposium*, Aug. 2009.

[6] Compare dedicated servers from CariNet. `https://www.cari.net/compare-all-servers`.

[7] Google Chrome OS. `www.chromium.org/chromium-os`.

[8] R. Cox, J. Hansen, S. Gribble, and H. Levy. A safety-oriented platform for web applications. In *IEEE Security & Privacy*, 2006.

[9] D. Dai Zovi. Hardware Virtualization Based Rootkits. In *Black Hat USA*, 2006.

[10] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol — version 1.2. IETF RFC 5246, Aug. 2008.

[11] Full-length version of present paper. `http://bitblaze.cs.berkeley.edu/cloudterm.html`.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[14] Intel Corporation. Intel Virtualization Tech. for Directed I/O.

[15] Intel Corporation. Intel Trusted Execution Technology (TXT).

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Linux Symposium*, 2007.

[17] K. Kortchinsky. Cloudburst: Hacking 3D (and breaking out of VMware). In *Black Hat USA*, 2009.

[18] B. Lampson. Accountability and Freedom. (Presentation), 2005.

[19] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[20] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM EuroSys*, Apr. 2008.

[21] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *NDSS*, 2009.

[22] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Tech. Journal*, 10(3):167–177, Aug. 2006.

[23] T. Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. In *CanSecWest*, 2007.

[24] B. Parno. Bootstrapping trust in a "trusted" platform. In *HotSec*, 2008.

[25] PolarSSL: Small cryptographic library. `www.polarssl.org`.

[26] T. Richardson. The RFB Protocol (ver. 3.8), Nov. 2010. `www.realvnc.com/docs/rfbproto.pdf`.

[27] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[28] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. Black Hat USA, 2006.

[29] J. Rutkowska and R. Wojtczuk. QubesOS. `www.qubes-os.org`.

[30] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *IEEE Sec. & Privacy*, 2007.

[31] S. Siddha, V. Pallipadi, and A. Van De Ven. Getting maximum mileage out of tickless. In *2007 Linux Symposium*, 2007.

[32] SiteKey – Bank of America. `www.bankofamerica.com/privacy/index.cfm?template=sitekey`.

[33] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, pages 337–352, Washington, D.C., USA, Aug. 2001.

[34] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[35] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.

[36] Trusted Boot. `http://tboot.sf.net/`.

[37] VMware Inc. Virtual Desktop Infrastructure.

[38] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *SOSP*, 2005.

[39] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, Dec. 2002.

[40] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36:195–209, December 2002.

[41] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat USA*, 2008.

[42] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy*, 2012.