Designing Trustworthy Layered Attestations

Will Thomas
The University of Kansas
Lawrence, KS, United States

Logan Schmalz
The University of Kansas
Lawrence, KS, United States

Adam Petz
The University of Kansas
Lawrence, KS, United States

Perry Alexander
The University of Kansas
Lawrence, KS, United States

Paul D. Rowe
The MITRE Corporation
McLean, VA, United States

Joshua Guttman
Zeno's Arrow Consulting
Newton, MA, United States

James Carter
National Security Agency
Ft. Meade, MD, United States

Abstract

Attestation means providing evidence that a remote *target system* is worthy of trust for some sensitive interaction. Although attestation is already used in network access control, security management, and trusted execution environments, it typically concerns only a few of the system components on which the sensitive interaction would depend. A clever adversary might manipulate these shallow attestations to mislead the relying party.

Reliable attestations require *layering*. We construct attestations whose layers report evidence about successive components of the target system. Reliability also requires structuring the target system so only a limited set of components matters for any one sensitive interaction.

We show how to structure an example system for reliable attestations despite a well-defined, relatively strong adversary. It is based on widely available hardware, such as Trusted Platform Modules, and software, such as Linux with SELinux. We isolate our principles in a few maxims that guide system development. We provide a cogent analysis of our mechanisms against our adversary model, as well as an empirical appraisal of the resulting system. The performance burden of our attestation is negligible, circa 1.3%.

After our first example, we describe an alternate application level, and then an alternate underlying hardware anchor that uses confidential computing with AMD's SEV-SNP. The same maxims help us design for trustworthy attestations.

Keywords. Layered Attestation; Run-Time Attestation; Hardware-based attestation; Copland; Cross-Domain Solutions

ACM Reference Format:
Will Thomas, Logan Schmalz, Adam Petz, Perry Alexander, Paul D. Rowe, Joshua Guttman, and James Carter. 2026. Designing Trustworthy Layered Attestations . In *Proceedings of* . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Security mechanisms seek to prevent bad things from happening and reduce the damage done when they do occur. *Attestation mechanisms* play a complementary role and report the bad news to those who need to hear it. Inadequate security mechanisms allow an adversary to suppress the reporting of the bad news or prevent the attestation system from even noticing the bad thing being done. Inadequate attestation mechanisms can be fooled into thinking that a corrupted system is still ok. A game is being played. Can the adversary corrupt or replace system components, achieve its objective, and then repair or restore system components in such a way that the attestation system never reports any bad news? In this paper we aim to determine ways of building good attestation mechanisms, ensuring that they have the security infrastructure they need to be effective, and determining what bad things they will reliably report.

Attestation is about trust (and distrust). To deliver sensitive information to a system, we need to trust the system will not mishandle the information contrary to our intent. When acting on information from a system, we rely on it not to mislead us. Even if the system was carefully designed with security mechanisms to prevent such misbehavior, we know no security design is immune to malicious attacks. Malware can gain root user privileges, and can even hide itself from security tools. It can hijack core mechanisms embedded in the operating system kernel. There are, therefore, reasons to distrust even well-designed systems.

Attestation *discloses* rather than *defeats* adversary attacks that make the system untrustworthy. The purpose of an attestation is to provide the *relying party*—who will engage in a *sensitive operation* with the system—with *evidence* to justify trust in the current state of the system. That is, a skeptical *appraisal* of the evidence will hopefully justify the inference that no adversary *currently* has a foothold on the system that would allow it to interfere with the proper processing of our information. Hence, an attestation should inspect, or *measure*, aspects of the *runtime* state. The *appraiser* evaluating the evidence may be the relying party or an expert already trusted by the relying party for this purpose.

Runtime attestation poses a big problem for a cautious appraiser: Trusting the evidence in an attestation requires a trustworthy attestation infrastructure on the system itself to produce it. How can we gain trust in the attestation infrastructure itself? Appealing only to runtime evidence never grounds our trust in anything solid.

Instead, we need a reliable root of trust and a way to bootstrap trust from the root to other system components. When the adversary has software-only access, hardware mechanisms can provide the reliability we need. This suggests that runtime attestation needs to be *layered*, combining mechanisms from at least two layers (hardware and software). In fact, well designed systems include security mechanisms that allow the software to be further subdivided into more layers with differing levels of exposure to an adversary, yielding differing levels of skepticism about their susceptibility to attack.

Trustworthy attestation thus requires a delicate composition of security and attestation mechanisms at various system layers that must all cooperate (in subtle ways, as we will see) to generate a body of evidence robust enough to justify an appraiser's inference that the system is currently in a trustworthy state.

Although attestation and appraisal are already in use in many situations [5, 6, 7, 10, 22, 35], combining existing mechanisms in effective ways is difficult. Some mechanisms on their own are *shallow*, providing only a few types of information, for example, targeting userspace malware without addressing more advanced adversaries that could install a rootkit to hijack kernel mechanisms to avoid detection. Other mechanisms are *untimely*, in recording information only at boot, failing to detect runtime attacks that are possible, especially if the system may run for a long time between boot cycles. Moreover, attempts to be comprehensive and timely may end up being *fragile* by delivering, say, one opaque value to summarize the system state, making it unmanageable to appraise trust as system components are updated piecemeal.

This paper has two primary goals. First we aim to show that it is possible to design effective combinations of existing attestation and security mechanisms to achieve trustworthy layered runtime attestation. Secondly, we aim to share with the reader some mental tools we have identified that have helped us think critically about how to design and evaluate layered attestation systems.

In this paper we focus on a concrete example that helps to ensure we are grounded in realistic concerns. We use very standard hardware, such as Trusted Platform Modules, and software, such as commodity Linux with Security Enhanced Linux [17] and the Integrity Measurement Architecture [32]. The example application is a Cross-Domain Solution (CDS) that sits at the boundary between networks with different security postures, and rewrites and filters messages of specific kinds before they cross the boundary. We fully implemented security and attestation mechanisms for a CDS. Our work illustrates how to use adversarial reasoning to design the mechanisms and—in tandem—to identify a clear *adversary model* against which the attestations can succeed. The general considerations that drove the process are formulated in a set of five *design maxims* that help thinking critically about layered attestation systems.

To show they are helpful beyond one example, we consider a different application level context as well as a different hardware root of trust, namely confidential computing with a Trusted Execution Environment rather than the TPM. Our strategy in each of these two variants revolves around the same maxims.

Contributions. We make five contributions:

(1) We identify an adversary model capturing advanced and realistic threats against Linux system equipped with Security-Enhanced Linux and Linux Integrity Measurement Architecture. The adversary model was a byproduct of how we designed and evaluated this layered attestation system.
(2) A combination of empirical and formal evidence shows that our mechanisms win the attestation game for our CDS example by detecting successful corruptions from these threats.
(3) Five reusable maxims summarize the adversary-driven reasoning that led to our solution.
(4) Two major variant designs show how the same maxims help shape attestation solutions for them.
(5) We make two recommendations to strengthen existing kernels and architectures to eliminate two shortcomings we encountered in how we followed our own maxims. The very standard mechanisms we worked with imposed these shortcomings on us. We limited our adversary to rule out the (challenging) attacks that exploit these shortcomings.

We consider it an advantage that, although our maxims cannot be implemented fully in every architecture, they allow us to identify compromises that existing hardware, software, and performance may impose on us, and that they aid in identifying specific improvements as in contribution (5).

Section 3 is devoted to contributions (1) and (3), with Section 4 adding detail to the CDS design. Section 4.3 discusses recommendation 1, and 4.4 discusses recommendation 2. Section 5 handles the evaluation (2), while Section 6 moves away from the CDS example to apply the maxims to the variant applications (4).

2 Mechanisms and Standards

We start from a selection of widely available mechanisms for security and attestation. Trustworthy layered attestation requires judiciously combining existing technologies; success requires craftsmanship. In this section, we briefly summarize several core mechanisms and standards for attestation so that in subsequent sections it will be clear how each part contributes to create a trustworthy whole. Our choices do not aim to be comprehensive. They focus on technologies that figure into the designs we will explore.

Trusted Platform Modules. The TPM has two relevant capabilities. One is to store cryptographic keys and serve as a *root of trust for storage*. Second, a TPM has a collection of *Platform Configuration Registers* (PCRs). Some of the PCRs are reset only when the platform is powered down, and are otherwise modified only by having their values *extended* when its old value is hashed together with a new value. A platform can use PCRs to reflect the state of the system at the time it booted. This is a core hardware basis for trustworthy remote attestation, or a *root of trust for reporting*. TPM also have the concept of localities which are restrictions on the source of commands. Some are associated with specific hardware sources, such as early boot events, but others are flexible. These are called *extended localities*, i.e. localities ℓ with $32 \leq \ell < 256$.

Confidential Computing. A more recent alternative is to use TEEs like AMD's SEV-SNP or Intel TDX as a basis for attestation. We will discuss confidential computing as an alternative basis for layered attestation in Section 6.2.

Security-Enhanced Linux. SELinux adds fine-grained mandatory access control to the Linux kernel. The SElinux *policy* associates *security contexts* with executing processes as well as with system resources such as files, directories, sockets, and so on, and constrains interactions depending on the source and target security context. The SELinux policy is loaded during boot and will remain in effect until another policy is loaded. SELinux depends on the kernel and can be bypassed if the kernel is compromised.

Integrity Measurement Architecture. The Linux Integrity Measurement Architecture (IMA) is also a module loaded into the Linux kernel. The IMA policy specifies what files are to be measured and when. It then raises an error if the computed hash value differs from the expected value. IMA can ensure that an executable is launched as a processes only if its computed hash matches a given value. The IMA policy can only be extended after boot. IMA depends on the kernel and can be bypassed if the kernel is compromised.

Linux Kernel Integrity Measurement. In contrast to IMA, the Linux Kernel Integrity Measurement (LKIM) tool [19] is designed for the purpose of re-measuring the state of the *running* Linux kernel. It is the result of careful design to uncover the vast bulk of modifications that an adversary could use to maintain control in a Linux kernel.

Copland and its Maestro implementation. Layered attestations will require a variety of steps to be taken, some in specific orders, at one or more execution locations in a platform or distributed system. Copland is a language to specify layered attestation protocols. Maestro[1] provides tools for specifying Copland [26] attestation protocols, then synthesizing Attestation Manager configurations and executable procedures that both gather *and appraise* structured evidence.

The IETF Remote Attestation Procedures. RATS defines the fundamental roles of *attester*, *verifier*, and *relying party*, terminology that overlaps with ours but differs slightly from our *target of attestation* and *appraiser*. RATS also standardizes the syntactic structure of attestation evidence, but makes no attempt to express how to orchestrate attestations. While Maestro does not currently generate its evidence in the RATS-standardized form, it will become a fully RATS-compliant tool.

The challenge. Our central question in this paper is how to use the techniques described above to provide attestation data that are a trustworthy basis for appraisal. None of them can do it on their own. For instance, a virtual machine running Linux inside a TDX or SEV TEE is protected from other activities on its platform, but it is still vulnerable to data-driven attacks and rootkits [21].

3 Decomposing for Attestation

In this section we consider a particular example system to be defended against a broadly capable adversary model. The adversary capabilities show the security and attestation mechanisms that are needed and lead to a design outline, the details of which we flesh out in Section 4. While this section focuses on one target example, we identify five maxims for designing trustworthy layered attestations that we contend have broad applicability. In Section 6 we demonstrate how these maxims might be applied to the design and evaluation of alternative design patterns for different use cases.

3.1 Example System: A Cross-Domain Solution

Suppose that we have two networks with different security postures, and would like to deliver some messages, in a supervised way, from the more restricted network RN to the more open network ON. For this we could use a Cross-Domain Solution (CDS), meaning a computer with a network interface on each of the two networks. Fig. 1 illustrates the application structure. An intake server process listens on RN for connections, each of which will deliver a validly formatted email message to the CDS. The intake process makes the message available to a rewrite process, which may modify or sanitize some of its headers to make it suitable for delivery, discarding irreparable messages. A filter process receives the surviving messages, and, without altering them, makes a final determination on which are acceptable for delivery to ON. These messages are delivered to an export process, which connects through a socket on the ON interface to a server on ON for delivery. The rewrite and filter processes are designed for a good degree of generality, and each one has a configuration file that determines the message processing rules it will apply. The application processes use the directories Incoming, Rewritten, and Outgoing to pass messages to successive stages.

3.2 Security Goals

The CDS is a good example for our purposes, since it provides a straightforward security service to the networks. Namely, any message leaving by the export should have been

- input from RN via the intake process
- rewritten by the rewrite process using the authorized configuration file
- output by the filter process only if no further rewrites would be required by its authorized configuration file
- exported only when received from the filter process.

Thus, this is a *pipeline*: Messages should enter the pipeline only at the intake; a message should be delivered only if it reaches the export; no message should skip any pipeline stage; and each step is taken by a known process, controlled by an authorized configuration file [4].

This must be the *only* pipeline from RN to ON on the platform; below we will see how SELinux can help ensure that only intake has access to the platform's interface on RN and only export has access to the interface on ON.

3.3 Application Threats

The description of the application and its goals immediately makes some potential threats clear.

Pipeline threats. Can the adversary, by controlling processes on the platform, read or modify directories and files on our system?

(1) Messages may enter the Incoming, Rewritten, and Outgoing directories without traversing earlier processes.

[1] Measurement and Attestation Execution and Synthesis Toolkit for Remote Orchestration.

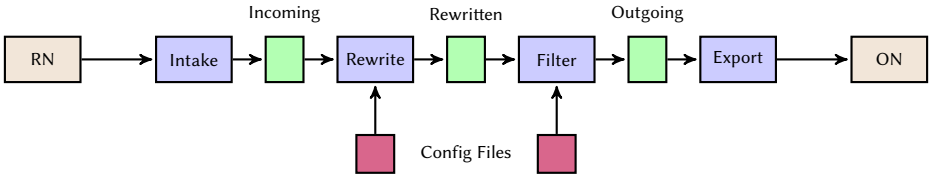

Figure 1: Cross-Domain System: Messages from Restricted Network to Open Network

(2) A configuration file may be modified to rewrite or filter a message via an unauthorized policy.
(3) An adversary may replace an executable file for a process, causing it to pass undesirable messages.

There are two strategies to respond. One is to engineer the whole system so we can be confident the adversary can never read or modify directories or files on our system. This strategy is intrinsically fragile [18].

The alternate strategy ensures that the properties we care about depend only on a small set of programs and functionality, and that a few robust mechanisms ensure other software on the system cannot damage the outcomes we are seeking. Applied to attestation, this means that the trustworthiness of an operation should depend only on a small number of mechanisms, each of which can be measured and the results assembled into a self-contained attestation.

Maxim 1. *Constrain possible interactions so attested properties depend only on limited, predictable, measurable parts of the system.*

A natural mechanism to leverage for this purpose is SELinux [20]. With a judicious use of security contexts applied to the pipeline directories, binaries, and configuration files, SELinux can enforce the pipeline property. The integrity of these executables can further be protected using IMA [33].

Thus, we need a way to attest to the fact that SELinux and IMA are running, and controlled by the intended policies. We also need to attest that the application-level processes and the underlying kernel will behave according to their expected behavioral semantics, and are not hijacked by the adversary. We turn next to these two concerns, taking the behavioral concern first, which is often undermined by data-driven corruptions.

Data-driven corruption threats. Crafted inputs cause unintended behavior in flawed programs. Attacks including stack-smashing and return-oriented programming [34] deliver data to change a program's control flow from the semantics of its source code.

A threat against a CDS would be a message crafted to corrupt memory, causing the process to handle subsequent messages wrongly. An adversary could cause a succession of sensitive messages to pass to the open network ON, disclosing contents that were not already under the control of the adversary. Memory-safe programming languages including Haskell, OCaml, and Rust [12, 15, 16] resist these attacks if the compiler is correct.[2] However, compilers may be imperfect, especially for security properties [39].

Another approach would be to inspect the memory of the long-running process periodically to ensure its control state is consistent with the control flow graph of the source program, and that its data structures match the expected invariants. This is *runtime process remeasurement.* The difficulty with this approach is that the expected invariants are not trivial to determine and must be found for each program that is to be inspected.

An approach that requires much less craftsmanship is to simply start a new process for each message. A malicious message may be able to cause unintended behavior in the process that handles it, but does not leave behind a corrupted process to misdeliver successive messages. This suggests a design in which the server forks a new intake process for each incoming connection from RN; if it never actually handles any message itself, it would thus be hard to degrade. The intake process could then invoke a rewrite process for the one message it has received. Similarly, instances of the filter and export processes could be started by their predecessor in the pipeline as the predecessor's last action. This would avoid contagion from one message to later messages.

The kernel itself cannot easily be restarted regularly, and can be the target of persistent attacks, namely rootkits. The importance of the kernel makes it worth the effort to determine the correct expected invariants of its state. LKIM, the Linux Kernel Integrity Measurement tool [19], is designed precisely for *Kernel runtime remeasurement.*

Maxim 2. *Long-lived services handling untrusted inputs need runtime remeasurement to recheck their state invariants. Short-lived, single-input processes avoid the need for runtime remeasurement.*

3.4 Safeguarding Attestation Mechanisms

Even if LKIM and the additional measurers determine that the platform is currently compliant, it is infeasible to determine whether a system has *always* been compliant or whether it has suffered from some kind of *transitory corruption.* Transitory corruption might have occured as a normal part of system maintenance which required an untrustworthy boot or it might have occured when an adversary briefly inserts a rootkit into the kernel. Either of these forms of corruption could give the adversary access to any secrets resident in kernel memory or the user level processes.

Disclosing a secret is a persistent result even if the cause is transitory. Thus we should avoid relying on memory resident secrets:

Maxim 3. *Attestation should be independent of secrets that could be disclosed by transient corruptions, lest transient corruptions cause persistent attestation failure.*

Signing keys are a crucial case, as attestation evidence must travel across a network to reach its appraiser. The signing key must be used by our attestation mechanism, but must be unavailable to other parts of the system to remain unavailable to the adversary. Maxim 3 says we cannot store this signing key in the kernel or in

[2] We chose to write our application programs in OCaml.

application-level processes handling attestation. A natural choice would be to maintain it in a TPM instead. We call this special key the *Attestation Signing Key ASK*.

A TPM can protect the confidentiality of *ASK* in the following way:

- The attestation signing key *ASK* is TPM resident in the sense that it exists in unwrapped form only in the TPM;
- It is exported from the TPM only in wrapped form, encrypted together with a signing policy that controls when the TPM will use it to generate signatures;

Although a TPM can protect the *ASK* from disclosure due to transient corruptions, this protection is not sufficient if an adversary can request signatures using this key, allowing it to forge attestations. This suggests another core principle. An appraiser never has a reason to accept purported evidence if that evidence could emerge from a system that might misuse signing keys, or for that matter generate the evidence to be signed by mere assertion rather than by actual measurement. Thus:

Maxim 4. *Each attestation must display traits showing that it originated from our uncorrupted attestation software, running under a trustworthy system boot.*

TPMs offer a relevant mechanism that we could leverage. Namely:

- The wrapped form of *ASK* exported by the TPM can also contain a signing policy that controls when the TPM will use it to generate signatures;
- The signing policy guarantees it can sign data only if TPM Platform Configuration Registers (PCRs) have values guaranteeing a desired outcome from the boot process.

Thus, any signature produced by *ASK* would guarantee that the system produced by the boot process includes SELinux and IMA, actively enforcing the intended policies. Its effectiveness depends on the appraiser having reason to believe the signing key cannot be misused outside the context of a trustworthy system boot and our intended software. This must rely on some trustworthy form of access control.

3.5 Layered Systems, Layered Attestations

Having worked our way downward from application to mechanisms providing security and attestation, consider now the upwards trajectory from measurements taken at startup. Initial hardware control starts with the `UEFI` and boot of the initial kernel and its `initramfs`, which can be recorded in the TPM which protects *ASK*. These deepest components need to be the oldest measurements in an attestation rooted in the TPM PCRs. Clearly, no further attestations can guarantee the `UEFI` and its firmware: This is where we run out of turtles [38]. Thus, our adversary model must exclude supply chain attacks or reflashing attacks against it.

For runtime remeasurement as for boot measurement, the *upwards* order is desirable [28]. We need the kernel to be uncompromised to believe the application measurements. A corrupted kernel could cause the application measurements to return the wrong results (e.g. by executing system calls wrong, or by corrupting application memory). It appears harder (and should take longer) for an adversary to corrupt the kernel after it is measured but before the upper layer measurement is taken than to repair a kernel it has already corrupted so that it will pass its measurement.

Hence, the dependency relation should determine a partial ordering on the measurement sequence. This suggests the maxim:

Maxim 5. *An acyclic dependency relation enables us to measure lower layers before components depending on them. Only very recent corruptions of underlying components can then undermine attestations.*

Generally, *very recent corruptions* mean corruptions during the course of attestation, or before the sensitive operation the attestation justifies.

We use the acyclic dependency maxim repeatedly in our CDS (see e.g. Section 4.1), but here too we faced a tradeoff given the current practical constraints (Section 4.3, 6.2).

3.6 Inferred Adversary Model

Against an all-powerful adversary, there is no hope of providing trustworthy attestations. However, our discussion allows us to define a very strong adversary model, against which we can still provide trustworthy attestations.

(1) The adversary is unable to modify the hardware of our platform—including the `UEFI` firmware that initiates boot—whether after deployment or via supply-chain modification.
(2) The adversary may log in to our system, even as root, and initiate processes including command shells. Thus, the adversary may remove, insert, and modify files in many parts of the filesystem, including files marked as executable.
(3) The adversary is constrained (even as root) by the SELinux policy. We assume that the boot-time SELinux policy is immutable and IMA policies can only be tightened during runtime.
(4) The adversary may boot the system into a state of its choice. However, the TPM PCR state will differ from our expected values throughout that boot cycle.
(5) The adversary may observe messages on the network and deliver messages of its choice to our system. So the adversary may engineer inputs to exercise any misbehavior of which our executables are capable.
(6) The adversary may corrupt a running Linux kernel, and also restore it to a non-corrupted state. The restoring may succeed in a short period of time.

A *short period of time* means the length of time between the beginning of an attestation and the completion of the sensitive operation the relying party will perform.

We distinguish a few finer types of adversary. If the corrupting can also, like the repair, occur in a short period, we call that a *fast* adversary. If an adversary is fast and able to choose when to do a fast corruption based on our attestation schedule, then we call it a *fast attestation-triggered* adversary.

(7) We assume the adversary is not a fast attestation-triggered adversary.

LKIM and the kernel (with its IMA component) form a *measurement cycle*(Section 3.5). LKIM measures the kernel, but IMA is used to ensure that the executable launched is actually LKIM. If the kernel is corrupted, IMA can be bypassed and the LKIM executables can

be modified to provide misinformation in an attestation. Thus, in Section 3.4 we also mentioned a limitation that required us to store a secret to authorize use of a TPM key. If an adversary compromises the kernel temporarily, they may be able to locate that secret and learn its value to use repeatedly (Section 3.4; cf. Sections 4.4, 6.2). An adversary that can craft unusual corruptions based on these two limitations in our adherence to our own maxims with the current mechanisms is *attestation attuned*.

(8) In the current architecture of our CDS example, we assume the adversary is not attestation attuned.

However, assumption (8) could be lifted if the architecture used virtualization to separate LKIM from the kernel in the attestation target, and if a different method is used to authenticate commands from the attestation infrastructure to the TPM (Sections 4.3–4.4).

Clause (7) is different: The advice in Maxim 5 would be pointless against fast attestation-triggered adversaries.

4 Attestation Structure via the Maxims

We now provide some additional detail on our CDS example system's security and attestation mechanisms, illustrating how the maxims and adversary model of Section 3 shaped them. There are four layers to this architecture:

(1) The boot-time, root-of-trust measurements;
(2) The kernel-level security and integrity mechanisms SELinux and IMA with their policies;
(3) The attestation manager (AM); and
(4) The attestation service providers (ASPs), such as LKIM, the application measurers, and the `tpm_sign` service.

Some additional detail is given in the appendices.

4.1 Booting from Roots of Trust: Linear dependencies

We rely on two *hardware roots of trust*, the Trusted Platform Module (TPM), which is the *root of trust for reporting and storage*, and the Unified Extensible Firmware Interface (`UEFI`), which is the *root of trust for measurement*. The `UEFI` starts the boot process and makes the first boot measurement, so it must be trusted, as reflected in our adversary model clause 1. The `UEFI` hashes the code of a small shim executable, together with a boot security configuration, and deposits them into PCRs 4 and 7 on the TPM. The shim is needed because it is signed by Microsoft, as standard `UEFI`s require; the shim's functionality is simply to hash the code of `GRUB`, the Linux boot loader, extending PCR 4 with this value, before invoking `GRUB`. `GRUB` in turn will hash the argument vector with which it will invoke its successor and deposit in into PCR 8. It hashes the contents of the initial file system in `initramfs` and deposits it into PCR 9 and then extends that PCR with a hash of the kernel it will invoke. After this it invokes an initial program in `initramfs`, which in turn executes a policy script `pol_scr`. This script hashes the system daemon code and deposits it into PCR 11 and then extends this PCR with a hash of the SELinux configuration and then the IMA configuration. Finally, `systemd` runs and installs SELinux and IMA with their configurations.

Each stage of this process depends on its predecessors. The successive PCR values tell us what code was in control at each stage and our inspection of that code justifies the claims we made in the previous paragraph about the behavior of each stage. The code during this boot phase executes without inputs from the outside world, meaning that data-driven corruption is not yet a threat. Thus, this phase provides a clean simple illustration of Maxim 5.

Table 1: PCR usage at start-up

Actor	Target PCR	Value
`UEFI`	4	`hash(shim)`
`UEFI`	7	`hash(sec_boot_cfg)`
`shim`	4	`hash(GRUB)`
`GRUB`	8	`hash(argv)`
`GRUB`	9	`hash(initramfs)`
`GRUB`	9	`hash(kernel)`
`pol_scr`	11	`hash(systemd)`
`pol_scr`	11	`hash(SELinux policy)`
`pol_scr`	11	`hash(IMA policy)`

This yields the pre-release configuration-compliant PCR values.

We "cap off" the PCRs when start-up is complete; see below, p. 7.

Actor	Checking	Target PCR	Value
`pol_scr`	4, 7, 8, 9, 11	11	`"ok"`

This yields the post-release configuration-compliant PCR values.

4.2 Kernel Security Policy: Creating separation

After the kernel is started up, it begins enforcing its SELinux and IMA policies.

Our IMA policy covers all of the executables that make up the CDS application, from intake through rewrite and filter to export and also the Attestation Manager and various Attestation Service Providers to ensure that these executables have not been modified when launched.

The SELinux policy protects the integrity of the attestation mechanisms and CDS system from everything except for the system management role. The policy for the attestation mechanisms ensures that the Attestation Service Providers (ASPs) can be invoked only by the Attestation Manager (AM); the results can be passed back only to the manager; and the manager passes the partial evidence only to subsequent services and completed evidence only to a socket for delivery to the appraiser. The policy for the CDS enforces the pipeline flow (Fig. 1). Each pipeline stage can `exec` only the next stage. Each pipeline process can read only from its intended source, and write only to its intended pipeline target, together with an additional directory for logging and errors. This condition also ensures that unexpected behavior in a pipeline process cannot lead to confidentiality failures that leak messages to other activities on the CDS platform.

Our IMA and SELinux policies provide strong kernel-level enforcement for the isolation that Maxim 1 requires. To justify the backbone of trust, the appraiser needs only a collection of evidence showing that (i) our SELinux policy is active, (ii) the processes with AM or CDS contexts are running the intended executables, and (iii) the measurements report acceptable values (which will include the kernel as well as IMA and SELinux policy measurements.)

4.3 Short-lived vs. Long-lived

We have designed the CDS application processes to be short-lived, in accordance with Maxim 2. The attestation mechanism is constructed similarly. A server listening on a socket forks a new AM process for each incoming attestation request. That server forks and execs ASPs as needed to perform measurements and also the signature requests for the TPM to apply the Attestation Signing Key *ASK* to each bundle of evidence as it becomes ready. Each of these processes executes one measurement or obtains one signature, returning its result to the AM and exiting.

By contrast, the Linux kernel is long-lived, and could not be replaced by a short-lived execution monitor in practice. Thus, by Maxim 2, a remeasurement technique is indicated. LKIM systematically re-evaluates data structures (including the system call table and other function pointers) and loaded code to ensure that the invariants on which normal kernel behavior depends are satisfied at the time of remeasurement.[3] In our demonstration case study, this is the only long-lived address space that we must inspect to revalidate invariants.

Except in special cases, the long-lived branch of Maxim 2 will rarely be preferred above the kernel level.

A tough choice. The current version of LKIM uses eBPF, the extended Berkeley Packet Filter, to obtain information from the running kernel from which it reconstructs the invariants. This makes it easy to deploy and means that it can be widely used on almost any installation of Linux. Misleading it requires a sophisticated rootkit that can identify which eBPF queries should receive false responses.

Older versions of LKIM made use of virtual machine inspection on the Xen hypervisor [36], [9, §5.1]. It ran in a virtual machine with read-only access to the target kernel running on a separate virtual machine. This had the advantage that even a sophisticated rootkit in the kernel under measurement could not tamper with the data LKIM received.

Following Maxim 5, we recommend an improvement to LKIM to make the original approach workable once again.

Recommendation 1. *A new version of LKIM should be able to run as a separate virtual machine on a widely available hypervisor, receiving kernel data via VM inspection.*

4.4 Providing Evidence

When the Attestation Manager has assembled its evidence, it needs to use the *ASK* to apply a digital signature so that the appraiser can infer it was correctly generated (Maxim 4). Signing must be unavailable unless the platform is currently executing under a boot complying with our intent, and in particular with our SELinux and IMA policies. Maxim 3 also stipulates that the *ASK* should not be resident in the AM's address space, since then a transient corruption would expose the *ASK* and allow the adversary to sign at will thereafter.

To meet this requirement, we arrange for the *ASK* to be a TPM signing key. It is usable only within the TPM, and outside the TPM it is always wrapped (i.e. encrypted). It is subject to a TPM key policy that stipulates it can be used only when the TPM PCRs 4, 7, 8, 9, and 11 contain the values resulting from a boot into the correct state. Updating our intended configuration on the platform requires revoking the old *ASK* or allowing its certificate to expire, and then provisioning a new key with the new key policy.

Crucially, the TPM must not use the *ASK* to sign evidence unless the AM (or its `tpm_sign` ASP, in fact) makes the request. A rogue process on the platform should never succeed when submitting a request for a value to be signed with this key. How can this be ensured?

Here there are two strategies.

Using TPM localities. The natural strategy would be to use the TPM localities. If we could control what process can use a locality and wrap a key such that it can only be used with a particular locality, then the following approach could be used:

(1) *ASK* is wrapped with a key policy stipulating that each signing command require a configuration-compliant PCR state P_A, and that the command should be delivered to the TPM labeled with a particular locality A.
(2) The kernel, when running our SELinux policy, labels commands with locality A only when the request is delivered by a process running in domain D_A.
(3) Our SELinux policy assigns domain D_A only to the executable at a particular path in the filesystem, `tpm_sign` in our case, which can be invoked only from a domain D_{AM} that only AM instances run within.
(4) Our IMA policy enforces that the `tpm_sign` and AM executables have the hashes determined for the correct signing program and Attestation Manager.
(5) The PCR state P_A incorporates our SELinux and IMA policies.

Unfortunately, this "natural approach" is hypothetical which leads to our second recommendation.

Recommendation 2. *The Linux kernel should have a standard way to deliver commands from user level processes to the TPM with specific extended localities* $32 \leq \ell < 256$. *SELinux should support this by implementing TPM localities as an object class, allowing configurations to determine which process security contexts may deliver commands with any given software locality.*

Key policies as secrets. There are two similar alternative strategies that use a secret that is either a key blob or the policy for using that key blob.

TPM key usage policies, which stipulate PCR values and localities that must be active for the key to be used, come in two forms. An *immediate* key usage policy is packaged with a TPM key such as *ASK* and always accompanies it, even when the key is wrapped and transmitted externally as a key blob. Alternatively, a key may have a *delegated* policy. In this case, a signature verification key or *delegation key* accompanies the TPM key when resident or wrapped in a blob. A *delegated key usage policy* is a signed data structure containing a usage policy. If it verifies under the delegation key, then the TPM will use the key under that policy.

For an immediate policy, we maintain the wrapped *ASK* with its immediate policy doubly wrapped with a second TPM-enforced encryption layer (via a LUKS container). The TPM policy for this

[3] The current version of LKIM is a product licensed to Invary https://www.invary.com; the exact set of invariants it enforces is not reported. LKIM appears to focus on the kernel data structures that determine control flow.

secondary encryption is subject to a key policy that stipulates that it can be decrypted only when the PCRs are in the "pre-release configuration-compliant" state, meaning that IMA and SELinux are running with our policies in force. The policy script `pol_scr` has the TPM unwrap the outer layer; `pol_scr` places the singly wrapped *ASK* key blob into the filesystem with a security context ensuring that only `tpm_sign` can read it and then extends the PCR 11 state with "ok" (ensuring a copy cannot be decrypted). See Table 1.

`tpm_sign` delivers this key blob in a TPM authenticated session along with each signing request. The TPM does not cache it. Thus, access to the key blob authenticates the requester as `tpm_sign`.

For the delegated key usage policies, we do not need to doubly wrap the actual key blob with its delegation key. Instead, we wrap the delegated key usage policy using a TPM key that can be used in the pre-release configuration-compliant state. This delegated key usage policy is placed in the filesystem for the exclusive use of `tpm_sign`. The delegated key usage policies ease system management, but come with a security risk: One must never use the same signing key to certify policies for two different keys that may have different purposes. Otherwise, the policy meant for one key could be misused with the other key.

Either version of this mechanism does roughly comply with Maxim 4 since the key blob will never become available in an untrustworthy boot cycle. In a trustworthy boot cycle it is readable only by `tpm_sign`. However, it does not follow Maxim 3: A transient corruption of the kernel would allow an adversary to either read the `tpm_sign` process memory that holds the key blob or directly read the file containing the key blob. Having obtained this value, the adversary could use the key blob at will. Hence, we regard this as a stopgap, and we believe that Maxim 3 provides a strong reason for implementing TPM localities in the Linux kernel and SELinux.

5 Implementation and Evaluation

How trustworthy is our attestation design? There are two different ways it could be untrustworthy. For one, the measurements we take could be unreliable. An adversary could cause a particular component to misbehave, and if our measurement does not reveal that, then we should not believe the attestation it is part of. Second, there could be structural problems in the way our attestation is put together. For instance, suppose we measured the rewrite configuration by hashing the file at the expected path, followed by a sequence of other measurements before finally using LKIM to check the kernel. An adversary who had corrupted the kernel could replace the good file at that location after the hash with a bad configuration followed by a fast kernel repair. This is compatible with the adversary model of Section 3.6, item (6). However, if the LKIM check happens before, then the kernel corruption would have to take place between this check and the configuration file measurement. This fast corruption would be challenging in reality, and item (7) records our assumption that corruption takes craftsmanship and therefore time.

We start in Section 5.1 with background on how we implement attestation. In Section 5.2 we consider the first type of failure, using testing to ensure that our individual measurements give good results. In Section 5.3 we consider the second, using a formal analysis to explore the attestation structure.

5.1 Prototype Implementation

The collection of software components that facilitate the transition from boot-time to run-time attestation is the *Attestation Manager Subsystem*. Among these is a server process called the *Attestation Manager Launcher* that listens for (run-time) requests and launches an *Attestation Manager* (AM) instance, which in turn orchestrates auxiliary executables called *Attestation Service Providers* or ASPs to gather and bundle runtime evidence. The Attestation Manager is instantiated using Maestro [24]. Maestro components on the CDS target platform orchestrate access to measurement primitives like the LKIM back-end and the TPM signing capability. Maestro is also able to generate *appraisal* components to run on an appraiser. These appraisal components unbundle the evidence that the *attestation* mechanism built, and evaluates their structure and the individual measurements they contain. These corresponding attestation and appraisal procedures ensure that all the evidence gathered is interpreted correctly by the appraiser to yield the right trust decision. Maestro-synthesized components adhere to the formal semantics of the Copland [26] domain specific language for attestation protocols, with accompanying proofs of correctness in the Rocq proof assistant.

Performance Benchmarking. Our prototype implementation is deployed on a Fedora 41 Linux desktop running an Intel Xeon E-2124 4-core CPU with 16GB RAM and a Samsung PM981 NVMe SSD. For ease of development we use the `swtpm` TPM emulator [3]. We also use the standard installations of IMA and SELinux packaged with Fedora 41. The only customizations to this commodity Fedora install are the hook scripts added to initramfs to measure IMA and SELinux policy configurations to TPM PCRs during boot. The CDS pipeline components are implemented in the memory-safe language OCaml.

We benchmark our system utilizing Phoronix Test Suite [25], a widely used benchmarking framework for Linux systems. We compare the performance of our system with and without the AM running, varying the attestation interval Δ (i.e., how often runtime attestation is performed) among 60s, 30s, and 15s. The benchmarks we run include: general OS tasks, building the Linux kernel from source, Nginx web server performance, and overall CDS processing throughput over a 10-minute period. The results are summarized in Table 3. The results indicate that the performance overhead introduced by the Attestation Manager is minimal across all benchmarks, even at the most frequent attestation interval of 15 seconds. In particular, the throughput of the security critical application (the CDS pipeline) remains high, with only a 1.3% reduction in processed requests when the AM is running with a 15-second attestation interval compared to the baseline without attestation.

We further experimentally establish an average time it takes for the attestation of the CDS system complete: 5.48 seconds. Of this, 5.05 seconds (or 92.2%) was dedicated to the LKIM measurement procedure. This was established by running the attestation procedure 100 times and averaging the total time taken.

5.2 Empirical Testing

In order for a relying party to trust the appraisal of the CDS system, they must also trust the *measurers* it uses to detect and report corruptions. In what follows, we provide empirical justification

that the measurers we use can detect typical attacks against target components. Further, we comment on why we expect other attacks to also be detected, when the components the measurers depend on are uncorrupted.

In Table 2, we summarize attacks against different layers of the CDS target system and mechanisms by which they are detected by our layered attestation system. The first column lists the major component classes, ascending from the deep Hardware mechanisms to the shallow User Space components. The other two columns indicate kinds of attacks and corresponding defenses for each component class.

Boot-time Attacks. The deepest level of integrity violations detected by our attestation framework occur during system boot. Attacks that target firmware are out of scope as `UEFI` is considered a root of trust according to item (1) of the inferred adversary model. The Boot Process component in Table 2 is shorthand for the configurable components involved in boot, including the `GRUB` bootloader, custom boot scripts, and their data dependencies. In testing, we used the measurement targets in the "Value" column of Table 1 to craft our example attacks. We first performed a "provisioned boot"—into an assumed secure state—to record golden values for each target. In subsequent boots, we modified each target in turn and validated that the key release scheme of Section 4.4 denied access to the ASK signing key because the PCR values were different.

Kernel-level Attacks. As a representative example of an attack against the Linux kernel, we used an out-of-the-box kernel module insertion attack [14]. Once it is loaded, the next LKIM measurement disclosed the corrupted kernel. As another example of a kernel-level attack, we replaced the installed CDS SELinux policy with a modified one while the kernel was corrupted. After this modification, the SELinux policy measurement ASP disclosed the alteration. If, however, a kernel corruption disables IMA, which we demonstrated can occur, then the LKIM executable or SELinux policy measurement ASP could be replaced with a dummy that would always return acceptable measurements. This is an example of an attack by an attestation attuned adversary, which we exclude from our adversary model in Section 3.6, item (8).

User-level Attacks. The CDS components, namely the pipeline executables and their configuration files, are prime targets for the adversary at the user-level layer of the system. Our layered attestation verifies the identity of these components using a binary hash equality.

We executed attacks that swap out each of the CDS user-level components in turn with a malicious version that violates CDS security goals. Our attestation protocol detected unrepaired changes to these components. Our IMA policy, with its integrity checks at process launch time, provides additional protection by preventing such modified programs from running altogether.

Attacks on the attestation mechanism. We explored various attacks on the attestation mechanisms themselves. First, we attempted to replace or modify the core AM and ASP executables that carry out the CDS attestation protocol. These attacks are prevented by IMA, just as the CDS user-level components are.

Next, by adversary model item 5, an adversary can deliver evidence to the appraiser that was not generated recently in the system's attestation machinery, or was not generated there at all.

An attacker can also store and replay the signed evidence package from a previous run of attestation on the target. However, in our attestation protocol the appraiser delivers the query with a freshly chosen nonce, which is embedded in the signed evidence package. Thus, the appraiser will reject a replayed evidence package, which contains a nonce that does not match its current expectation.

An adversary operating outside the system—or its attestation machinery—faces the task of signing evidence with the *ASK*, without which the appraiser will not accept it. This key is never usable except within the system's TPM and can be used only if accompanied by the wrapped key policy (Section 4.4), which is available (in the absence of a kernel corruption) only to the TPM signing ASP. Thus, within the scope of our adversary model, item (8), the adversary will not corrupt the kernel and also successfully retrieve the wrapped key policy. Thus, the forged evidence will not be signed by the *ASK*, and the appraiser will reject it. We did not attempt to execute these two attacks.

5.3 Formal Analysis

The formal analysis uses the techniques and tools of Rowe et al. [28, 29]. Our attestation protocol is written in the Copland language, whose formal semantics determines a partial order on the events such as measurements that the protocol requires. The analysis for a particular query systematically explores all ways the adversary could interpose corruption and repair events in this partial order subject to rules that reflect the adversary model and the structure of the system. It reports cases where the component mentioned in the query would be reported as acceptable when it was in fact corrupted. The analysis reflects dependencies, for instance the dependency of a user-level process on the kernel, because a user-level measurement process may report that its target is in an acceptable state when it is in fact corrupted, if the underlying kernel is corrupted when it runs.

We performed several analyses by running the tools on a model of our system design under varying sets of assumptions. This section summarizes the inputs and the results of those analyses at a high level.

Given a collection of measurement events temporally ordered in some specified way, a relying party of an attestation wants to know: Is it possible for some component—for example, the CDS code—to be compromised when it is measured without the appraiser receiving evidence of this (or any other) compromise? The analysis proceeds by enumerating all collections of adversary actions (subject to an adversary model described below) that would be necessary for the answer to that question to be "yes" [27, 29]. If the analysis completes without finding any way for the adversary to avoid detection, this is an indication that our design is resistant to the identified adversary model.

The attestation structure described in Section 4 specifies measurement events temporally ordered. For the analysis we consider the kernel with the IMA subsystem and its policy as one component. In the analysis we assume that the AM depends directly on the kernel to function properly. Finally, we assume that each of the

Components	Attacks	Defense
Hardware (CPU, TPM, etc.)	*Outside of Adv. Mod.* §3.6, item (1)	—
Boot Process	Corrupt boot scripts, IMA/SELinux policies	Signing key unavailability; TPM quote
LKIM	Corrupt LKIM binary	IMA policy (prevents launch)
	Corrupt kernel–LKIM cycle *Outside of Adv. Mod.* §3.6, item (8)	—[†]
Kernel (incl. IMA/SELinux)	Corrupt kernel, modules	LKIM appraisal
	Corrupt IMA or SELinux policies	Immutable (up to kernel corruption)
User Space (AM, ASPs, CDS)	Corrupt executables, configs	IMA/SELinux policies; application-level measurers
	Send malicious inputs	Short-lived processes

Table 2: Components, attacks on them, and defense mechanisms for our CDS system.
[†]In a virtualized setting (as described in Section 6.2), this cyclic limitation can be lifted by isolating LKIM via a hypervisor.

ASPs depends on the kernel because a corrupt kernel could alter memory in such a way to cause an ASP to measure a target of the adversary's choosing.

The default adversary model used in prior work [29] assumes the adversary can compromise any component (including the kernel) at any time. This is consistent with items (2), (4), and (6) of the adversary model in Section 3.6. Since compromised components might be detected by measurements, the adversary also sometimes has the incentive to *repair* a component returning it to an uncompromised state. These repair actions can also occur at any time as per item (6) of the adversary model. This default adversary model is stronger than the one in Section 3.6. As a result, we should expect an analysis using this adversary model to return possible attacks that defeat the attestation, and indeed it does.

The method of analysis allows us to specify patterns of attack to exclude from the search. By doing so, we can further constrain the adversary model to accurately align with Section 3.6. Based on (1) we exclude any corruption to the uefi. Similarly, since the IMA and SELinux policies are immutable at runtime (item (3) of the adversary model), we impose a constraint that any corruption of the IMA or SELinux policies must be preceded by a corruption of the kernel. In fact, our choice to model the IMA policy as part of the kernel already imposes this constraint for that policy. To account for item (7) which assumes it's not a fast attestation-triggered adversary, we exclude results containing a corruption of some component during boot or runtime. That is, we only consider what can happen if the adversary can corrupt components before boot or between boot and runtime. Each of these additional restrictions excludes some attacks found by the original analysis.

The added constraints above do not preclude an attestation attuned adversary. As a result, we would like any remaining attack to be one that exploits the acknowledged cycles between LKIM and the kernel (including the embedded IMA module and policy). The cycles are of two types. First, LKIM depends on the kernel to provide it information. So an uncorrupted LKIM could be lied to by a corrupted kernel. If we remove the assumed dependency of LKIM on the kernel, this will excluded attacks that exploit that dependency. Second, there is a measurement cycle in which the kernel (via IMA and its policy) measures LKIM and LKIM measures the kernel (including IMA and its policy). Exploiting this measurement cycle requires the adversary to corrupt *both* the kernel and LKIM between boot and runtime. By finally adding a constraint that at most one of them can be corrupted in that window, we effectively exclude an attestation attuned adversary. The resulting analysis reveals that no other attacks are possible against the system.

As a whole, the analyses described above demonstrate that our system design is able to withstand an adversary with the capabilities described in Section 3.6. They also highlight the role played by each of the restrictions imposed on the adversary (items (1), (3), (7), and (8) of the adversary model). Loosening any of those restrictions results in an adversary strong enough to defeat our attestation. While we believe the first three are quite reasonable restrictions, item (8) is only needed due to limitations in the available technologies that forced us to compromise on Maxim 5. In Section 6.2, we explore a possible alternative design that could potentially also resist an attestation attuned adversary.

6 Breadth of Applicability

We formulated our maxims while working on the CDS example, in response to the strategic choices we faced. But in writing them, we focused on aspects that would determine attestation trustworthiness across a range of cases. In this section, we will discuss two changes in the system context, showing that our maxims continue to shape our decisions effectively.

We have not implemented these alternate contexts, unlike the fully implemented CDS, so our discussion will be sketchier.

6.1 Distributed Document Control

Our CDS example is a single platform. Would we need essentially different strategies for attestation in a distributed application? For instance, consider a system for allowing controlled access to a collection of sensitive (possibly proprietary) documents. The documents normally reside on one or more platforms that serve them and function as the document store.

They may be checked out by a collection of workstations distributed throughout the organization's network. While a document

is checked out, the user of that workstation may read it or may make alterations to be checked back later. The sensitive documents must be under the care of specific, trustworthy application software while checked out. A specific workflow is needed on the workstations, assuring that each document is safely deleted when the user signals the end of the session using it. If the document is updated, there should be checks to ensure its contents are well labeled before committing it back to the document store.

The document store must ensure that it delivers documents only to authenticated and attested workstations for authenticated end users to use through the intended application. The trustworthy application on the workstation must also authenticate the document store and ensure it is trustworthy when retrieving documents (to ensure their integrity) and when delivering updates (to preserve their confidentiality). Mutual TLS is a mechanism to provide the reciprocal authentication we need, with suitably prepared certificates.

Maxim 1 suggests that each of the platforms—including the workstations and the document stores—has a separation between the software implementing the document system and the remainder of the platform. We would expect to use SELinux and IMA policies to ensure this in a way similar to the CDS.

In addition, in this context, there is an additional type of separation that matters, namely separating the appraisal from the primary platforms. In order to keep appraisal separated from all of the functionality of the system, we would run the appraiser as its own independent platform. The appraiser platform issues a periodic attestation request to each active platform, namely the workstations and document stores.

On successful completion, the appraiser, doubling as the Certifying Authority for the mTLS certificates, issues a short-term digital certificate for a new signing key pair generated during the attestation process. The signing key in this key pair will be protected via SELinux for the exclusive use of the network component of the application on the workstation. This is compatible with Maxim 3: If the signing key is disclosed in a transitory corruption, it can be misused only during the lifespan of the short-term certificate. After that, continued participation in the distributed application requires a new certified key, which will be based on attestation that uses a TPM-resident key. This TPM-resident key will be handled in accordance with Maxim 4, i.e. it will be available only during a compliant boot, much as we described in Section 4. Our recommendation (Section 4.4) that TPM localities be properly controlled via SELinux is still needed here.

Thus, the platforms need only interact via mTLS using the short-term certificates issued by the appraiser/CA. This ensures that the peer has recently passed an appraisal, at least assuming the systems have trustworthy access to the time. Time would have to be non-decreasing and loosely synchronized with the appraisal/CA platform.

Maxims 2 and 5 play much the same role as in our main example.

And who will appraise the appraiser? For this, manual scrutiny may suffice. For one thing, the appraiser, being also a CA, should really be used for no other purposes, this being a platform-wide application of Maxim 1. Its boot process can be constructed along the same lines as our main example, with the CA signing key made available only on a compliant boot. If the appraiser is rebooted regularly—a variant version of the Maxim 2 idea of short-lived computational processes—then an administrator can review its configuration on each reboot using a script to check the hashes of executables.

6.2 Confidential Computing as Hardware Root

In our main example, we used a widely available hardware basis in which `UEFI` initiates a boot sequence and records the hash of the program to which it passes control by extending a TPM PCR. Successive stages of the boot sequence record additional information in the PCRs, providing a context that controls whether the Attestation Signing Key *ASK* can be used to sign. This is a well-established strategy but it is lengthy, and it intertwines the attestation basis with many hardware-specific aspects of the boot process.

Confidential computing provides a more recent—and briefer—route to software attestation. In this approach, the early boot phases are not critical to attestation trustworthiness. Instead, after the hardware-specific aspects of boot start a hypervisor, the system starts a virtual machine within a Trusted Execution Environment. This TEE provides protection to the VM within it from the surrounding untrusted system, and also provides a root of trust for reporting. Intel's version is called Trusted Domain Extensions (TDX), and AMD has a Secure Encrypted Virtualization [1]. We will focus on AMD's Secure Nested Paging version of SEV.

AMD SEV-SNP uses trusted hardware and firmware (AMD Secure Processor or AMD-SP) to provide confidentiality and integrity for the runtime memory of a VM. Confidentiality is provided by encrypting all memory writes (and thus decrypting all memory reads) performed by the VM, while integrity is provided by implementing an access control mechanism that prevents the hypervisor or other VMs from writing to memory they do not own. It also further subdivides the VM's memory into up to four privilege levels. A virtual CPU acting at a higher privilege level within the VM may grant access to memory pages at lower privilege levels, choosing a subset of read, write, and execute permissions. The page tables for individual vCPUs are an additional control mechanism, i.e. an access succeeds only if it resolves under the page tables and is also authorized for the current privilege level.

When each VM is launched, SEV-SNP generates a *launch digest* of the launch detailing configuration options chosen, memory layout, and a hash of the contents of memory at launch. The manifest is held by the Secure Processor AMD-SP, which, when requested by the VM, can generate an *attestation report* that signs the launch digest together with a field that contains additional data chosen by the VM. When the attestation report's additional data contains a public key or its fingerprint, then the signed manifest effectively certifies that this VM controls the key. The key can then be the cryptographic basis for attestations as required by Maxim 4.

SEV-SNP does not seem amenable to access to a TPM. In a trust model where the underlying platform is controlled by a cloud provider, and may support multiple mutually suspicious tenants, the platform should not be trusted to ensure TPM communication would always accept commands from a VM that should be issuing that command, or would always deliver data (e.g. secret keys) to a VM that should receive it. It is uncertain whether SEV-SNP provides another mechanism for long-term storage of secrets, to be kept and

delivered again only when a new VM has a manifest agreeing with one that executed earlier. Thus, we regard each launch of a VM as a new principal, and its keys must be certified anew via the manifest mechanism before other parties treat it as trusted for a particular purpose.

The upper levels of our CDS (or document control platforms) can be replicated on SEV-SNP as a hardware root of trust. However, the maxims also suggest ways to use the new functionality to improve our design.

First, Maxim 5 motivates us to eliminate the cycle by which LKIM depends on the kernel while also measuring it. The SEV-SNP privilege levels appear to provide several possible strategies for this. We will describe one of them here.

LKIM should be able to read the Linux kernel memory, ensuring that a corrupted kernel cannot lie to LKIM. However, LKIM should not be able to modify the kernel memory, which might offer another route to corrupting the kernel. Conversely, the Linux kernel should not be able to modify LKIM's local memory, which could allow it, if rootkitted, to distort LKIM's report.

If the kernel runs at privilege level 2 and LKIM at the less privileged level 3, the kernel can have read-write access to its memory, while LKIM has only read access to this memory. The memory allocator at level 0 can use the page tables of the two vCPUs to ensure that the kernel has no access to LKIM's local memory. LKIM can thus reliably determine that the kernel's state respects its intended invariants, including that IMA remains active with the authorized policy. Confidential computing offers this as one solution to the cyclicity problem of Section 4.3, restoring the virtualization-based separations originally envisaged for LKIM [9, 36]. The AM and other ASPs might be located either with the target kernel or with LKIM depending on details of the scheme.

The TPM locality problem of Section 4.4 is also easier to solve with SEV-SNP. Our VM may set up a key management vCPU at privilege level 1—possibly a virtual TPM (vTPM)—ensuring that its address space is inaccessible to the target kernel and to LKIM. If LKIM and the AM communicate with it as a device with memory-mapped IO, then the normal SELinux mechanisms will allow controlling access to its command interface. This vCPU cannot preserve long-term secrets, but must establish keys each time the VM is started. Maxim 4 requires us to certify the new vTPM keys based on the key in the VM manifest. This scheme provides a good compliance with Maxim 3 by storing sensitive keys in areas well-separated from target kernel, which is the component most accessible to the adversary's potential corruptions.

Maxim 1 encourages us to separate the components we need to attest from the remainder of the system, and running the CDS functionality within a TEE-encapsulated VM provides a strong hardware-supported separation between these and other system components. SELinux is still needed with the CDS VM, because measuring its policy provides an attestation that the CDS pipeline is in place and protected from the rest of the VM. IMA continues to provide a reliable trust basis to ensure that the CDS executables are unaffected. Maxim 2 yields the same conclusions as in our original arrangement, because it assures us that data-driven corruptions cannot change the component behaviors for subsequent data items.

7 Related Work

Remote attestation is first presented by Haldar et al. [11] as introspection of virtual machines. Our work builds from Coker et al. [9] who define the "Principles of Remote Attestation", attestation manager, attestation protocols, and attestation service providers. Further extensions of these principles [2, 37] also provided insights into how our design and maxims should be structured. Carpent et al. [7] introduced the notion of a "Quality of Attestation Metric" that we have not directly utilized for our analysis, but provides an interesting avenue for future work.

Rowe [28, 30] introduces layered attestation and evidence bundling as a mechanism for building trust from context. Layered attestations use multiple attestation managers and attestation protocols to assemble attestations of dependencies into evidence bundles with the attestation target. The structure of evidence bundles captures semantics of the attestation process. Bundling and layering are central concepts in the attestation system described here.

Of particular note is the connection between attestation and hardware. Our work focuses on *hardware-based* attestation where the root of trust is a hardware component we presume to be uncorruptible. We leverage the TPM and boot process utilizing techniques presented by Sailer et al. [32].

Copland [26] allows for a diverse set of attestation techniques to be utilized with a single common interface. In particular, we utilize *binary-based* attestation techniques [5, 10, 13] to verify that CDS component binaries are the exact binaries that were expected. Further, we make use of *property-based* attestations [8, 19, 31] as a means for verifying the correctness of components without exact binary equality being determined. Our utilization of the LKIM tool is a notable example of this *property-based* attestation technique.

8 Conclusion

Designing effective layered attestation systems is difficult. One class of challenges are definitional—stating clearly what components and capabilities comprise a target system and its adversary. In this work we codify a detailed and realistic adversary model for commodity Linux systems equipped with SELinux and IMA. This adversary model is derived from a top-down security analysis of a Cross Domain Solution application. The remaining challenges are in choosing and *composing* available attestation and security mechanisms to produce trustworthy attestations of target behavior in the presence of the adversary. Towards this goal, we motivate a collection of five maxims that guide co-design of a target system and its attestation mechanisms. The maxims were developed and refined in parallel with adversarial reasoning applied to threats against the CDS example system. To demonstrate subtleties of the maxims, we walk through a bottom-up design and prototype implementation of the CDS example system that leverages the maxims (when possible) at key decision points in the design.

An empirical analysis of the CDS example system showed that our attestation mechanisms detected realistic attacks derived from our adversary model with negligible performance impact. Because the maxims guided the design of the CDS system, our positive analysis results support the efficacy of the maxims. A complementary formal analysis also gives confidence that we didn't overlook any class of attacks within our chosen adversary model. We also show

the broader applicability of the maxims by outlining how they influence the design of two additional example systems with distinct layered architectures.

Finally, we propose two actionable recommendations for changes to the linux kernel and system architectures to enable future designs that fully adhere to our maxims. The first is to add virtualizaton support for LKIM to eliminate the cycle between LKIM and the kernel (Recommendation 1). The second is to add support for TPM localities in the Linux kernel to fully protect against transient corruptions of the kernel that may lead to permanent misuse of the TPM signing key (Recommendation 2). Realizations of these two recommendations would allow us to strengthen our adversary model and support trustworthy layered attestations for an even wider collection of applications and architectures.

References

[1] Advanced Micro Devices, Inc.: SEV-SNP: Strengthening vm isolation with integrity protection and more (2020), https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more

[2] Banks, A.S., Kisiel, M., Korsholm, P.: Remote attestation: A literature review (2021), https://arxiv.org/pdf/2105.02466

[3] Berger, S.: swtpm: Libtpms-based TPM emulator (2025), https://github.com/stefanberger/swtpm

[4] Boebert, W.E., Kain, R.Y.: A practical alternative to hierarchical integrity policies. In: Proceedings of the Eighth National Computer Security Conference (Oct 1985)

[5] Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 132–145. ACM (2004)

[6] Carpent, X., ElDefrawy, K., Rattanavipanon, N., Tsudik, G.: Lightweight swarm attestation: A tale of two lisa-s. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 86–100 (2017)

[7] Carpent, X., Tsudik, G., Rattanavipanon, N.: Erasmus: Efficient remote attestation via self-measurement for unattended settings. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1191–1194. IEEE (2018)

[8] Chen, L., Landfermann, R., Löhr, H., Rohe, M., Sadeghi, A.R., Stüble, C.: A protocol for property-based attestation. In: Proceedings of the First ACM Workshop on Scalable Trusted Computing. pp. 7–16. STC '06, Association for Computing Machinery, New York, NY, USA (2006), https://doi.org/10.1145/1179474.1179479

[9] Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. International Journal of Information Security 10(2), 63–81 (June 2011)

[10] Gu, L., Ding, X., Deng, R.H., Xie, B., Mei, H.: Remote attestation on program execution. In: Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing. pp. 11–20. STC '08, Association for Computing Machinery, New York, NY, USA (2008), https://doi.org/10.1145/1456455.1456458

[11] Haldar, V., Chandra, D., Franz, M.: Semantic remote attestation – a virtual machine directed approach to trusted computing. In: Proceedings of the Third Virtual Machine Research and Technology Symposium. San Jose, CA (May 2004)

[12] Hutton, G.: Programming in haskell. Cambridge University Press (2016)

[13] IETF RATS Working Group: Remote ATtestation ProcedureS (RATS). https://datatracker.ietf.org/wg/rats/about/ (2023)

[14] Invary Runtime Integrity: Invary Test Kit. https://github.com/Invary-Runtime-Integrity/invary-test-kit (2025), a Linux kernel module that hijacks the kill system call to test Invary's Runtime Integrity sensor software.

[15] Klabnik, S., Nichols, C.: The Rust programming language. No Starch Press (2023)

[16] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Sivaramakrishnan, K., Vouillon, J.: The OCaml system release 5.3: Documentation and user's manual. Inria (2025), https://ocaml.org/manual/5.3/index.html

[17] Loscocco, P., Smalley, S.: Meeting critical security objectives with security-enhanced linux. In: Proceedings of the 2001 Ottawa Linux Symposium (Jul 2001)

[18] Loscocco, P.A., Smalley, S.D., Muckelbauer, P.A., Taylor, R.C., Turner, S.J., Farrell, J.F.: The inevitability of failure: The flawed assumption of security in modern computing environments. In: In Proceedings of the 21st National Information Systems Security Conference. pp. 303–314 (1998)

[19] Loscocco, P.A., Wilson, P.W., Pendergrass, J.A., McDonell, C.D.: Linux kernel integrity measurement using contextual inspection. In: Proceedings of the 2007 ACM workshop on Scalable trusted computing. pp. 21–29. STC '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1314354.1314362

[20] Mayer, F., MacMillan, K., Caplan, D.: SELinux by Example. Prentice Hall (2007)

[21] Niu, Y., Shi, J., Han, R., Liu, Y., Ma, C., Lyu, Y., Lo, D.: What you trust is insecure: Demystifying how developers (mis)use trusted execution environments in practice (2026), https://arxiv.org/abs/2512.17363

[22] Nunes, I.D.O., Eldefrawy, K., Rattanavipanon, N., Steiner, M., Tsudik, G.: Vrased: A verified hardware/software co-design for remote attestation. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1429–1446 (2019)

[23] Petz, A., Alexander, P.: An infrastructure for faithful execution of remote attestation protocols. Innovations in Systems and Software Engineering (2022)

[24] Petz, A., Thomas, W., Fritz, A., Barclay, T.J., Schmalz, L., Alexander, P.: Verified configuration and deployment of layered attestation managers. In: Software Engineering and Formal Methods: 22nd International Conference, SEFM 2024, Aveiro, Portugal, November 6-8, 2024, Proceedings. pp. 290–308. Springer-Verlag, Berlin, Heidelberg (2024)

[25] Phoronix Media: Phoronix test suite. https://www.phoronix-test-suite.com/ (2026), gitHub repository: https://github.com/phoronix-test-suite/phoronix-test-suite

[26] Ramsdell, J., Rowe, P.D., Alexander, P., Helble, S., Loscocco, P., Pendergrass, J.A., Petz, A.: Orchestrating layered attestations. In: Principles of Security and Trust (POST'19). Prague, Czech Republic (April 8-11 2019)

[27] Ramsdell, J.: Chase: A model finder for finitary geometric logic. https://github.com/ramsdell/chase (2020)

[28] Rowe, P.D.: Confining adversary actions via measurement. Third International Workshop on Graphical Models for Security pp. 150–166 (2016)
[29] Rowe, P., Ramsdell, J., Kretz, I.: Automated trust analysis of copland specifications for layered attestations. In: Principles and Practice of Declarative Programming (PPDP 21) (Sep 2021)
[30] Rowe, P.D.: Bundling Evidence for Layered Attestation. In: Trust and Trustworthy Computing, pp. 119–139. Springer International Publishing, Cham (Aug 2016)
[31] Sadeghi, A., Stüble, C.: Property-based attestation for computing platforms: caring about properties, not mechanisms. In: Proceedings of the 2004 workshop on New security paradigms. pp. 67–77. ACM (2004)
[32] Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementatation of a tcg-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium. USENIX Association, Berkeley, CA (2004)
[33] Sailer, R., Zhang, X., Jaeger, T., Doorn, L.V.: Design and implementation of a tcg-based integrity measurement architecture. In: USENIX Security symposium. vol. 13, pp. 223–238 (2004)
[34] Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security CCS. pp. 552–561. ACM (2007), https://doi.org/10.1145/1315245.1315313
[35] Sultana, N., Shands, D., Yegneswaran, V.: A case for remote attestation in programmable dataplanes. In: Proceedings of the 21st ACM Workshop on Hot Topics in Networks, HotNets 2022, Austin, Texas, November 14-15, 2022. pp. 122–129. ACM (2022), https://doi.org/10.1145/3563766.3564100
[36] Thober, M., Pendergrass, J.A., McDonell, C.D.: Improving coherency of runtime integrity measurement. In: ACM workshop on Scalable Trusted Computing. pp. 51–60 (2008)
[37] Usman, A.B., Cole, N., Asplund, M., Boeira, F., Vestlund, C.: Remote attestation assurance arguments for trusted execution environments. In: Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems. pp. 33–42. SaT-CPS '23, Association for Computing Machinery, New York, NY, USA (2023), https://doi.org/10.1145/3579988.3585056
[38] Wikipedia: Turtles all the way down. https://en.wikipedia.org/wiki/Turtles_all_the_way_down (Accessed Jan 2026)
[39] Xu, J., Lu, K., Du, Z., Ding, Z., Li, L., Wu, Q., Payer, M., Mao, B.: Silent bugs matter: A study of compiler-introduced security bugs. In: USENIX Security Symposium. pp. 3655–3672 (2023)

A Performance Benchmarks Table

In Table 3 (page 15) we summarize the results of our benchmarks.

Table 3: Performance Benchmarks: Baseline vs. AM Running (varying Δ)

Metric	Baseline	AM (Δ = 60s)	AM (Δ = 30s)	AM (Δ = 15s)	% Reduction
CDS Throughput (over 600s period)					
Total Requests Processed	84060	83328	83040	82956	1.3%
OSBench (Lower Better)					
Create Threads (μs)	19.18	18.96	18.81	19.03	-0.8%
Launch Programs (μs)	120.25	122.43	121.49	122.01	1.4%
Create Processes (μs)	32.85	33.38	33.47	34.31	4.3%
Memory Allocations (ns)	87.62	88.21	88.62	87.78	0.2%
System Compilation (Lower Better)					
Linux Kernel 6.15 (s)	775.51	777.73	786.70	793.59	2.3%
Server Performance (Higher Better)					
nginx 1.23.2 (req/s)	9845.89	9828.28	9774.56	9651.66	0.7%