Verifying System Integrity by Proxy*

Joshua Schiffman, Hayawardh Vijayakumar, and Trent Jaeger
{jschiffm,hvijay,tjaeger}@cse.psu.edu

Pennsylvania State University

Abstract. Users are increasingly turning to online services, but are concerned for the safety of their personal data and critical business tasks. While secure communication protocols like TLS authenticate and protect connections to these services, they cannot guarantee the correctness of the endpoint system. Users would like assurance that all the remote data they receive is from systems that satisfy the users' integrity requirements. Hardware-based integrity measurement (IM) protocols have long promised such guarantees, but have failed to deliver them in practice. Their reliance on non-performant devices to generate timely attestations and ad hoc measurement frameworks limits the efficiency and completeness of remote integrity verification. In this paper, we introduce the *integrity verification proxy* (IVP), a service that enforces integrity requirements over connections to remote systems. The IVP monitors changes to the unmodified system and immediately terminates connections to clients whose specific integrity requirements are not satisfied while *eliminating the attestation reporting bottleneck* imposed by current IM protocols. We implemented a proof-of-concept IVP that detects several classes of integrity violations on a Linux KVM system, while imposing less than 1.5% overhead on two application benchmarks and no more than 8% on I/O-bound micro-benchmarks.

1 Introduction

Traditionally in-house computing and storage tasks are becoming increasingly integrated with or replaced by online services. The proliferation of inexpensive cloud computing platforms has lowered the barrier for access to cheap scalable resources, but at the cost of increased risk. Instead of just defending locally administered systems, customers must now rely on services that may be unable or unwilling to adequately secure themselves. Recent attacks on cloud platforms [8] and multinational corporations [55] have eroded the public's willingness to blindly trust these companies' ability to protect their clients' interests. As a result, the need for effective and timely verification of these services is greater than ever.

Recent advances in trusted computing hardware [64, 22, 1] and integrity measurement (IM) protocols [39] aim to achieve this goal, but current approaches are insufficient for several reasons. First, existing protocols depend on *remote attestation* to convey information about a proving system's configuration to a relying party for verification. However, an attested configuration is only valid at the time the attention was generated, and any changes to that configuration may invalidate it. Since the proving system's components may undergo changes at anytime, a relying party must continually request fresh information to detect a potential violation of system integrity. This problem is made worse by the significant delay introduced by many IM protocols' reliance on the Trusted Platform Module [64] (TPM), a widely-deployed and inexpensive coprocessor, to generate attestations. Since the TPM was designed for cost and not speed, it is only capable of producing roughly one attestation per second [59, 33, 34]. This renders TPM-based protocols far too inefficient for interactive applications and high demand scenarios.

* This material is based upon work supported by the National Science Foundation under Grant No. CNS-0931914 and CNS-1117692.

Another limitation of current IM approaches is how *integrity-relevant events* are monitored on the proving system. Systems undergo numerous changes to their configurations due to events ranging from new code execution to dynamic inputs from devices. While various measurement frameworks have been developed to enable these components to report arbitrary events and its associated content (e.g., memory pages and network packets), conveying everything is impractical due to the sheer volume of data and effort placed on relying parties to reason about it. Moreover, not every event may have a meaningful effect on the system and communicating such events is a further waste. Thus, proving systems often make implicit assumptions to remove the need to collect particular measurements (e.g., programs can safely handle all network input), which may not be consistent with the trust assumptions of the relying party. This problem stems from the onus placed on the proving system's administrator to choose and configure how the various IM components will collect information without knowledge of relying party's requirements.

To improve the utility of existing IM mechanisms, we propose shifting verification from the relying party to a *verification proxy* at the proving system. Doing so eliminates the bottleneck caused by remote attestation (and thus the TPM) from the critical path, by using traditional attestation protocols to verify the proxy and the proxy to verify the proving system's runtime integrity is maintained. Monitoring the system locally also permits the proxy to examine information relevant to the relying party's integrity requirements. Moreover, this approach supports the integration of fine-grain monitoring techniques like virtual machine introspection (VMI) into remote system verification that would otherwise be difficult to convey over traditional attestation protocols [16, 17, 30] or require modification to the monitored system.

In this paper, we present the *integrity verification proxy* (IVP), an integrity monitor framework that verifies system integrity at the proving system on behalf of the relying party clients. The IVP is a service resident in a virtual machine (VM) host that monitors the integrity of its hosted VMs for the duration of their execution through a combination of loadtime and VMI mechanisms. Client connections to the monitored VM are proxied through IVP and are maintained so long as the VM satisfies the client's supplied integrity criteria. The IVP framework is able to verify a variety of requirements through an extensible set of measurement modules that translate a client's requirements into VM-specific properties that are then tracked at runtime. When an event on the VM violates a connected client's criteria, immediate action is taken to protect that client by terminating the connection.

However, we faced several challenges in designing an IVP that can be trusted to verify the target system. First, the proxy itself must be simple to verify and able to maintain its integrity without the need for frequent attestation. We employed previous efforts in deploying static, verifiable VM hosts [46] to achieve this. Second, introspecting directly on the running VM can introduce significant performance overhead if done naively. Instead, we monitor the integrity of the VM's enforcement mechanisms by leveraging practical integrity models [28, 60, 49] to identify specific enforcement points that are critical for protecting the system's integrity. By monitoring these enforcement points, we reduce the frequency and impact of verification. Finally, managing multiple channels from the same and different clients introduces redundant criteria verification. We eliminate this redundancy by aggregating multiple connections for a single criteria.

We implement a proof-of-concept IVP for an Ubuntu VM running on a Linux Kernel-based Virtual Machine (KVM) host. We constructed both loadtime and custom CPU register-based VMI modules for monitoring VM enforcement mechanisms. We validated our proxy's ability to detect violations correctly by building and attacking a VM designed to satisfy integrity criteria based on a practical integrity models and several kernel integrity requirements. We further evaluated the performance impact the IVP imposed on monitored VMs, finding that it introduced less than 1.5% overhead on two application-level benchmarks.

The rest of this paper is organized as follows. Section 2 provides background on current IM approaches and elaborates on the limitation of current IM protocols. Section 3 enumerates our design goals, presents the IVP architecture broadly, and highlights the main design challenges. Section 4 describes of our implemen-

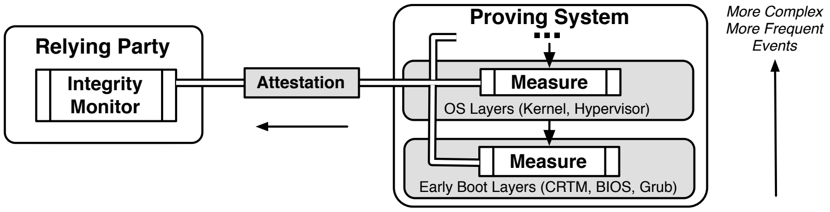

Fig. 1. A relying party's *integrity monitor* inspects a remote system's integrity by requesting *attestations* of *integrity-relevant events* collected by the proving system's layers of *integrity measurement*.

tation, which is followed by evaluation of functionality and performance in Section 5. Finally, we provide related work in Section 6 before concluding in Section 7.

2 Remote Integrity Verification

In this section, we present background on remote integrity verification and its building blocks: measurement and attestation. We then discuss the challenges current approaches face and show why they are insufficient for monitoring dynamic systems.

2.1 Integrity Verification Overview

Figure 1 provides a conceptual view of the remote integrity verification, where a *relying party* wants to determine whether a *proving system*'s current configuration (e.g., running code and data) satisfies the verifier's *integrity criteria* for a trustworthy system. The proving system has integrity measurement for its early boot layers that then measures the operating system code and data, which in turn may measure user code, data, and operations (e.g., VMs and processes). Each individual layer aims to measure the *integrity-relevant events* occurring at the layer above. The relying party *monitors* these events by requesting *attestations* of the measured events to evaluate satisfaction against the integrity criteria. If the proving system fails to satisfy the criteria, the monitor protects the relying party by denying access to the untrustworthy system. Thus, the monitor enforces an integrity policy (the criteria) over the communication to proving systems. Its role is similar to that of a reference monitor [2] that enforces access control policies over resources.

Traditionally, the monitor resides on the relying party and receives measurements provided by the proving system. *Remote attestation protocols* enable proving systems to attest to the integrity and authenticity of measurements collected on the system to relying parties. The Trusted Computing Group specifications use a request-response protocol to ensure freshness of attestations as well [43].

In order to assess system integrity accurately, the integrity monitor must observe events relevant to its integrity criteria. For example, criteria demanding enforcement of an information flow lattice might require that only trustworthy code are loaded into privileged processes and critical system files may only be written to by such processes. Thus, the monitor would require the combination of measurement mechanisms on the proving system (its *integrity measurement (IM) framework*) to record these events. We now provide a brief overview of existing measurement and attestation techniques to illustrate how an integrity monitor would use them, but provide a broader review in Section 6.

Measurement A relying party's ability to judge system integrity is limited by which events are recorded and their detail. A framework with greater coverage of system events will be more capable of measuring the required integrity criteria for more complex configurations at higher layers. We divide these measurement techniques into two categories: (1) *loadtime* and (2) *runtime*. Loadtime measurements involve capturing changes to the system like code loading and data input *before* they occur. For example, the Integrity Measurement Architecture (IMA) measures binaries before they are mapped into a running process [43] and Terra hashes VM disk blocks before they are paged into memory [16]. Others like Flicker [33] and TrustVisor [32] leverage hardware isolation to reduce the TCB down to a single running process. To measure other events, such as the data read and written by processes, some IM approaches measure other loadtime events. For example, PRIMA [23] measures the mandatory access control policy governing processes at loadtime.

Loadtime only frameworks assume that system integrity is maintained if all loadtime measurements are trustworthy. However, unexpected runtime events like code injection attacks or difficult to assess inputs like arbitrary network packets can subvert system integrity. To address this, runtime measurement techniques have been designed to record this class of events. Furthermore, mechanisms like Trousers [65] for userspace processes and the vTPM [9] for virtual machines (VMs) enable these entities to report integrity-relevant events to an external IM framework.

However, mechanisms that report on a component's integrity from within run the risk of being subverted if the processes or VM is compromised. As an alternative, external approaches like VM introspection (VMI) enables a hypervisor to observe runtime events isolated from the watched VM [41, 20, 40]. Recent VMI techniques [30, 50, 25] use hardware memory protection and trampoline code to trap execution back to the host for further inspection. While runtime measurement can detect changes at a finer granularity than loadtime measurements, they also introduce greater complexity. In particular, external approaches introduce a semantic gap that require domain knowledge like memory layouts to detect malicious modifications [7].

Attestation Early remote attestation efforts like Genuinity [26] and Pioneer [48] demonstrated the feasibility of software-based attestation, but were limited to specific, controlled environments. Specialized hardware approaches offered increased protection for the measurement framework by isolating it from the monitored system [4, 42]. Hardware security modules (HSMs) like the IBM 4758 used an early attestation technique called Outbound Authentication [54] to certify the integrity of installed code entities via certificate chains. However, such specialized hardware imposed a significantly higher deployment cost and complexity.

The Trusted Platform Module (TPM) [64] was introduced to provide commodity HSMs across numerous consumer electronic devices. The TPM facilitates several cryptographic features like key generation, signing, and encryption. It also supports remote attestation through a set of platform configuration registers (PCRs) that store measurements (e.g., SHA-1 hashes) of integrity- relevant events. Measurements taken on the system are *extended* into the PCRs to form an append-only hash-chain. A relying party then requests an attestation of the recorded measurements by first providing a nonce for freshness. In response, the TPM generates a digital signature, called a *quote*, over its PCR values and the nonce. An asymmetric private key called an Attestation Identity Key (AIK) is used to sign this quote. The AIK is certified by a unique key burned into the TPM by the device's manufacturer, thereby binding the attestation to the physical platform. The proving system then provides the quote and list of measurements to the relying party. If the quote's signature is valid and the measurement list produces the same hash-chain as the quoted PCRs, then the measurements came from the proving system.

2.2 Integrity Monitoring Challenges

For the integrity monitor to verify system integrity accurately, its view of the proving system must be both fresh and complete. Stale or incomplete measurements limit the utility of the verification process. However, we find that current attestation-based verification model are insufficient for several reasons.

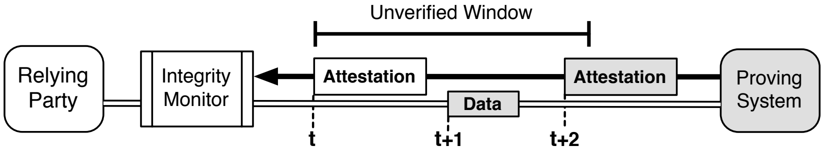

Fig. 2. A window between each attestation exists where the integrity of the proving system is unknown.

Stale measurements Attestation-based protocols introduce a window of uncertainty, which we illustrate in Figure 2. Here, the integrity monitor residing on the relying party requests an attestation at time t and finds it satisfies its integrity criteria. Since the prover is verified, the monitor permits it to send data to the relying party at $t + 1$. Later, the monitor requests a second attestation at $t + 2$ and finds the prover no longer satisfies the criteria. Because this violation could have happened at anytime between t and $t + 2$, it is not clear without additional information if the data at $t + 1$ was generated when the system was unacceptable. Classic attestation protocols like IMA [43] avoid this issue by buffering inputs until a later attestation is received, but this is not an option for high throughput or interactive applications.

Hardware bottleneck Many systems are dynamic and undergo numerous changes at any time. Thus, the monitor must continually poll for new attestations to detect changes. This problem is exacerbated by the TPM's design as a low performance device for attesting infrequent loadtime measurements like the boot process. In fact, current TPM implementations take approximately one second to generate a quote leading to major bottlenecks in any high demand scenario [59]. Designs that batch remote attestations to eliminate queueing delays have been proposed [34], but still incur a significant overhead.

Criteria insensitive measurements A relying party's ability to assess system integrity is also limited by what events are measured. Since the proving system's administrator decides what the measurement framework will record, a remote verifier must often settle for the information provided by proving system. If that system provides only hashes of code loading operations, then a criteria requiring knowledge the possible runtime operations of those processes cannot be satisfied. However, it is difficult to know what information arbitrary clients require, which is especially challenging for public-facing services used across multiple administrative domains. On the other hand, designing an IM framework to record excessive measurements may be wasteful if they are inconsequential to the verifier's integrity criteria. Moreover, complex events occurring within an entity like may be difficult to assess. For example changes to kernel memory may indicate a rootkit, but it is hard to make that judgement without knowledge of where certain data structures are located. However, providing this context (i.e., entire memory layouts) via attestation can be impractical.

3 Integrity Verification Proxy

We now present the design of the *integrity verification proxy* (IVP), an integrity monitor framework that verifies system integrity at the proving system on behalf of the relying party. By shifting a portion of the integrity monitor to the proving system, the IVP eliminates the need for continuous remote attestation and provides direct access to the system's IM framework to support a broad range of integrity criteria. We begin by describing our design goals and trust assumptions. We then give an overview of the IVP's architecture and detail how it achieves these goals.

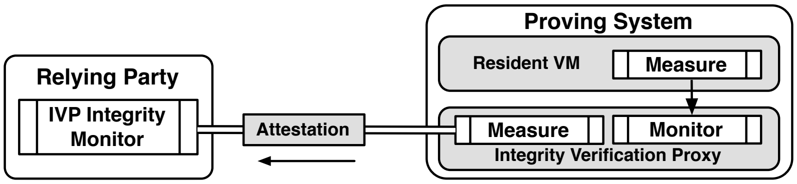

Fig. 3. The *integrity verification proxy* (IVP) acts as an integrity monitor on the proving system that monitors the resident VM to enforce the relying party's criteria over the communication channel. The long-term integrity of the IVP and its host (i.e., layers below the resident VM) is verified by traditional loadtime attestation.

3.1 Design Goals

Our aim is to extend the traditional notion of an integrity monitor into the proving system to overcome the limitations of current attestation-based verification protocols. Figure 3 shows the conceptual model of this approach. This model supports the following design goals.

Enforce integrity criteria at the proving system. Monitoring system integrity remotely is insufficient because stale knowledge of the remote system's more complex events undermines the monitor's ability to correctly enforce its criteria. Instead, a relying party can establish trust in an integrity monitor on the proving system that enforces its integrity criteria. The IVP has direct access to resident VM's IM framework to eliminate the window of uncertainty caused by attestation protocols. Moreover, the IVP can terminate connections immediately when an integrity violation is detected to protect the relying party. If the relying party is also the administrator of the VM, the IVP can take further remedial measures such as rebooting the VM. However, the relying party must still monitor the IVP itself to justify such trust. Thus, the IVP must be deployed at a software layer whose integrity can be verified by the relying party without the need for continual attestation, or the purpose of moving monitoring to the proving system is defeated.

Criteria-relevant measurement. The problem with traditional IM frameworks is that they measure events irrespective of what the relying party requires. Moreover, entities on the resident VM may be implicitly trusted by the administrator and thus are not monitored. An effective IVP must support various integrity criteria that may even differ from administrator's criteria. To do this, the IVP leverages the available information about the resident VM to capture a broad set of integrity-relevant events to support differing criteria. In Figure 3, the IVP extracts information from both the IM framework on the proving system and additional information through external measurement techniques like VM introspection.

3.2 Assumptions

We make the following trust assumptions in the IVP design. First, we do not consider physical attacks on hardware, denial-of-service attacks, or weaknesses in cryptographic schemes. Next, we assume that the relying party and all the events allowed by the integrity criteria to be trustworthy. Moreover, we treat events that cannot be captured by the IM framework to be acceptable because we cannot say anything about their existence. It is important to note that such unobserved events may be harmful, but unless a mechanism can detect the degradation, it is hard to know the harm that has occurred. We consider the following threats in the IVP design. We assume a powerful external adversary who can produce external events upon the proving

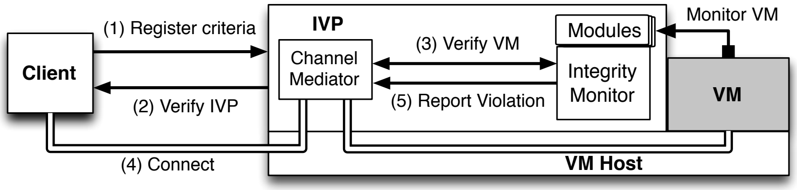

Fig. 4. Integrity verification proxy architecture.

system that may exploit vulnerabilities. Such external events may affect both loadtime (e.g., modify files in a downloaded distribution) and runtime events (e.g., network communications). Finally, we consider attacks that modify remote storage and offline attacks on the proving system's local disk.

3.3 Architecture Overview

Figure 4 illustrates our architecture for enforcing the integrity criteria of a relying party (the client) over a network connection to an application VM. Here, the IVP is a service resident in the VM's host that verifies the integrity of the VM on behalf of the client. The client first (1) registers her integrity criteria with the IVP service. Next, (2) she establishes trust in the VM's host and IVP service by verifying their integrity through traditional attestation protocols. These components are designed to maintain their integrity at runtime, thereby enabling simple verification through loadtime measurements similar to existing protocols like IMA [43]. This verification is needed to trust the IVP to correctly enforce her criteria.

The client then requests a connection to a specific hosted VM the criteria to enforce over the channel. The IVP's *integrity monitor* is responsible for tracking the ongoing integrity of the hosted VMs relative to the client's criteria. It uses a set of *measurement modules* to interface directly with the host's IM framework and capture integrity-relevant events, which are reported back to the monitor. If the monitor (3) determines that the VM satisfies the client's criteria, it then (4) establishes a secured network tunnel between the client and VM through the IVP's *channel mediator*. The mediator associates each tunnel with the client's criteria. If the integrity monitor detects a that a VM has violated the criteria of any connect client, it notifies the mediator to (5) terminate each associated connection.

3.4 Verifying the IVP Platform

The IVP verifies VM integrity on behalf of the client, thereby requiring trust in the IVP. Since our aim is to reduce client verification effort and eliminate the need for repeated remote attestation, we want an IVP that can be verified by a single attestation at channel setup unless a reboot occurs. The challenge is then building IVPs and their hosting platform in such a way that they maintain their integrity to obviate the need for remote monitoring.

This endeavor is difficult in general because systems often have large TCBs consisting of numerous components that may not be trusted. Moreover, changes to these systems at runtime like upgrades may be overlooked without frequent monitoring. However, various research projects have explored techniques for building VM hosting platforms that may be small enough to verify formally [27, 3, 5, 57, 32, 58]. While the design of a specific platform is outside the scope of this paper, we envision a host would incorporate such approaches. As for the IVP, it only relies on a small number of services, such as networking, the introspection interface, and VM management. Research projects like Proxos [61] and work by Murray et. al. [35], have demonstrated that it is possible to build minimal VMs that depend only on the VMM and use untrusted services in other VMs securely (e.g., by encrypting and integrity-protecting the data). This would enable the IVP to function as an independent service in the host without depending on a large host VM like in Xen Dom0. We intend to develop future IVP prototypes for various hypervisors that support this separation.

3.5 Channel Mediation

The IVP is responsible for mediating connections to ensure they are active only when their respective client's criteria are satisfied. The channel mediator creates an integrity association (IA) for each tunnel as the tuple (C, V, I), where C is the client, V is the VM, and I is the integrity criteria to check. Before a tunnel is brought up, the IA is registered with the integrity monitor to verify that V continues to satisfy I. If it does, the tunnel is brought up and shutdown either voluntarily or when the integrity monitor notifies the mediator that an I has been violated.

One challenge in designing the channel mediator is proving to clients that the channel is controlled and protected by the proxy. The connection is formed as an Ethernet tunnel between the client and the VM through a virtual network managed by the mediator. This effectively places the client and VM on the same local subnet. Other mediated connections to the VM connect over the same virtual network, but are isolated from each other by the mediator using VLAN tagging. During setup, the tunnel is protected via cryptographic protocols like TLS that mutually authenticate the client and mediator. The VM is provided a certificate signed by the host's TPM at boot time to bind the platform's identity to the VM's credentials. This binding approach is similar to previous work on VM attestation [9, 19]. The client can then setup further protections directly with the VM over the tunnel. Having direct control over the network tunnel also lets the mediator tear down the connections as soon as a violation is detected.

3.6 Integrity Monitoring

The IVP's integrity monitor is tasked with verifying each VM's integrity against integrity criteria registered by clients connected to it. To do this, the monitor collects events from its measurement modules (see Section 3.7) to update its view of each VM's configuration. When the mediator registers an IA, the monitor first checks if the IA's criteria is satisfied by the current VM configuration. If so, the monitor adds a reference to the IA to a list of IAs to verify. When the VM's configuration changes, (e.g., through code loading) the integrity monitor pauses the VM and checks whether any registered IA has been violated. If so, the channel mediator is notified of the invalid IA, so it may tear down the tunnel before the VM can send data on it. The monitor then resumes execution of the VM.

In order to verify a VM's integrity, the monitor must be able to capture all integrity-relevant changes from VM creation until shutdown. To monitor loadtime events, we give the integrity monitor direct control over VM creation through the platform independent virtualization API, libvirt. This lets the monitor collect information about the VM's virtual hardware, initial boot parameters, kernel version, and disk image. The monitor spawns individual watcher threads for each VM and registers with the IVP's measurement modules. When the modules capture an event at runtime, the watcher is alerted with the details of the change. Since multiple IAs to the same VM may have redundant requirements to verify, the monitor keeps a lookup table that maps IAs with the same criterion together. When a change to the VM is detected that violates one of these conditions, all IAs mapped to that criterion are invalidated by the monitor.

3.7 Measurement Modules

Integrity criteria consist of various loadtime and runtime requirements. The integrity monitor divides up these criteria into a set of discrete measurement modules tasked with tracking changes to specific aspects of the VM's configuration. The modules interface directly with the available IM framework to measure events in real time. For example, loadtime modules measure information like boot time parameters of the VM, while runtime modules attach a VMI to watch critical data structures. Since IM frameworks often consist of several components responsible for measuring various events, modularizing the interface allows for a more flexible design. Administrators can then write or obtain modules for the specific IM mechanism installed on the host without having to modify the monitor.

Capturing runtime events. Detecting violations at runtime requires modules to be able to capture events as they happen. The module must then notify the integrity monitor's watcher of the event before the VM continues to execute. We employ VMI to enable our modules to monitor runtime criterion. Many hypervisors now offer VMI mechanisms like xenaccess [40, 15, 21] in Xen and VMSafe [66] for VMware that enable direct access to VM resources. In addition, QEMU supports introspection through debugging tools like `gdb` and previous work has demonstrated the feasibility of VMI in KVM [50].

Each runtime module monitors a specific property on the VM. The modules actively monitor the VMs by setting *watchpoints* (e.g., locations in memory) that are triggered by integrity-relevant operations. Watchpoints can be set on sensitive data structures or regions of memory such as enforcement hooks [53], and policy vectors [37] stored in kernel memory. Other structures like function pointers and control flow variables are possible candidates [11]. Triggering a watchpoint pauses the executing VM so the module that set the watchpoint can examine the how the configuration has been altered. Pausing the VM prevents the VM from sending any data on the connection until the module can assess if the event violated an IA's criteria. After the module finishes invalidating any IAs, the VM is permitted to resume execution.

Improving efficiency. VMI gives runtime modules direct memory access, but creates a semantic gap [12] when reading directly from the VM's memory. Since the module does not have the full context of the running system, changes to complex and userspace data structures are difficult to assess. Our modules leverage the VM's extant enforcement mechanisms to report events without having to pause the VM as often. For example, instead of pausing the VM to measure every executed program, we use Linux kernel's Integrity Module (LIM) [29] framework to record hashes of every previously unseen program and executable memory-mapped file before loading them. We set a watchpoint on the in-kernel measurement list to catch each addition to it. This way, the module can avoid pausing except when LIM detects new binaries. Other in-VM monitor techniques could be leveraged to report integrity measurements to the modules to reduce the overhead of pausing the VM. Virtual devices like the vTPM [9] and co-resident monitors like SIM [50] provide potential reporting frameworks.

4 Implementing an IVP

We implemented a proof-of-concept IVP for a Linux KVM system. Figure 5 illustrates the IVP's services residing in the host. Clients interact with the IVP through a *proxy manager* to (1) register their criteria, (2) request attestations of the host's configuration, and (3) manage connections to VMs. We used a TLS-protected VPN tunnel to the VM's virtualized private network to implement the IVP's channel mediator. Initially, VMs are firewalled from the client's tunnel and all clients are isolated from each other through the VPN as well. Once the tunnel is active, a client can establish an IA with a specific VM by first (3a) sending a request to the proxy manager and specifying which criteria previously registered should be used to mediate that connection. The proxy manager then creates the IA tuple and (3b) registers it with the integrity monitor, which in turn checks if the client's criteria are satisfied by the VM. If it is, the monitor (3c) informs the proxy manager to change the VPN firewall to allow the VM to send data to the client over the tunnel. The client can now receive data from the monitored VM as well as (4) authenticate the identity of the VM to establish an

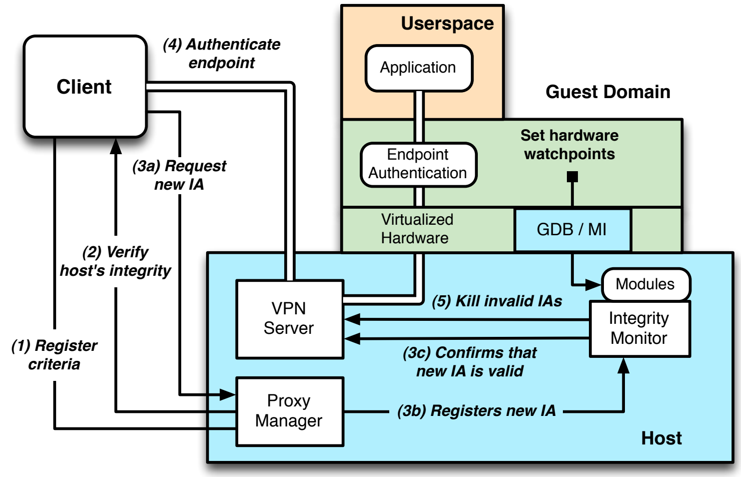

Fig. 5. IVP implementation and protocol.

encrypted connection if desired. Finally, if at anytime the VM violates the IA's criteria, the integrity monitor (5) deletes the IA and informs the VPN server to firewall the client tunnel from the VM.

4.1 Verifying the Host

To verify the IVP platform's integrity, we use the Root of Trust for Installation (ROTI) approach to attest to the trusted distribution of the host [46]. At install time, a TPM signed proof is generated that binds the installed filesystem to the installer that produced it. We also employ the `tboot` bootloader to establish a measured launch environment (MLE) for the host using Intel's Trusted eXecution Technology (TXT) in recent CPUs [22]. The MLE establishes a dynamic root of trust for measurement (DRTM) through the processor that isolates, measures, and executes the kernel and initial ramdisk (initrd). This allows the boot process to be started from a trusted starting point. The initrd loads the system enforcement policies into the kernel and takes a measurement of the current filesystem before passing execution off to the root filesystem. When a client requests an attestation of the IVP platform, the ROTI proof is included with the normal attestation. The client then checks that the proof indicates no tampering with the installation has occurred and that the installer source is trusted to produce a system designed to maintain its integrity at runtime to meet the long-term integrity requirements of the IVP platform.

4.2 Channel Mediator

We implemented the channel mediator using OpenVPN server to manage Ethernet tunnels from remote clients to the internal virtualized network for the hosted VMs. All mediated connections from the client are aggregated through a single VPN tunnel with the individual VM endpoints permitted to transmit on that tunnel if a corresponding IA exists. VPN tunnels are established by first mutually authenticating the client's account certificate and a host certificate signed by the host's AIK. Each connection is TLS-protected and uses a Linux tap device to provide kernel supported Ethernet tunneling from the physical network interface to the virtual network bridge. Once connected, the OpenVPN server opens the firewall for traffic from the

VM's virtual interface to the tunnel for each VM in the active IAs to the client. When the integrity monitor deletes an IA, it tells the OpenVPN server to firewall the VM from the client in the deleted IA.

4.3 Integrity Monitor

We created the integrity monitor as a 439 SLOC Python daemon that manages VM execution and monitors VM integrity. The daemon uses the hypervisor independent interface, libvirt, to start and stop VMs, collect information about virtual device settings, and control loadtime VM parameters. When the daemon receives a request to start a VM, it spawns a separate *watcher* thread to control the VM and monitor integrity information. When the proxy manager registers a new IA with the monitor, the monitor forwards the IA to the appropriate VM's watcher, which in turn checks that each criterion is satisfied by querying the registered measurement modules for current VM configuration. If all the modules indicate the requirements are satisfied, the watcher notifies the VPN server that the IA is valid.

The watcher registers with loadtime measurement modules to collect information about the VM before the VM is started. Next, the VM is created and the watcher attaches `gdb` to running VM process, which pauses the VM. We use `gdb` as a proof of concept VMI interface because VMs in Linux KVM run as userspace processes, making them it simple to monitor. Moreover, `gdb` can determine where kernel structures are in memory by reading debug information in the kernel or from a separate system map file that is easily obtained. The watcher then loads the runtime modules, which collect the necessary context from the paused VM and set any desired watchpoints through the `gdb` interface. After the runtime modules are registered, the VM resumes execution. When watchpoints are triggered at runtime, the VM is paused and control is passed from the watcher to the runtime module that set it. The module then introspects into the VM's memory and updates the accumulated VM configuration with any modified values detected during introspection. The module notifies the watcher if any values have changed, which checks if those changes have violated any of the registered IA's criteria. Finally, the module resumes the VM's execution.

We use hardware-assisted watchpoints in `gdb` to avoid modifying the VM code and introducing additional overhead. This raises an issue because the x86 architecture only contains 4 debug registers, which limits the number of hardware-assisted watchpoints that can be set for a process. Since software watchpoints require single stepping through the VM's execution, they are not a viable option. However, similar watchpoint functionally is feasible by using memory protection features of the KVM shadow page table for VMs as demonstrated in SIM [50]. While we did not implement this VMI approach, we plan to explore it and further implementation options in future work.

5 Evaluation

We evaluated our IVP implementation in terms of functionality and performance. First, we validated the IVP's ability enforce relying party criteria correctly by performing attacks that violated various integrity requirements. We then evaluated the performance overhead imposed on the monitored VM using both microbenchmarks and application-level benchmarks performance.

Our experimental testbed consisted of a Dell OptiPlex 980 with a 3.46GHz Intel Core i5 Dual Core Processor, 8GB of RAM, and a 500GB SATA 3.0Gb/s hard disk. The Linux KVM host ran in an Ubuntu 10.10 distribution using a custom 2.6.35 Linux kernel. Our guest VMs were allocated a single 3.46GHz vCPU without SMP, 1GB of RAM, and an 8GB QCOW2 disk image connected via virtio drivers. Each VM ran an Ubuntu Linux 10.10 server distribution with default SELinux policy and a custom LIM module.

5.1 Functionality

To test the IVP's functionality, we designed a target application VM running the Apache webserver. We constructed a VM image that approximates the CW-Lite [49] integrity model and designed an integrity criteria for verifying that approximation. We then had a client connect to the VM through a mediated channel associated with the CW-Lite criteria. We performed several attacks on the VM's loadtime and runtime integrity both before and after the connection was established to see if the IVP would correctly detect the violations and terminate the connection.

Building a CW-Lite Enforcing VM We constructed an application VM that satisfies the CW-Lite integrity model. This practical integrity model differs from strict integrity models like Biba [10] and Clark-Wilson [13] by allowing for an integrity policy that identifies trusted exceptions where illegal flows are required for the system to function properly. Other practical integrity models would also be viable [28, 60]. To enforce CW-Lite, trusted processes with high integrity labels (e.g. privileged daemons) must only (1) load trustworthy code, (2) receive trustworthy inputs, and (3) handle untrusted inputs through designated filtering interfaces that can upgrade or discard low integrity data.

We configured our Apache VM with SELinux, which enforces a mandatory access control policy through Domain Type Enforcement [6]. This labels every process and system object with policy-defined types. We use the Gokyo [24] policy analysis tool to identify 79 labels from which data can flow to the Apache process and system TCB labels [49]. This included processes that access critical resources like kernel interfaces and privileged daemons. We then modified SELinux LSM to hook into the kernel's LIM [29] to receive hashes of code executed in trusted processes. The modified LSM module then denies execution of hashes that are not on a white list obtained from the Ubuntu 10.10 main repository. This secure execution monitor satisfies the first CW-Lite requirement because only trusted code from the hash list may run in trusted processes.

In addition to the identified trusted processes, several untrusted sources like the network provide necessary input to Apache. Per the third CW-Lite requirement, we must ensure untrusted inputs are only received by interfaces[1] designed to properly handle (e.g. sanitize) such input. To do this, we added additional checks to the LIM policy to whitelist only the Apache binary, designed to handle such inputs, to be loaded into the process with labels to access these interfaces. Before the interface is permitted to read data, our modified SELinux LSM checks if the interface is intended to receive untrusted data based on a CW-Lite policy and deny the read if it is not.

Specifying integrity criteria We defined our client's integrity criteria with both loadtime and runtime requirements. For loadtime criteria, we specified hashes of a trusted VM disk image, kernel, initrd, and CW-Lite enforcement policies to match those we created above. The runtime criteria, by contrast, checks for common signs of intrusion by rootkits and unexpected modification of the VM's enforcement mechanisms and policies at runtime.

For example, previous research [7] has shown that some rootkits modify the netfilter hook in the kernel to enable remote control of the system via specially crafted network packets [36]. Other attacks replace the binary format handlers to obtain privilege escalation triggered by program execution. We specified runtime criteria that require no changes to the kernel structures located by the kernel symbols `nf_hooks` for the netfilter and `formats` for binary format handlers attacks. We also identified function pointers used to hook execution by SELinux and LIM and in-kernel policy structures that should not be modified at runtime.

[1] Interface here refers to the read-like syscalls. While programs have many interfaces, only some are intended to handle untrusted inputs.

Furthermore, we specified that only the Ubuntu repository code was to be executed in the TCB, which would catch the case where the secure execution protections were bypassed. To do this, we specified that all measurements of code loads taken by the LIM hooks should match the hash list we specified above.

Building Measurement Modules We constructed several measurement modules to monitor various integrity requirements on the target VM. The modules were written in an inheritable base class that exposes a register function for setting watchpoints and a callback handler that is called when the watchpoint is triggered. Each module averaged 25 additional lines over the base class definition. The integrity monitor's watcher thread instantiates and registers loadtime modules before the VM is first created to measure the kernel, disk image, and enforcement policies.

Runtime modules are instantiated after VM initialization and set watchpoints through the `gdb` interface. When a watchpoint is triggered, the watcher is notified and invokes the appropriate module's callback to inspect the event. We placed watchpoints on various kernel structures including SELinux, LIM, and netfilter function pointers and the binary format handler list. We also monitored the in-kernel LIM policy by set a watchpoint on the kernel's `ima_measurements` list head. This traps to the runtime module whenever a new binary is executed. The module reads the hash from the list tail and adds it to the module's list of measured code. Doing this, we can monitor all code loaded in the TCB and check for inconsistencies between the expected LIM policy and executing programs. Leveraging the LIM framework to record new code hashes lets the integrity monitor pause the VM only when new binaries are loaded.

Detecting Violations We tested if the IVP properly mediates the CW-Lite criteria before and after connecting a client to the VM over the mediated channel. We exercised each measurement module through a series of attacks on the VM's integrity. For loadtime modules, we modified boot parameters, kernel versions, disk image contents, and policy files to values not permitted by the criteria. The modules then recorded these configuration values at VM creation. When the client initiated connection request to the IVP, the integrity monitor's watcher compared the measured values to the criteria and correctly rejected the connection. For our runtime modules, deployed attacks on the monitored data structures using attack code that exploits an x86 compatibility vulnerability in Linux kernels older than 2.6.36 [14]. This let us illegally change an unprivileged process' SELinux label to the full privileged `kernel_t` label, thereby enabling arbitrary code execution. We used this vector to easily modify kernel memory and modify the monitored structures to violate our runtime requirements. The IVP correctly detected these changes and disconnected the connection to the VM and prevented future connection requests.

5.2 Performance

Next, we examined the performance impact the IVP has on monitored application VMs. We first performed a series of CPU and I/O micro-benchmarks within the monitored VM to identify any overhead in system performance indicators. We then performed macro-benchmarks with a webserver and distributed compilation VM to see the impact at the application-level.

Passive Overhead We first evaluated the impact of runtime monitoring on the VM when integrity-relevant events are not occurring. We used three types of benchmarks to test CPU, network, and disk performance of the VM with and without the IVP active. For CPU-bound benchmarking, we used the SPECINT 2006 test suite (see Table 1), which performs several training runs to identify the expected standard deviation (under 1.1%) before sampling. Most tests show negligible overhead with the IVP with the largest at 0.61%.

SPECINT '06	Median (sec)		Diff
Benchmarks	Base	Test	(%)
perlbench	403	404	0.25
bzip2	683	686	0.43
gcc	367	369	0.54
mcf	557	560	0.53
gobmk	467	467	0.00
hmmer	544	545	0.18
sjeng	575	576	0.17
libquantum	664	667	0.45
h264ref	762	763	0.13
omnetpp	494	497	0.61
astar	664	667	0.45

Table 1. Benchmarks with and without the IVP obtained by the median of three runs, as reported by the SPECINT 2006. The test suite does training and test runs in addition to the actual runs so the results are reproducible.

Operation	Mean (± 95% CI) (ms)
Watchpoint Trigger	
VM Exit and Entry	.006 (± 0.000)
QEMU overhead	.496 (± 0.081)
GDB overhead	.327 (± 0.054)
Monitor Overhead	0.172 (± 0.028)
Runtime Modules	
Collect LIM Hash	66.76 (± 0.215)
Read kernel variable	0.132 (± 0.002)

Table 2. Active Overhead Micro-benchmarks of overhead incurred when watchpoint is triggered. World switches and GDB contributes 82.2% of the trigger overhead excluding modules. Collected from 100 runs.

Table 3 shows our results for network and disk benchmarks after 30 runs of each. We used `netperf` to evaluate network overhead. It samples maximum throughput and transactions per second after saturating the network link. These tests also indicated negligible impact on networking. For disk I/O performance, we used the `dbench` benchmarking tool, which simulates a range of filesystem level operations using a configurable range of parallel processes. It presents results as the average throughput for the client processes. We found that the throughput was negatively affected as we increased the number of simultaneous clients. Our intuition for this trend was that more client processes led to more syscalls that, in turn, cause the VM process to raise signals to perform I/O through virtual devices. We profiled the VM with systrace while the benchmarks were executing and confirm this correlation. Since `gdb` uses the `ptrace` interface in the kernel to monitor processes for debug signals, every syscall incurred a small processing overhead by `gdb` to parse the signal and resume process execution. A possible solution for this would be to modify the ptrace interface to notify the `gdb` process only when debug signals are raised. Even with this overhead, our disk I/O benchmarks demonstrate overhead under 8% for 50 clients.

We also tested the effect of our IVP on two real-world applications, an Apache webserver and a `distcc` compilation VM. We initiated all of our tests from a separate computer over the TLS-protected VPN tunnel setup by the IVP. We ran 30 runs of the `ab` tool to simulate 100 concurrent clients performing 100,000 requests on the Apache VM. For the `distcc` test, we compiled Apache-2.2.19 across 3 identical VMs on separate machines with 8 concurrent threads. Again, the average of 30 such runs are taken. Our results show that the IVP introduced a 1.44% and 0.38% overhead on Apache and distcc VMs, respectively. We suspect the primary cause for the Apache overhead is the frequent network requests and disk accesses made to service the requests.

Active Overhead Finally, we explored the delays introduced by the IVP when handling changes to monitored data structures. We profiled our measurement modules using the ftrace framework in the Linux kernel by setting markers to synchronize timings between our userspace monitor and events happening in the kernel, such as VM exits and enters. Table 2 shows that interrupting the VM on a tripped watchpoint introduces a 1 ms pause regardless of the measurement module involved. For simple runtime modules that read sin-

Benchmarks	Mean ± 95% CI		Diff
	Baseline	With IVP	(%)
Network: `netperf`			
TCP_STREAM (Mb/s)	268 ± 0.23	269 ± 0.22	0.2
TCP_RR (Trans/s)	1141 ± 5.65	1141 ± 1.96	0.05
Disk: `dbench`			
1 Client (Mb/s)	11.14 ± 0.02	11.12 ± 0.14	0.18
5 Clients (Mb/s)	32.64 ± 0.67	32.49 ± 0.76	0.46
10 Clients (Mb/s)	40.94 ± 1.01	40.21 ± 0.98	1.78
20 Clients (Mb/s)	47.46 ± 1.50	44.69 ± 1.12	5.83
50 Clients (Mb/s)	40.58 ± 3.09	37.41 ± 1.86	7.81

Table 3. Network and disk benchmarks. `netperf` measures throughput (tcp_stream) and transactions per second (tcp_rr) after 30 second network saturation. `dbench` measures 20 seconds disk throughput intervals during a 10 minute read / write workload after 2 minute warmup. 30 runs per benchmark.

gle variables, approximately 100μs additional overhead is incurred. However, more complex measurement modules take more time. For example, the LIM measurement module reads a SHA1 hash from a nested kernel list, which causes a 67 ms delay. We found the majority of this is caused by `gdb` parsing the kernel symbol table to locate the memory addresses in the VM to read. Caching these addresses when the monitor is registered would greatly speed this measurement process. Regardless, measurement modules that perform more complex measurements like reading and parsing multiple structures will increase the time the VM is paused. Moreover, watchpoints on frequently modified memory locations will result in more pauses.

6 Related Work

Introduction of the TPM has led to numerous IM techniques (see the comprehensive survey by Parno et. al. [39]). Initial approaches focused on TCG-style verification of the boot process, the OS kernel, modules, userspace binaries [43, 29] and system policies [23]. Application-level measurements through frameworks like Trousers [65] enable processes to pass measurements to the TPM for integrity protection and reporting. Other techniques measured VM integrity through hypervisor support [16, 44, 31] and even virtualized the TPM for VMs [9]. More recently, Sirer et. al. [52] proposed an authorization logic supported by a custom OS kernel that enables verification using high-level statements instead of binary hashes. This approach greatly simplifies the complexity of verifying attestations and provides a richer measurement framework for both local and remote entities. However, these approaches place the verification burden on the relying party to interpret potentially stale and incomplete information. The IVP can leverage these disparate measurement techniques to verify a relying party's criteria at the proving system and supplement them with more fine-grain monitoring.

Other approaches have focused on reducing the TCB that must be verified. Bind [51], Flicker [33], and TrustVisor [32] use CPU hardware support to measure and protect the execution environment of application code and associate it with the computation's result. These approaches provide guarantees to the relying party that the result was protected from external threats during execution, but still require verification of each result's attestation.

Instead of attesting system configurations, other research has focused on maintaining runtime integrity guarantees [47, 30, 17, 50, 45, 5, 56] that remote parties can verify are being enforced. For example, Terra's

Optimistic Attestation ensure certain VM disk blocks are unaltered by shutting down the VM if a modification is detected at loadtime. These approaches offer a strong foundation for monitoring runtime integrity, but do not support verifying remote verifier specified requirements. Our IVP can leverage these runtime enforcement mechanisms to maintain the IVP host's integrity. Furthermore, remote parties can use the IVP to monitor the integrity of enforcement mechanisms in the VM and their policies. Also, our design does not explicitly provide remediation like shutting down the VM because we assume the remote clients are not administrators of the VM and may have differing criteria.

IM has also been incorporated into secure communication channels. Trusted Network Connect [63] requests periodic attestations of clients before and after they join a private network and evicts systems with invalid attestations. OpenTC PET [38] uses SSL proxies in a VM host to provide attestations of the VM to the remote client. However, the proxy simply provides attestations instead of verifying the VM's integrity on behalf of the connected client. Other work [19, 62, 9, 18] has incorporated TPM attestations into public key certificates to bind integrity states to platform identities. However, the reported integrity of these approaches is only valid as long as the attested system's configuration does not change. This requires the client to continually request new certificates that function exactly like attestations. Our IVP eliminates the need for continual polling by enforcing the client's criteria at the VM's host.

7 Conclusion

In this paper, we presented the integrity verification proxy (IVP), a service resident in a proving system that mediates connections on behalf of remote clients. By shifting the task of monitoring a client's integrity criteria to the proving system's host, we enable relying parties to connect to remote systems without the need for frequent attestations or further verification. We designed and implemented a proof of concept IVP for a Linux KVM host and evaluated its effectiveness and impact on performance. Our results show the IVP incurs only minor overhead for network and CPU-bound applications, but with additional delay that increases modestly as a function of I/O load. As future work, we plan to improve our VMI interface to minimize passive overhead and increase expressiveness of client's integrity criteria.

References

1. Processor-Based Virtualization, AMD64 Style. http://developer.amd.com/documentation/articles/pages/630200615.aspx
2. Anderson, J.P.: Computer Security Technology Planning Study. Tech. Rep. ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA (1972)
3. Andronick, J., Greenaway, D., Elphinstone, K.: Towards Proving Security in the Presence of Large Untrusted Components. In: Proc. 5th Workshop on Systems Software Verification (2010)
4. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A Secure and Reliable Bootstrap Architecture. In: Proc. IEEE SSP (1997)
5. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In: Proc. 17th ACM Conference on Computer and Communications Security (2010), http://doi.acm.org/10.1145/1866307.1866313
6. Badger, L., Sterne, D.F., Sherman, D.L., Walker, K.M., Haghighat, S.A.: Practical domain and type enforcement for unix. In: IEEE Symposium on Security and Privacy (1995)
7. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proc. ACSAC (2008), http://dx.doi.org/10.1109/ACSAC.2008.29
8. BBC: Amazon apologises for cloud fault one week on. http://www.bbc.co.uk/news/business-13242782
9. Berger, S., et al.: vTPM: Virtualizing the Trusted Platform Module. In: USENIX Security Symposium (2006)

10. Biba, K.J.: Integrity Considerations for Secure Computer Systems. Tech. Rep. MTR-3153, MITRE (1975)
11. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 16th ACM conference on Computer and communications security
12. Chen, P.M., Noble, B.D.: When Virtual Is Better Than Real. In: Proc. HotOS (2001)
13. Clark, D.D., Wilson, D.R.: A Comparison of Commercial and Military Computer Security Policies. Security and Privacy 00 (1987)
14. CVE-2010-3081. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3081
15. Fraser, T., Evenson, M.R., Arbaugh, W.A.: VICI Virtual Machine Introspection for Cognitive Immunity. In: Proceedings of the 2008 ACSAC (2008), http://dx.doi.org/10.1109/ACSAC.2008.33
16. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: Proc. 19th ACM SOSP (2003)
17. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. NDSS (2003)
18. Gasmi, Y., Sadeghi, A.R., Stewin, P., Unger, M., Asokan, N.: Beyond Secure Channels. In: Proc. ACM Workshop on Scalable Trusted Computing (2007)
19. Goldman, K., Perez, R., Sailer, R.: Linking Remote Attestation to Secure Tunnel Endpoints. In: Proc. First ACM Workshop on Scalable Trusted Computing (2006), http://doi.acm.org/10.1145/1179474.1179481
20. Haldar, V., Chandra, D., Franz, M.: Semantic remote attestation: a virtual machine directed approach to trusted computing. In: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium (2004)
21. Hay, B., Nance, K.: Forensics examination of volatile system data using virtual introspection. SIGOPS Oper. Syst. Rev. 42, 74–82 (April 2008)
22. Trusted Execution Technology. http://www.intel.com/technology/security/
23. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: Proc. 11th ACM SACMAT (2006)
24. Jaeger, T., Sailer, R., Zhang, X.: Analyzing Integrity Protection in the SELinux Example Policy. In: Proc. 12th USENIX-SS (2003)
25. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting past and present intrusions through vulnerability-specific predicates. In: SOSP. ACM (2005)
26. Kennell, R., Jamieson, L.H.: Establishing the genuinity of remote computer systems. In: USENIX Security Symposium (2003), http://portal.acm.org/citation.cfm?id=1251353.1251374
27. Klein, G., et al.: seL4: Formal Verification of an OS Kernel. In: SOSP '09 (2009)
28. Li, N., Mao, Z., Chen, H.: Usable Mandatory Integrity Protection for Operating Systems. In: Proc. IEEE SSP (2007)
29. Integrity: Linux Integrity Module(LIM), http://lwn.net/Articles/287790/
30. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor Support for Identifying Covertly Executing Binaries. In: Proc. 17th Usenix Security Symposium (2008)
31. Maruyama, H., Seliger, F., Nagaratnam, N., Ebringer, T., Munetoh, S., Yoshihama, S., Nakamura, T.: Trusted Platform on Demand. Tech. Rep. RT0564, IBM (2004)
32. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB Reduction and Attestation. In: Proc. IEEE SSP (2010), http://dx.doi.org/10.1109/SP.2010.17
33. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proc. 3rd ACM SIGOPS/EuroSys (2008)
34. Moyer, T., Butler, K., Schiffman, J., McDaniel, P., Jaeger, T.: Scalable Asynchronous Web Content Attestation. In: ACSAC '09 (2009)
35. Murray, D.G., Milos, G., Hand, S.: Improving xen security through disaggregation. In: VEE. VEE '08, ACM (2008)
36. Linux Kernel Backdoors And Their Detection. http://invisiblethings.org/papers/ITUnderground2004_Linux_kernel_backdoors.ppt
37. Security-enhanced linux, http://www.nsa.gov/selinux
38. OpenTC: OpenTC PET. http://www.opentc.net/publications/OpenTC_PET_prototype_documentation_v1.0.pdf
39. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping Trust in Commodity Computers. In: IEEE SP '10 (2010)
40. Payne, B.D., Carbone, M., Lee, W.: Secure and Flexible Monitoring of Virtual Machines. In: ACSAC (2007)

41. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: IEEE Symposium on Security and Privacy (May 2008)
42. Petroni, N.L., Timothy, J., Jesus, F., William, M., Arbaugh, A.: Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor. In: In Proc. 13th USENIX Security Symposium (2004)
43. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: USENIX Security Symposium (2004)
44. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards Trusted Cloud Computing. In: HOTCLOUD (2009)
45. Schiffman, J., Moyer, T., Shal, C., Jaeger, T., McDaniel, P.: Justifying integrity using a virtual machine verifier. In: Annual Computer Security Applications Conference. pp. 83 –92 (dec 2009)
46. Schiffman, J., Moyer, T., Jaeger, T., McDaniel, P.: Network-based Root of Trust for Installation. IEEE Security & Privacy (2011)
47. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: A Tiny Hypervisor To Provide Lifetime Kernel Code Integrity For Commodity Oses. In: Proceedings of twenty-first ACM SOSP (2007)
48. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying Code Integrity And Enforcing Untampered Code Execution On Legacy Systems. In: Proceedings Of The 20th ACM SOSP (2005)
49. Shankar, U., Jaeger, T., Sailer, R.: Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In: Proc. 2006 NDSS (2006)
50. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-vm monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on Computer and communications security (2009)
51. Shi, E., Perrig, A., van Doorn, L.: BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In: IEEE SP '05 (2005)
52. Sirer, E.G., de Bruijn, W., Reynolds, P., Shieh, A., Walsh, K., Williams, D., Schneider, F.B.: Logical attestation: an authorization architecture for trustworthy computing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 249–264. New York, NY, USA (2011), `http://doi.acm.org/10.1145/2043556.2043580`
53. Smalley, S., Vance, C., Salamon, W.: Implementing SELinux as a Linux Security Module. Tech. Rep. 01-043, NAI Labs (2001)
54. Smith, S.W.: Outbound Authentication for Programmable Secure Coprocessors. In: ESORICS (2002)
55. Sony: Update on playstation network and qriocity. `http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity` (April 2011)
56. Srinivasan, D., Wang, Z., Jiang, X., Xu, D.: Process out-grafting: an efficient ”out-of-vm” approach for fine-grained process execution monitoring. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 363–374. New York, NY, USA (2011), `http://doi.acm.org/10.1145/2046707.2046751`
57. St. Clair, L., Schiffman, J., Jaeger, T., McDaniel, P.: Establishing and Sustaining System Integrity via Root of Trust Installation. In: Annual Computer Security Applications Conference (2007)
58. Steinberg, U., Kauer, B.: Nova: a microhypervisor-based secure virtualization architecture. In: Proceedings of the 5th European conference on Computer systems. pp. 209–222. EuroSys '10, ACM, New York, NY, USA (2010)
59. Stumpf, F., Fuchs, A., Katzenbeisser, S., Eckert, C.: Improving the scalability of platform attestation. In: ACM Workshop on Scalable Trusted Computing (2008)
60. Sun, W., Sekar, R., Poothia, G., Karandikar, T.: Practical Proactive Integrity Preservation: A Basis for Malware Defense. In: Proc. 2008 IEEE SSP (2008)
61. Ta-Min, R., Litty, L., Lie, D.: Splitting interfaces: making trust between applications and operating systems configurable. In: OSDI. USENIX Association, Berkeley, CA, USA (2007)
62. TCG: Infrastructure Subject Key Attestation Evidence Extension Version 1.0, Revision 5. Tech report (2005)
63. TCG: Trusted Network Connect: Open Standards for Integrity-based Network Access Control. Technical report (2005), `http://www.trustedcomputinggroup.org`
64. TCG: Trusted Platform Module. `https://www.trustedcomputinggroup.org/specs/TPM/` (2005)
65. Trousers, `http://trousers.sourceforge.net/`
66. VMWare VMsafe. `www.vmware.com/go/vmsafe`