Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor

Himanshu Raj, David Robinson, Talha Bin Tariq, Paul England, Stefan Saroiu, Alec Wolman
Microsoft Research

Abstract

This paper presents the Credo architecture to enable trustworthy virtualization based cloud computing platforms. A key feature of Credo is a small platform *Trusted Computing Base (TCB) for a customer VM that consists only of a securely launched hypervisor and minimal hardware components, without any* privileged partitions and their administrators*. Credo achieves this reduction in TCB via* emancipation*, a mechanism that provides VMs enhanced secrecy and integrity protection guarantees from privileged partitions. Trust in an emancipated VM is established via its* measured launch *by the hypervisor and an* attestation *of a dynamically established trust chain rooted in the Trusted Platform Module (TPM). Experimental results from a prototype implementation based on Hyper-V demonstrate that Credo provides enhanced security guarantees to emancipated VMs at a modest cost, most of which is a one-time startup cost from a VM's perspective, while adding only a small amount of code to a VM's TCB.*

1 Introduction

A key issue related to security properties of a virtual machine in virtualized environments is a large TCB due to components *external* to the VM, including hardware, firmware, hypervisor, and one or more privileged partitions, e.g., the root partition [25] and dom0 [3]. These privileged partitions usually run a large, full featured OS, and are managed by administrators who can arbitrarily extend these partitions with privileged code. Since these partitions are part of a VM's TCB by design, privileged code in these partitions can maliciously observe and modify a guest VM's virtual resources, such as memory and virtual device state, in an untrusted manner. This makes it very hard to provide meaningful security and trust guarantees to VMs when hosted in virtualization based third-party cloud computing environments that are not owned/controlled by VM's owners (also referred to as clients or customers). Such large and dynamic nature of external components in a VM's TCB makes it hard to trust a VM's state even if a hardware root of trust such as TPM is used. The recorded trust chain may be either too small to be meaningful, e.g., limited to just the BIOS and the bootloader [23], or arbitrarily long [32] to be of practical use. Also, a VM in cloud must trust in the security management policies of the infrastructure provider [13], since the physical machines belong to a different administrative domain. Trust in these policies is not easy to quantify, since these are hard to write and verify, and mostly kept opaque from the clients. We argue that such lack of trustworthiness is a key challenge facing wider adoption of cloud computing and is the chief contributor to security being the primary concern [11].

The Credo architecture presented in this paper addresses these issues in two ways - 1) by reducing a VM's *external* TCB to the hypervisor and minimal platform hardware/-firmware using *isolation and cryptography* based protection mechanisms, and 2) by providing a small and measurable hardware rooted trust chain that a client may trust, which includes trusted launch of the hypervisor using Intel's Trusted eXecution Technology (TXT) [16]. We demonstrate how a client of the infrastructure can remotely attest this trust chain while a VM is running in the cloud infrastructure, and provision resources to a VM, such as cryptographic keys, that can only be used in the presence of a valid trust chain.

This paper makes several novel contributions. First, it presents mechanisms to provide *enhanced security guarantees to general purpose virtual machines*. Other approaches that use hypervisor as the TCB fall short in that they either a) don't provide security guarantees at the right level of abstraction and require application level re-programming [21], or b) have large TCB that includes privileged VMs and their administrators [27, 9]. We argue that Credo's isolation and cryptography based approach is less costly and more practical compared to these other approaches. Second, Credo enables "on demand" security for guest VMs, with a trust model that is more representative of today's third party virtualization based cloud environments where clients has no control over the infrastructure. This trust model differs significantly from that of virtualized platforms [2], where a VM's TCB includes privileged VMs and their administrators. Third, Credo implements practical use cases of trusted computing technologies to provide a platform suitable for hosting security sensitive services inside a development and deployment friendly VM model. Examples of such services include a financial computation service employing proprietary code and secret data in a cloud and an out-of-partition OS health agent for VMs. We build a prototype of one such service termed Isolated Crypto Service (ICS) that manages cryptographic keys inside a VM and is used to perform cryptographic operations such as message signing and generation of session keys.

Experimental results from a prototype implementation based on Hyper-V [25] demonstrate that Credo enables a trustworthy virtualized platform at a low cost. In particular, there is negligible runtime cost imposed by Credo on computational workloads inside a VM. Cryptographic mechanisms for protection of storage impose overheads of $\sim 33\%$, and are comparable to existing encryption based data protection techniques [4]. Most of the overhead is a fixed setup cost - applicable only at the platform boot time and at the VM startup time. Our prototype adds only $\sim 11K$ lines of code to core hypervisor, and $\sim 4K$ lines of code to VM's runtime to provide enhanced security guarantees.

2 Credo Architecture

2.1 Threat Model

We motivate the need for Credo via security related scenarios pertinent to today's virtualized platforms. Since Credo is based on Hyper-V [25], we use Hyper-V architecture as the background for discussion, although these scenarios are generalizable to other *type-1* [12] virtualization systems such as Xen [3].

Key components of the Hyper-V architecture are a microkernel hypervisor and a privileged management partition, called the *root partition*. The hypervisor virtualizes core platform resources (e.g. CPU and memory) while the root partition owns all I/O devices and does I/O virtualization. Hyper-V supports two forms of virtualized I/O – *emulation based I/O*, where a guest VM performs memory mapped or port-based I/O that is intercepted by the hypervisor and forwarded to the the root partition; and *enlightened I/O*, where the root partition and a guest VM communicate over a shared-memory channel using the *vmbus* protocol. The root partition provides memory pages to back a guest VM's address space. These pages, as well as a guest VM's CPU state, are accessible to the root partition to aid in the VM's execution.

In Credo, we address two types of threats: originating from the hypervisor and originating from the root partition. First, a malicious administrator can boot the system into a rogue version of the hypervisor [17], or can modify the hypervisor binary before it is started. Second, a compromise of the root partition may use the interface between the root partition and a guest VM to attack a guest VM. In particular, a kernel level exploit of the root partition can use the implicit trust in the root partition by the hypervisor to read and manipulate a guest VM's memory, virtual register state, and virtual I/O state. Alternatively, a user-mode component on the root partition may perform a privilege escalation attack to gain administrative rights, and then either use a kernel level malware as described above, or map guest VM memory pages in its own address space to read/modify guest data. This threat model of misusing the trust between the hypervisor and a privileged partition is akin to a *malicious motherboard*, that can snoop the buses and modify memory and CPU state willy nilly. A special case of "malicious motherboard" model is a *rogue device*, where the malware compromises the virtualization stack in the root partition that provides virtual device and firmware like functionality (e.g., virtual BIOS) to a guest VM. These "rogue" virtual devices can then compromise online and offline VM I/O state, and online VM memory state via illegal reads and writes to guest VM memory.

Compromise of the root partition may be the result of an exploit through the large surface area of the OS kernel and system services, or a deliberate act from an adversarial administrator. The administrator may be genuinely malintentioned, or "honest but curious", or simply ignorant of threats posed by malware.

The Credo threat model does not consider certain types of attacks. First, we assume that a VM (privileged or otherwise) cannot compromise the hypervisor via the hypercall interface, and the hypervisor itself doesn't have any security bugs. There are steps towards formally verifying an OS kernel that mat be applied to hypervisors [18, 19], although there is no formal proof of the correctness for any commercial hypervisors yet. Second, we do not consider denial of service (DoS) attack by not allowing a guest VM to run, any side-channel attacks [31], or certain hardware attacks. This includes Cold boot attacks [14] on system RAM to read secret data from it; DMA attacks using a bad peripheral with the pre-boot environment to compromise the hypervisor or a privileged partition [5]; and DMA attacks on guest VM memory through physical devices that a privileged partition may control. Most of the DMA based attacks can be defended against using a capable IOMMU, as demonstrated by TrustVisor [21] and NOVA [34].

2.2 Design Goals

1. **Strong Isolation from Privileged VM(s)** . In particular, provide secrecy and/or integrity protection of a VM's virtualized state from the root partition. From a cloud customer's point of view, this removes the need for trust in the software environment and administrators of the root partition on provider's infrastructure.

2. **Trust, but Verify**. Establish a small, measurable TCB for a VM as well as a measure of VM's trustworthiness, and enable mechanisms to verify this at runtime. This requires specific hardware support, such as trusted launch capability, TPM, and IOMMU, components that are becoming commonplace on today's server class systems. We believe that any additional cost, if so needed, is worth the investment for the cloud infrastructure provider, given the benefits it provides to the clients.

3. **Enable "on demand" security model**. Cost of security should be imposed only if security is required by a VM. Since performance is the key requirement in any cloud computing environment, any such cost should be low.

To meet these goals, the Credo hypervisor provides a secure execution environment for a guest VM using a set of

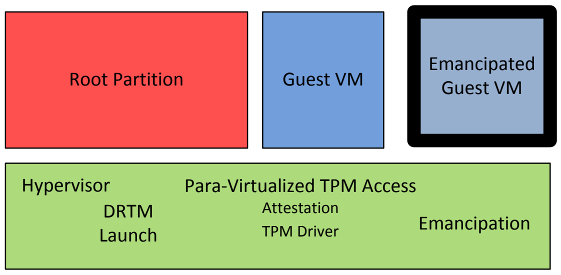


Figure 1. High Level View of Credo Architecture. Components inside the hypervisor highlight the additions made by Credo to Hyper-V hypervisor. Bold lines around *emancipated* guest VM denotes the secure execution environment.

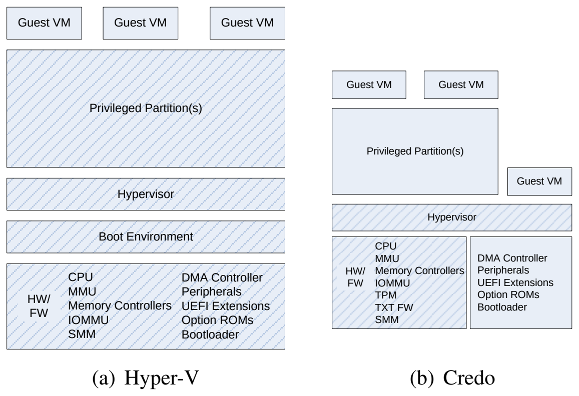


Figure 2. TCB comparison in Hyper-V and Credo. Shaded components denote the platform TCB.

mechanisms collectively referred to as *emancipation*. Emancipation enables the VM to execute without taking any security dependency on the root partition, thereby effectively removing the root partition from the guest VM's TCB, in particular its OS kernel, device drivers, and administrators. Further, Credo reduces the overall *platform* TCB to a small hypervisor and minimal hardware/firmware components by using *trusted launch* of the hypervisor based on Intel's TXT technology [16]. Trusted launch of the hypervisor establishes a *Dynamic Root of Trust for Measurement* (DRTM). Hereafter, we use the term DRTM launch and TXT based trusted launch interchangeably.

Using measured launch of a VM with a trust chain recorded in the TPM, Credo enables TPM-based local and remote attestation for an emancipated VM. The client who owns this VM may use this attestation to verify the trustworthiness of the VM and the physical platform. These components are shown in Figure 1, and are described in detail in following sections.

Together, these aspects of Credo enable an alternative trust model for guest VMs running in the secure execution environment, where the external TCB of a guest VM is restricted to the platform TCB only. This TCB is denoted in Figure 2(b), which is much smaller as compared to Hyper-V's TCB denoted in Figure 2(a).

3 Emancipation

The Credo architecture provides a way to execute guest virtual machines in a secure and trustworthy environment without taking a trust dependency on the root partition using a mechanism similar to DRTM launch. The hypervisor provides a hypercall to trigger a *v(irtual)DRTM* event for a guest VM. When this event is triggered (either by the guest VM or by the the root partition on its behalf), the hypervisor suspends the VM, creates the secure execution environment for the VM using the *emancipation* procedure, *measures* and *records* the "execution state" of the VM. As the last step, the hypervisor resumes the execution of the guest VM inside the secure execution environment.

The "execution image" of the guest VM is produced on a separate *trusted system*, which is *necessarily outside the control of the infrastructure where the VM runs later under Credo*. This trusted system is owned and managed by the owner of the guest VM, who is responsible for maintaining its security and trustworthiness via other mechanisms outside of scope for this paper. We defer the details of the "execution image" – how it is created, and how it is used - to Section 5.

Emancipation collectively refers to mechanisms employed by the hypervisor and the guest VM to safeguard the VM from a potentially malicious privileged partition. Emancipation affords the guest VM a secure execution environment where *it can protect secrecy and integrity* of its information from the components that otherwise form its external TCB on commodity virtualization environments. Notice that ultimately it is a guest VM's responsibility to make use of this execution environment to enforce these security guarantees for its information. Stated other way, a guest VM can always choose to divulge its secrets to other VMs using I/O channels, or as in the case with Credo, can disable emancipation protection entirely.

We distinguish between emancipation mechanisms that protect a guest VM's memory and virtual CPU state versus those protecting its I/O. The former set of mechanisms are provided by the hypervisor, since those virtual resources are directly controlled by the hypervisor. In most commodity virtualization platforms, I/O virtualization is the responsibility of the root partition or a driver VM. The guest VM directly controls the mechanisms for emancipating I/O with assistance from the hypervisor.

3.1 Emancipating memory and virtual CPU state

To emancipate a guest VM's memory, the hypervisor removes root partition's access from its page tables for all system memory pages backing a guest VM's physical address space. Both reads and writes to these pages are intercepted by the hypervisor - reads return all $0xFF$s, while writes are silently thrown away. After the VM is emancipated, the hypervisor disallows creation of any new mappings in an emancipated VMs physical space. This implies that that a guest VM's address space must be completely populated before the VM is emancipated. However, this does not preclude dynamic memory management (e.g. via a balloon driver [36]) where the VM can unemancipate memory pages before returning them to the root partition.

Because an emancipated VM has exclusive control over memory, it must explicitly release control of the memory pages backing its physical space in order for the resources to be reclaimed by the root partition after the VM shuts down. This step is accomplished using the "unemancipate partition" hypercall, which resets the root partition's access to its original state for all memory pages backing the guest VM. It is imperative that the guest VM must explicitly remove any secret information from pages explicitly before calling the unemancipate partition hypercall in order to maintain the secrecy and integrity guarantees.

A guest VM's vCPU state may be modified outside the control of the guest VM as a result of *intercepts*. These intercepts are either caused by guest VM itself, e.g., by accessing some virtual resource such as MSR or I/O port, or by external events, such as a virtual interrupt associated with a virtual device.

Most intercepts related to a virtual resource maintained by the hypervisor, e.g., a Model Specific Register (MSR), are handled by the hypervisor, thus any guest VM's vCPU state change in that path is trusted. Of concern are intercepts that are not handled by hypervisor and are forwarded to the root partition. For example, the OS in guest VM may try to access virtual devices that cause an intercept to be generated, which the hypervisor forwards to the root partition. In order to service this intercept, the root partition needs to access the guest VM's vCPU state and its memory. Guest VM's memory is safeguarded using page tables as described earlier. However, the vCPU state is still vulnerable to attack by a malicious the root partition.

Our approach to vCPU state emancipation is two pronged: limit the number of intercepts that are forwarded to the root partition, and enhance access controls on virtual registers that intercepts may need to access.

The former approach implies that a guest VM must limit its use of intercept or emulation based I/O. We enforce this in the hypervisor – the root partition is not allowed to add any new memory areas in an emancipated guest VM's GPA which could cause an intercept to occur. Only a limited number of I/O ports are allowed to cause intercepts, currently only those belonging to the timer, keyboard, and serial I/O. On the intercept path for these ports, the hypervisor enforces that the root partition only accesses the appropriate vCPU registers. For example, for serial I/O the root partition needs to access only the EAX register and access to any other register is denied. Note that this kind of specialized handling in the hypervisor can be enabled only for a handful of simple, well known emulated devices.

We do not find these restrictions on emulated device usage burdensome for a guest VM. Most commodity virtualized platforms already discourage the use of intercept and emulation based I/O due to high performance overheads and provide enlightened or para-virtualized I/O devices. Paravirtualization uses shared memory channels to perform I/O, a much faster mechanism than the slow hypervisor driven intercept path. Further, emancipating I/O for legacy emulated devices requires changes to their proprietary device drivers, which is not possible for third party closed-source drivers.

3.2 Emancipating I/O

In Credo, it is the responsibility of the guest VM to use *cryptographic* measures for I/O emancipation as the hypervisor is not involved in the para-virtualized I/O path. Making the guest VM aware of I/O emancipation is compatible with the para-virtualized I/O model.

Emancipated para-virtualized I/O from a guest VM involves two steps: first, a shared memory based channel is established between an emancipated guest VM and the untrusted root partition; and second, a guest VM uses secrecy and/or integrity protection techniques to read or write data to or from this shared memory channel.

For the root partition to use shared memory for communication it needs to be able to access guest memory which is by default protected by memory emancipation. We provide the "unemancipate page" hypercall to selectively remove protection for the pages used by the shared memory channel. One such channel is created for each para-virtualized device. Messages sent over the channel may contain pointers to buffers on data pages that must also be unemancipated. As an optimization, instead of calling the "unemancipate page" hypercall for every page, the vmbus keeps a pool of unemancipated pages that are setup at VM startup. Drivers allocate and free memory pages to and from this pool. The vmbus driver in the guest VM can grow/shrink this pool on demand as needed.

This encryption based approach to emancipating I/O works in a cloud environment since the guest VM mostly requires just storage and network based I/O to execute in such an environment. In fact, Credo explicitly disallows any emulation based I/O, and all VM management should be performed using a network based remote access connection.

As a concrete example, we have implemented a Secure Disk (SDisk) driver by modifying the enlightened, SCSI-based, virtual hard disk (VHD) driver. SDisk encrypts *every block* of data stored on the VHD. The encryption key is stored in a protected manner that prevents malware in the root partition from accessing it. Exact details on how this is accomplished is deferred to Section 5. A similar encryption based approach for protection can be applied to networking via an emancipation aware network card driver, by using an IPsec channel, transport layer security such as SSL/TLS [6], or application level encryption of data. To facilitate communication with the root VM (which also forwards communication from the outside world to the guest VM), we currently provide an emancipation aware vmbus based communication channel called vmbus pipe. This channel is used, e.g., to exchange the remote attestation that a VM obtains from the Credo hypervisor, and by services such as the shutdown indicator service and the time synchronization service provided to a guest VM by Hyper-V.

4 Building Trust in Credo

4.1 Trusted Computing Overview

We briefly recapitulate some of the trusted computing technologies we leverage in this work. A more detailed description is provided by [28]. Trusted computing technologies include hardware protection of cryptographic keys, non-repuditable platform state recording and reporting, and strong machine identityA key hardware component that provides much of these functionalities is a Trusted Platform Module (TPM).

A TPM is similar to a smartcard tied to the motherboard of a platform. Each TPM is equipped with a unique Endorsement Key (EK), which provides a strong platform identity and a Storage Root Key (SRK) is the root key for the TPM key hierarchy. The EK is rarely ever used – rather an AIK is used as an alias for the EK due to security and privacy concerns. A TPM has multiple 20-byte sized Platform Configuration Registers (PCRs) that can be used to record the measurements of various software components on the platform. Most of these PCRs are non-resettable by software, and can only be *extended* providing the basis for maintaining platform integrity.

The recording of platform integrity is used to create a chain of trust starting from the CPU itself. This chain of trust, or Root of Trust for Measurement (RTM), is established either statically (SRTM), or dynamically (DRTM). In SRTM the trust chain starts in the processor at the system reboot, and is built by previous software component measuring and extending the next component into PCRs. One example use of SRTM is the Microsoft BitLocker full volume disk encryption technology [23]. One key weakness of SRTM process is that the trust chain is *brittle* in the face of hardware and software environments where changes to platform configuration and initial software execution frequently occur.

To address the fragility of the SRTM process, TCG is developing a new standard called the Dynamic Root of Trust for Measurement (DRTM), provided by Intel's TXT [16] and AMD's SVM [1] technologies. The fundamental technique is to allow untrusted code to establish the state of a platform, reset the platform with the DRTM event using a special CPU instruction (*GETSEC[SENTER]* on TXT) into a trusted environment. This removes a host of pre-boot software, such as bootloaders, and firmware from the platform TCB. DRTM launch starts a trusted environment that measures and records a Dynamically Launched Measured Environment (DLME). Measurements of DLME in a specific set of PCRs also form the evidence of the DRTM event. The DLME has a very small TCB (minimal hardware and TXT firmware) and runs in a secure environment that provides memory isolation from DMA, among other protections. The DLME is responsible for TCB management for the rest of the software environment.

4.2 Establishing Minimal Platform TCB: DRTM Launch of the Hypervisor

The platform TCB for commodity virtualization platforms is quite large. Besides dynamic privileged partitions, it includes a number of hardware and firmware components, such as option ROMs, and the pre-boot environment, which itself can be extensible. Credo uses TXT technology to establish a minimal platform TCB, composed of the hypervisor and a small number of other hardware and firmware components, including the TPM and the TXT firmware.

DRTM launch process of the Credo hypervisor involves two phases:

- In the first phase potentially untrusted code configures the platform, loads a small hypervisor specific bootloader called HvDLME, and the hypervisor into memory, and asserts the DRTM event.
- In response to the DRTM event the platform measures and launches HvDLME, which in turn measures and launches the hypervisor. The hypervisor takes control of the resources necessary to protect itself (e.g., MMU) and the TPM, and establishes itself as part of the platform TCB.

At each step the measurements are recorded in a subset of TPM's resettable PCRs (17-23). Access to these PCRs is controlled via the notion of a *locality*. The platform uses locality 4 *not available to the software* to record the measurement of HvDLME in PCRs 17 and 18 in response to the DRTM event. HvDLME uses locality 2 and PCR 22 to record the measurement of the hypervisor. As described later in Section 4.3, the hypervisor only exports locality 0 through a para-virtualized interface *which cannot reset any of these PCRs*. Hence, measurements reflected in PCRs 17, 18, and 22 after the hypervisor startup reflect the trustworthiness of the platform TCB. Finally the hypervisor starts the privileged root partition which sets up the rest of the machine to make it available for hosting virtual machines.

4.3 Para-virtualized TPM Access

In Credo architecture, the hypervisor owns and controls the TPM, as TPM records the trust chain and forms the basis for trustworthiness of the platform TCB. We leverage prior work by [7] to enable para-virtualized TPM access for guest VMs. The hypervisor implements a software implementation of PCR registers, called virtual PCR registers or vPCRs, for each VM. One such vPCR, called *details* vPCR, is initialized by the hypervisor at the emancipated start of a VM using the measurement of VM's execution state. This vPCR forms the *code identity* for the VM, and is reset only when the VM leaves the secure execution environment. Others hardware resources, such as TPM contexts and keys, are shared among VMs. Guest VMs use the hypercall interface to communicate with the TPM, which is very similar to the physical TPM interface. The hypercall includes additional header information for vPCR specific commands (e.g., extend and reset) and a virtual TPM context. The virtual TPM context includes vPCR information if any vPCR are needed by the command. Hypervisor, then, loads a representation of these vPCRs and their contents into the hardware PCR 23. Guest VMs are free to select any combination of physical and virtual PCRs to suit their needs. For example, to authenticate DRTM launch of the hypervisor, a guest VM may select physical PCRs 17, 18, and 22 for Credo in the TPM_Quote command. If the guest VM wishes to authenticate itself and the hypervisor, it selects the details vPCR as part of the virtual TPM context, and then calls TPM_Quote via hypercall with PCRs 17, 18, 22, and 23.

5 Emancipated VM Lifecycle

5.1 Preparing a VM for Emancipated Execution

The secure execution environment provided by the emancipation process poses certain restrictions on the guest VM. In particular, it requires that: 1. The guest VM starts execution from a memory and vCPU state that *does not* have any security dependency on the root partition; 2. In order to be used as the *code identity*, the state must be repeatable, with a measurement that is unique; and 3. The execution state must be self-sufficient to start the enlightened I/O.

These requirements are a significant departure from the traditional model of VM startup. In current virtualization environments, a VM's startup is prepared by the virtual BIOS, *untrusted in the Credo model*, and relies heavily on emulated, non-enlightened I/O from the root partition. Additionally, in order for the measurement based approach to work, it is imperative that the vDRTM event is asserted at a point in time where not only the execution image meets the criterion 1 and 3, but also it is predictably same for repeated execution. Assuring that execution state of the VM at the point when vDRTM is asserted remains fixed presents a practical challenge, since OS boot process usually performs many asynchronous startup steps. Said differently, initiating the secure execution environment via vDRTM event with traditional model of VM boot may not result in a repeatable measurement.

One possible approach for building an execution state with required properties is to enlighten the boot process of the OS itself. In particular, the bootstrap code of the kernel needs to be modified. This kernel is loaded in VM's address space via a separate bootloader inside the root partition. This approach is possible for an open source OS such as Linux [3]. However, this approach has a high engineering cost as it requires major modifications to the OS kernel.

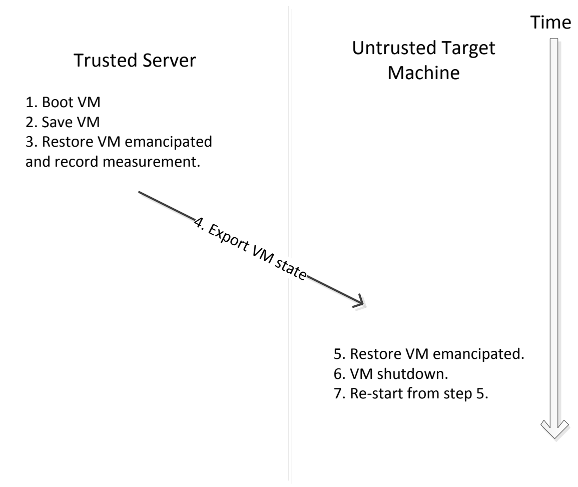


Figure 3. Running VMs emancipated in Credo

We overcome this issue of repeatability via VM save/restore functionality. Based on the invariant that the execution state of a VM is the same each time it is started from the same saved state, Credo creates an "execution image" to be later executed inside the secure execution environment using steps outlined in Figure 3. This process involves a *trusted system* running a Hyper-V environment where the root partition *is trusted*. As described earlier, this trusted system belongs to the client – VM's owner – and is outside the control of the cloud infrastructure, where the VM executes later. In step 1, the *trusted server* boots up the required VM to a *quiesced* state where the OS does not need emulation based I/O any more. The para-virtualized emancipated I/O drivers, including vmbus are loaded in memory, but aren't started yet. This state meets criterion 1 and 3 for vDRTM event assertion outlined earlier. In step 2, a VM save operation by the root partition at the trusted server produces a snapshot of VM's memory contents and vCPU register state. The root partition discards any virtual device state (except for the state that Credo doesn't emancipate, such as contents of video RAM). This saved state needs to be persisted for later emancipated runs of the VM. The trusted server produces a measurement of the saved state. Currently, this is accomplished via a emancipated restore of the VM on the trusted server, which results in the hypervisor populating the details vPCR with the desired measurement. In step 4, we use VM export/import functionality in Hyper-V to transfer the saved VM state to the target machine. Once the saved VM state arrives at the target machine, it may use this state to restore the VM emancipated as many times for desired duration of activity (between steps 5 and 6) as needed.

5.2 Provisioning Emancipated VMs

Emancipation provides the necessary protection guarantees to a guest VM from the privileged root partition. However some key questions still remain, e.g., how does a VM know (or convince an external machine on the network) that it is running emancipated? Since the environment is virtualized, a VM cannot trust any response it receives from the root partition or the hypervisor, as these components may be compromised. Additionally, a long-lived customer VM would need to maintain some persistent storage that survives across VM restarts. A VM may use encryption to preserve the confidentiality of data on this persistent storage. But how deos the VM obtain this key and safeguard it from a malicious root partition?

We leverage TPM-based attestation to solve these problems. In order to establish trustworthiness, a VM either performs a *remote attestation* using *TPM Quote* as part of communication setup with the outside world, and/or *local attestation* using *TPM Seal/Unseal* to safeguard the encryption key for offline storage [35]. Protection of a key when it is VM's memory during VM's execution is provided by emancipation, as described earlier. Next we describe two scenarios that detail the use of TPM-base attestation to establish the required trustworthiness in the VM.

5.2.1 Remote Attestation of an Emancipated VM

Assume that a machine owned by the client $\mathtt{M}_{\mathtt{client}}$ wants to assess the trustworthiness of a VM, $\mathtt{VM}_{\mathtt{client}}$, running on machine $\mathtt{M}_{\mathtt{target}}$. Additionally, $\mathtt{M}_{\mathtt{client}}$ wants to provide $\mathtt{VM}_{\mathtt{client}}$ a secret S *only if* $\mathtt{VM}_{\mathtt{client}}$ *is trustworthy*. We assume the presence of an *Attestation Identity Key* (AIK) available for use by the TPM on $\mathtt{M}_{\mathtt{target}}$ which is trusted by $\mathtt{M}_{\mathtt{client}}$. The process of setting up an AIK is described elsewhere[30].

In this scenario, $\mathtt{VM}_{\mathtt{client}}$ creates a new random RSA keypair $\mathtt{K}$ during initialization, and creates a vPCR *keyhash* extended with the hash of $\mathtt{K}_{\mathtt{pub}}$. $\mathtt{M}_{\mathtt{client}}$ challenges $\mathtt{VM}_{\mathtt{client}}$ with a random nonce $\mathtt{N}$ (to guarantee freshness). In response, $\mathtt{VM}_{\mathtt{client}}$ uses the para-virtualized TPM interface to send a `TPM_Quote` command over N to associate the public key with the environment in which it is running by including the physical PCRs 17, 18, 22, and 23, and details and keyhash vPCRs. The hypervisor resets the PCR 23 and extends it with desired vPCRs, and then forwards the `TPM_Quote` request to the TPM. The TPM prepares a quote blob including a signature over $\mathtt{N}$ using $\mathtt{AIK}_{\mathtt{priv}}$ and the desired PCRs. This quote is then sent back to $\mathtt{M}_{\mathtt{client}}$ along with $\mathtt{K}_{\mathtt{pub}}$. $\mathtt{M}_{\mathtt{client}}$ examines the quote blob, and if it meets policy, encrypts the secret $\mathtt{S}$ with $\mathtt{K}_{\mathtt{pub}}$, and sends it to $\mathtt{VM}_{\mathtt{client}}$. In particular, $\mathtt{M}_{\mathtt{client}}$ ensures that:

- $\mathtt{M}_{\mathtt{target}}$ has a valid physical TPM (association between the AIK and a specific TPM).
- $\mathtt{M}_{\mathtt{target}}$ is running a good known version of the hypervisor launched with DRTM (captured in PCRs 17, 18, and 22),
- $\mathtt{VM}_{\mathtt{client}}$ is running emancipated (captured in details vPCR),
- $\mathtt{VM}_{\mathtt{client}}$ is indeed the VM that it intends to attest (captured in details vPCR)

$\mathtt{VM}_{\mathtt{client}}$ can then use $\mathtt{K}_{\mathtt{priv}}$ to recover the secret $\mathtt{S}$.

5.2.2 Managing the SDisk Encryption Key

If there is an online connection between an emancipated VM and the trusted server *outside the infrastructure where VM is running* where the secure disk for this VM is initialized and encrypted with, say, a symmetric key $\mathtt{K}_{\mathtt{SD}}$, the quote based mechanism described in previous section can be used to provide the VM with this key. The provisioning protocol we have implemented works *without an online connection* between the emancipated guest VM, and the trusted server. Steps for this protocol are described next.

1. The trusted server creates a random symmetric key $\mathtt{K}$, and "remote-seals" it to the target machine with the TPM containing the AIK above, running a known good version of DRTM launched hypervisor, and running the intended guest VM. This results in an *activation* blob. The remote sealing process is described later.

2. The trusted server encrypts $\mathtt{K}_{\mathtt{SD}}$ with $\mathtt{K}$. Encrypted $\mathtt{K}_{\mathtt{SD}}$ and the activation blob are sent to the root partition on the target machine.

3. The root partition stores encrypted $\mathtt{K}_{\mathtt{SD}}$ and the activation blob as part of SDisk's metadata. It also changes SDisk state to "activate". It then starts the emancipated guest VM and attaches SDisk to it.

4. Emancipated guest VM calls `TPM_ActivateIdentity` command with the activation blob using para-virtualized TPM interface. The TPM reveals $\mathtt{K}$ to the guest if all the PCR measurements establishing the trustworthiness of platform and guest VM are valid, and AIK is loaded and is a valid identity key.

5. Emancipated guest VM decrypts $\mathtt{K}_{\mathtt{SD}}$ with $\mathtt{K}$. Next it mounts SDisk using $\mathtt{K}_{\mathtt{SD}}$.

6. Emancipated guest VM *Seals* $\mathtt{K}_{\mathtt{SD}}$ using `TPM_Seal` command [35] to the platform configuration, its details vPCR, and TPM's Storage Root Key (SRK). It stores the *sealed* blob on the SDisk as metadata, and changes the SDisk state to "initialized".

7. From this point onwards, the emancipated guest VM can use `TPM_Unseal` command [35] to obtain $\mathtt{K}_{\mathtt{SD}}$ on further restarts.

Remote Sealing

We perform remote sealing of provisioning data using the Endorsement Key (EK) of target machine's TPM as an

Component	SLOC
DRTM Launch of Hv	3183
vDRTM Launch of VM	875
ParaVirt. TPM Access	8505
vmbus enhancements	323
SDisk	1890
TPM driver	961
Total	15737

Table 1. SLOC metric for Credo modifications to Hyper-V

asymmetric encryption key. The trusted server prepares a TPM_EK_BLOB_ACTIVATE [35] data structure. This structure contains a TPM_PCR_INFO_SHORT describing the expected and demanded PCR configuration on the target machine. The data structure is then encrypted with the EK_{pub} of the target machine's TPM. This approach is similar to certifying AIK as described in [30].

We establish a simple server-side database of public EKs out-of-band. We expect that a VM's owner would obtain EK certificates for set of machines the cloud infrastructure would use to run her VM(s).

6 Implementation Details

Credo infrastructure requires modification to the hypervisor and the vmbus interface in order to provide the secure execution environment to a guest VM. Table 1 lists the specific components and source lines of code (SLOC) required for these modifications. In all, Credo adds $\sim 15K$ SLOC to a VM's TCB — $\sim 12K$ of which are added to the hypervisor and $\sim 3K$ are added to the OS runtime for I/O emancipation and para-virtualized TPM access. Next, we provide implementation specific details of some of the components that were omitted from the architectural description.

Emancipated VM startup: The saved VM state that forms the "execution image", captured via VM save operation in step 2 of Figure 3, cannot be using any non-enlightened, emulation based I/O.

In order to build such an image on the trusted server using traditional Windows boot process, we use a bootable RAMDISK stored on a virtual IDE disk. Here, the IDE disk is only used by the bootloader on the trusted server. By disabling the IDE disk driver (disk.sys) from this RAMDISK installation, we ensure that once Windows finishes booting, the OS does not use the IDE disk with which the VM started executing. We save the memory image and CPU state using Hyper-V's save mechanisms after Windows finishes booting. A similar approach can be taken for Linux as well using *initrd* image.

Although limited by the size of RAM allocated to the VM, we find this method adequate for running completely functional versions of Windows, e.g., Windows Preinstall Environment [26]. This minimal environment provides enough functionality for building special purpose VMs.

Once the saved image is started emancipated on an untrusted machine in the cloud infrastructure, a startup script loads the modified vmbus driver (which also starts the vmbus pipe communication channel), and the SDisk driver (both are part of the RAMDISK image). Any futher setup operations must be run from SDisk.

In Hyper-V, start from saved state by default deletes the saved state. This adds an extra copy step to save the image to another file to emancipated start of VM. This copy operation takes time proportional to the size of the saved VM image. In a future optimized implementation, this cost can be removed by changing the default Hyper-V action to not delete the saved state.

SDisk is implemented as a filter driver in the virtual storage stack that sits right above the virtual SCSI driver. It mainly interposes itself on the read/write path, and performs data encryption/decryption before passing it down to virtual Encryption mechanism used is AES with 256-bit key as K_{SD}. SCSI driver, which then sends emancipated I/O data on untrusted vmbus communication channel. SDisk driver stores metadata related to SDisk on the last 1MB of vhd. Block containing metadata are kept hidden from upper layers in the storage stack of the guest VM, thereby reducing the available storage capacity by 1 MB.

An optimization used by SDisk driver is to use unemancipated pages as target for encrypted data. Similarly, the data to decrypt is directly targeted to upper layer buffers, rather than first copying then decrypting. This ensures less overhead, if any, than doing encryption/decryption of data pages in place and then copying the content to/from unemancipated pages. As shown by experimental results for SDisk, the cost of copying (aka bouncing) buffer to/from unemancipated pages is minimal - most of the cost is attributed to actual encryption/decryption operations.

One limitation of virtual SCSI implementation in Hyper-V is that it can't be used for booting the VM. This restricts our SDisk implementation as well, which uses virtual SCSI driver underneath. Hence, the current SDisk prototype can only be used for data storage after VM is already running emancipated.

DRTM launch of hypervisor: TXT architecture puts certain restrictions on the DLME that make it difficult to directly launch the hypervisor as a DLME. Instead a small Hyper-V aware DLME (HvDLME) that understands the specifics of Hyper-V hypervisor is used as DLME. We extend Windows bootloader (winload.exe) to load HvDLME and the actual hypervisor binary in the memory, and perform a small amount of DRTM specific configuration to establish the required memory mappings, e.g., to allow TPM access. Next, the boot loader launches HvDLME using DRTM launch as described earlier. As a result of DRTM operation, PCRs 17 and 18 are set with measurements related to HvDLME.

The HvDLME uses the signatures and code of the hypervisor loaded by the boot loader to record the hypervisor's identity in PCR 22, and validates that it was loaded correctly. Lastly it transfers control to the hypervisor establishing a robust platform TCB with a trust chain based on DRTM launch.

7 Evaluation

7.1 Security Analysis

In this section, we revisit threat scenarios introduced in Section 2.1, and illustrate how Credo counters them via mechanisms introduced in this paper.

DRTM launch of the Credo hypervisor removes the possibility of an administrator starting the platform in an untrusted state without getting detected, e.g., by starting a rogue hypervisor. Similarly, any malicious modifications to the execution image before it is started to launch the emancipated VM would get recorded in details vPCR. Unseal and remote attestation mechanisms will protect emancipated guest VMs from leaking any secret data, and facilitate detection of the fact that the platform is untrusted, or the execution image was modified in an untrusted manner.

Secure execution environment established by Credo affords runtime immunity to emancipated VMs from malware in the root partition, which may potentially be installed by a malicious administrator. Further, cryptography based mechanisms enforce protection of sensitive data from the same at rest or during I/O.

7.2 Performance

Experiments are performed on a machine with dual core Core 2 CPU at 2.53 GHz, 2GB RAM, and a 7200 RPM ATA hard disk. This machine also has the TXT trusted launch capability and a TPM v1.2. In stock configuration, machine runs 64-bit Server 2008 R2 Enterprise version of Windows in the root partition and the Hyper-V hypervisor. The other configuration is running the Credo environment, which runs our version of the hypervisor, and a corresponding private build of 64-bit Server 2008 R2 Enterprise version of Windows in the root partition. The guest VM is configured with a single core and 1GB RAM, and runs a minimal Windows PE environment, that is also based on 64-bit Server 2008 R2 Enterprise version of Windows. When booted, $\sim 40\%$ of the memory is dedicated to RAMDISK, of which 32MB is free for temporary storage. The corresponding .wim file is $\sim 330MB$ in size on disk and contains the base Windows PE environment, utilities required to load drivers via command line, Credo specific drivers for vmbus, SDisk, and para-virtualized TPM, and PassMark Performance Test benchmarking suite v6.1 [29]. The vhd used by SDisk is 2GB in size.

Microbenchmarks DRTM launch of hypervisor adds no noticeable latency to machine boot up from power-on reset compared to the research prototype version of Hyper-V ($\sim$ 160 seconds till windows logon screen for both the cases). Emancipated VM startup takes $\sim$ 35 seconds longer than an unemancipated startup, most of which is spent in hashing the guest GPA and generating the details vPCR. The extra copy step required to start VM in emancipated state also adds $\sim$ 25 seconds. Extra $\sim$ 20 seconds are taken by the script that starts emancipated I/O drivers. The unseal operation by SDisk driver to obtain K_{SD} takes ~ 2.5 seconds, while the quote operation to generate remote attestation takes $\sim$ 1 second. Note that most of these are setup costs and are applicable only once per execution of an emancipated guest VM.

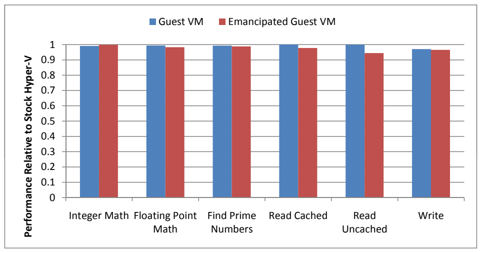
PassMark We use the PassMark Performance Test benchmark suite [29] for benchmarking the performance of various tests inside the guest VM. The benchmark is run in three scenarios:

- With stock Hyper-V configuration.
- With Credo but without the secure execution environment. This measures the impact of Credo on non-security sensitive VMs.
- With Credo within the secure execution environment. This measures the impact of Credo and secure execution environment on security sensitive VMs.

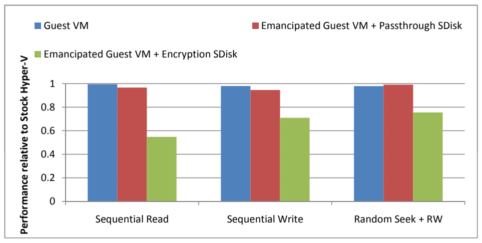
We report performance results for scenario 2 and 3 relative to scenario 1. These results are broadly categorized as CPU, memory, and I/O, based on the subsystem being stressed by the benchmark. All tests were executed for a 30 second duration, and we use the average of 20 runs to compute relative performance.

PassMark's CPU tests include integer math, floating point math, prime number finding, SSE instructions, compression, and string sorting. Memory tests include allocation, read cached, read uncached, write, and Large RAM. Results are depicted in Figure 4(a). For brevity, we omit results for some of these cases, but the trends are similar to ones that are shown. These results demonstrate that the Credo does not have appreciable impact on CPU and memory intensive workloads in a guest VM, whether using a secure execution environment or not. This is as expected, since Credo does not impose any runtime cost for these workloads. Any minor variation in performance compared to stock Hyper-V is due to the fact that our research prototype version is compiled without optimizations and with debugging features.

For disk based tests, we use the Hyper-V virtual SCSI driver for scenarios 1 and 2. For scenario 3, we sub-divide results in two categories: one, using a version of SDisk driver that does just data copying to unemancipated memory and doesn't perform any encryption or integrity protection (*passthrough disk*); and two, a version of SDisk driver that does encryption (*Encrypted disk*). This identifies the cost imposed by two steps of I/O emancipation - data copying and encryption, respectively. Disk tests include sequential read, sequential write, and random seek + read/write. These results are presented in Figure 4(b). These results show that impact of data copying in/out of unemancipated pages on overall I/O throughput is negligible. Encryption adds $\sim 33\%$ overhead on average, which is similar to other similar technologies [4].



(a) CPU and Memory (first 3 tests are CPU, last 3 tests are memory)



(b) Disk

Figure 4. PassMark benchmark results. Guest VM and Emancipated VM denote a VM running in Credo without and within secure execution environment, respectively.

In summary, experimental results show that Credo imposes mostly one time setup cost. Credo does not impact performance for virtual machines that do not require the security benefits when compared to a stock Hyper-V environment, while only imposing modest cost on emancipated VMs.

Isolated Crypto Service

This section provides a qualitative evaluation of Credo, by showing how Credo enables security sensitive functionality in a cloud environment. In particular, we present the implementation of *Isolated Cryptographic Service* (ICS). ICS acts as a headless virtual security appliance and offers a subset of the standard Microsoft Cryptography Next Generation library (CNG) [24], including secure key generation and asymmetric cryptography. By leveraging the secure execution environment provided by Credo, ICS provides strong protection of cryptographic keys in memory, isolated execution of cryptographic operations, and secure generation and storage of keys. These properties are of utmost importance for many services, such as Certificate Authorities, secure data storage, and security sensitive applications in finance and health care industries.

The operating environment running in the ICS is a minimal, enlightened version of Windows based on Windows Preinstall Environment (WinPE). Startup of ICS is performed using mechanisms described in Section 3. The confidentiality and integrity of the executing ICS instance and the confidentiality of stored data are ensured via emancipation and SDisk, respectively. ICS provides a vmbus based pipe communication channel to other VMs to communicate with the CNG service running inside ICS.

We used a micro-benchmark user application running in the root partition to sign a 16KB message using a 1024-bit RSA key using ICS. The latency for this operation is $\sim .95$ ms. Latency for the same operation without ICS, i.e., when using a CNG service running locally in the root partition, is $\sim .88$ ms. The micro-benchmark result shows that ICS provides the added isolation benefits at a modest cost.

8 Limitations

This section describes some limitations of the Credo architecture and its prototype implementation, and proposes possible solutions.

- No direct save/restore. An emancipated VM can not be saved/restored and migrated live during execution. Saving a VM requires access to memory and vCPU state from the root partition, which is explicitly disallowed by emancipation. However, a limited version of save and migration can be implemented using application specific knowledge. In particular, the applications should serialize their state onto SDisk, and emancipated VM should be shutdown. This state can be deserialized at next emancipated start of the VM.
- No live migration. Related to the issue above, an emancipated VM can not be migrated live since it requires access to guest VM's memory from the root partition. However, a model of application specific save/restore based migration is still possible. After the guest VM is safely shutdown, it needs to be *re-provisioned* for the target machine where it needs to be migrated to. Steps for those are similar to the initial provisioning steps, and require mediation from the trusted server. Note that the trusted server only needs to provision a new *activation blob* specific to the target machine. VM execution image and SDisk can be directly transferred to the target machine.
- No emancipation support for networking and no integrity protection for storage. These limitations of our prototype implementation are currently work-in-progress.

9 Related Work

Past research has proposed many approaches for building secure execution environments on commodity platforms. Flicker [22] uses AMD SVM [1] technology to execute a security sensitive code fragment, called *PAL*, inside a hardware based minimal secure execution environment. However, on demand setup of this execution environment is costly and it cannot execute concurrently with the untrusted environment. TrustVisor [21] provides a hypervisor based secure execution environment for executing PALs that provides better performance and eases certain constraints. TrustVisor uses mechanisms similar to that of Credo, namely trusted launch of hypervisor, page-table based memory protection from hypervisor, and a hypervisor based virtual TPM to record integrity measurements. However, an application partitioning based solution, such as Flicker and TrustVisor, does not lend itself easily to a virtualization based cloud environment that requires execution of full fledged VMs, some of which might be security sensitive. We argue that Credo provides a practical, more flexible, approach to security in a cloud since the VM based secure execution environment runs a full featured OS, and services don't need to be rewritten in order to run in this environment.

Credo shares many of its design goals with Terra [10], which provides a trusted virtual machine monitor *TVMM* that enables security for isolated *closed box* VMs, similar to emancipated VMs provided by Credo. However, the system has a large platform TCB which includes the VMM and the host OS. Disaggregation based approaches for security in virtualized systems [27, 9, 34] do not reduce the size of platform TCB. However, they do improve the overall security of the system by making it harder for compromise of one component to impact the whole TCB of the VM. Disaggregation presents engineering and management challenges - functionality of virtualization stack must be torn apart, dependencies may need to be replicated, new interfaces must be created and maintained for inter-partition communication, and all of these privileged component must now be managed to keep them up-to-date. Credo avoids these costs by leveraging DRTM principles to reduce the runtime platform TCB of a VM just to the hypervisor.

Trust in virtual machines, and cloud computing infrastructures in general, is a matter of active ongoing debate. In certain specific scenarios, such as storage [20], group collaboration [8], and detecting tampering with online gaming state [15], it is possible to relax trust requirements from the infrastructure. However for security sensitive general purpose computation, it is required that the infrastructure provide strong trust guarantees. Credo provides such an environment with minimal TCB based on the hypervisor that is easy to attest to, compared to other virtualization based approaches to maintaining VM integrity that have large TCBs that includes privileged partitions, their administrators, and complex policies [33].

10 Conclusions and Future Work

This paper presents the design and prototype implementation of the Credo architecture. Credo provides a *small platform TCB* based on minimal hardware, firmware, and hypervisor to *emancipated* VMs. This results in a huge reduction of dynamic and extensible components that are otherwise in the TCB of a VM on commodity virtualization platforms. Although we don't have authoritative numbers, general purpose OSes are usually tens of millions of lines of code [37]. Credo removes these from VM's TCB, while adding only $\sim$ 15K lines of code overall to Hyper-V virtualized environment. In particular, Credo protects guest VMs against threats from a malicious administrator and malware in the root partition by removing it from the platform TCB. Using isolation, supplemented with cryptographic and trusted computing based mechanisms, such as TPM-based sealing of information and remote attestation, Credo provides secrecy and integrity guarantees to an emancipated guest VM against the privileged root partition.

As part of future work, we plan to extend the current *details* only model of guest measurement to include *authorities* model as well. In this model, the trusted server preparing the VM image will also sign the details vPCR with a certificate. The public key from this certificate will form the basis for the *authorities* vPCR. This will enable further attestation scenarios based on "who" provided the VM, as well as "what" runs inside them. We also plan to extend Credo with mechanisms for runtime integrity management of guest VMs. This will enable the attestation of not only the load time integrity state of VMs as provided by Credo today, but also of the dynamic runtime state.

References

[1] AMD. Secure Virtual Machine Architecture Reference Manual, 2005. `http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf`, last accessed Oct 5, 2010.

[2] B. Baker and W. Arbaugh. Hyper-V Security, 2008. Keynote address at VMSec 2008, Slides available from `http://csis.gmu.edu/VMSec/presentations/Hyper-V_Security_Baker.ppt`.

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

[4] M. Brinkmann. Bitlocker Versus True Crypt Performance, 2009. `http://www.ghacks.net/2009/11/26/bitlocker-versus-true-crypt-performance/`, accessed February 2010.

[5] C. Devine and G. Vissian. PCI bus based operating system attack and protections, 2009. Presented at EU-SECWEST 2009 conference, Amsterdam.

[6] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2, 2008. IETF RFC 5246. Available from `http://tools.ietf.org/html/rfc5246`, last accessed Oct 5, 2010.

[7] P. England and J. Löser. Para-virtualized tpm sharing. In *TRUST*, pages 119–132, 2008.

[8] A. J. Feldman, W. P. Zeller, M. J. Freedman, , and E. W. Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Proc. of OSDI*, 2010.

[9] K. Fraser et al. Safe hardware access with the Xen

virtual machine monitor. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

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

[11] F. Gens. IT Cloud Services User Survey, pt.2: Top Benefits & Challenges, 2008. Available from `http://blogs.idc.com/ie/?p=210`, last accessed Oct 8, 2010.

[12] R. P. Goldberg. *Architectural principles for virtual computer systems*. PhD thesis, 1972.

[13] A. Haeberlen. A case for the accountable cloud. In *Proc. of LADIS*, 2009.

[14] J. A. Halderman et al. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5), 2009.

[15] A. Harberlan, P. Aditya, R. Rodrigues, and P. Druschel. Accountable Virtual Machines. In *Proc. of OSDI*, 2010.

[16] Intel. Intel Trusted Execution Technology, 2010. `http://www.intel.com/technology/security/`, last accessed Oct 9, 2010.

[17] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proc. of IEEE Symposium on Security & Privacy*, 2006.

[18] G. Klein et al. seL4: formal verification of an OS kernel. In *Proc. of SOSP*, 2009.

[19] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009.

[20] P. Mahajan et al. Depot: Cloud Storage with Minimal Trust. In *Proc. of OSDI*, 2010.

[21] J. M. McCune et al. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE Symposium on Security & Privacy*, Oakland, CA, 2010.

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

[23] Microsoft. BitLocker Drive Encryption, 2010. `http://technet.microsoft.com/en-us/library/cc731549(WS.10).aspx`, accessed May 2010.

[24] Microsoft. Cryptography API: Next Generation, 2010. `http://msdn.microsoft.com/en-us/library/aa376210(VS.85).aspx`, accessed January 2010.

[25] Microsoft. Hyper-V, 2010. `http://msdn.microsoft.com/en-us/library/cc768520.aspx`, accessed May 2010.

[26] Microsoft. Windows Automated Installation Kit (Windows AIK), 2010. Available from `http://technet.microsoft.com/en-us/library/cc748933(WS.10).aspx`.

[27] D. Murray, G. Miob, and S. Hand. Improving Xen Security Through Disaggregation. In *Proc. of VEE*, Seattle, WA, March 2008.

[28] B. Parno, J. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers . In *Proc. of IEEE Symposium on Security & Privacy*, 2010.

[29] PassMark. PassMark PerformanceTest Benchmark, 2010. `http://www.passmark.com/`.

[30] O. PKI. AIK certificate creation cycle, 2010. `http://opentc.iaik.tugraz.at/index.php?item=pca/pca_aik_create`, accessed January 2010.

[31] 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 *Proc. ACM CCS*, 2009.

[32] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[33] J. Schiffman, T. Moyer, C. Shal, T. Jaeger, and P. McDaniel. Justifying integrity using a virtual machine verifier. In *Proc. of ACSAC*, Honolulu, HI, Dec. 2009.

[34] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proc. of ACM EuroSys*, 2010.

[35] TCG. Tpm specification version 1.2, 2007. Available from `www.trustedcomputing.org`.

[36] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[37] Wikipedia. Source lines of code, 2010. `http://en.wikipedia.com/wiki/Source_lines_of_code`.