Solving Attestation Fragility: A Software-Based Trust Anchor for Agile Confidential VMs

Julian Funk[1], Matti Schulze[1], Patrick Sieber[1], Manuel Vögele[2], Jonas Röckl[1], and Tilo Müller[3]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg
{julian.funk, matti.schulze, patrick.sieber, jonas.roeckl}@fau.de
[2] Ruhr-University Bochum
manuel.voegele@rub.de
[3] Hof University of Applied Sciences
tilo.mueller@hof-university.de

Abstract. Remote attestation is a cornerstone of trust in Confidential Computing, leveraging measurements to provide cryptographic proof of a workload's integrity within Trusted Execution Environments. However, in production environments, such as confidential VMs hosted by AWS, Azure, or Google Cloud Platform based on Intel TDX or AMD SEV-SNP, current deployments often suffer from *Attestation Fragility*: As the hardware-based measurements usually include the entire operating system kernel, every kernel update alters the measurement hash. This creates a security paradox where frequent updates, though essential for a robust security posture, change the measurements and thus break the trust relationship with the relying party, necessitating cumbersome manual policy updates.
To this end, we present ANCHOR, an architecture that decouples hardware-rooted measurements from the operating system. By utilizing a multi-stage boot process powered by `kexec`, ANCHOR introduces a minimal, static Trust Anchor which loads a runtime kernel after successful attestation. This Trust Anchor remains unchanged over long periods, providing a consistent hardware measurement. By encrypting the runtime kernel and cryptographically binding its integrity to the hardware-attested Trust Anchor, ANCHOR enables VM lifecycle management without compromising security guarantees. We implement ANCHOR for Intel TDX and demonstrate its operational flexibility for cloud deployments.

Keywords: Confidential Computing · Remote Attestation · Cloud Security · Trusted Execution Environments

1 Introduction

In cloud computing, cloud providers offer tenants their hardware, often in the form of Virtual Machines (VMs), which are managed by a hypervisor. However, since a cloud provider can introspect any workload executed and data stored on its servers, the widespread adoption raises privacy concerns, particularly when

sensitive or confidential information is processed [49, 32, 1]. To address these concerns, the Confidential Computing (CC) paradigm has emerged, providing a hardware-backed Trusted Execution Environment (TEE) which protects *data in use*, i.e., data that is being actively processed, thus making it inaccessible even to the cloud provider [39, 31]. Most major cloud providers like Amazon Web Services (AWS), Google Cloud Platform (GCP), or Azure offer CC solutions to their end-users [15]. Recent CC designs enable a lift-and-shift approach, allowing entire VMs to be spawned inside a TEE. Such VMs are also referred to as confidential VMs (cVMs). CC leverages *remote attestation* to enable a relying party to verify the integrity of the cVM in the TEE as well as the underlying platform.

Remote attestation relies on comparing a hardware-rooted measurement of the cVM against a pre-computed golden hash. For this mechanism to be effective, the relying party typically must possess an *exact bit-for-bit* representation of the cVM's software stack (Firmware, kernel, and initial filesystem). If any part of the software stack changes, e.g., due to an update, the relying party needs to re-calculate the golden hashes. However, achieving such binary identity in modern cloud environments is fraught with technical challenges:

To maintain robust security, Linux kernels in modern distributions such as Ubuntu or RHEL require frequent patching. For instance, the Long Term Support (LTS) kernel 5.4, which was published in November 2019 with End of Life (EOL) in December 2025, received 302 updates over its lifetime, resulting in an update every week on average. Current LTS kernels like 6.18 and 6.12 show a similar update frequency. Measuring the guest Operating System (OS) kernel for remote attestation combined with frequent updates creates a paradox: To remain secure, a system must be patched frequently; but in a CC environment, every patch changes the measurement hash, thereby breaking the trust relationship with the relying party. We refer to this as *Attestation Fragility*.

Additionally, even in the open-source ecosystem, reconstructing a binary image that is identical to the one deployed in the cloud is notoriously difficult. Initiatives like the *Reproducible Builds* project [37, 25] report that while progress has been made, the Linux kernel remains one of the most complex components to reproduce deterministically. Binary discrepancies are triggered by environmental differences, such as toolchain version mismatches, metadata effects (non-deterministic elements such as embedded timestamps), or configuration options in general. To use remote attestation meaningfully, this lack of binary reproducibility for complex OS kernels forces relying parties to blindly trust opaque, pre-compiled binaries.

Especially for users that operate fleets of thousands of cVMs, with different kernel configurations across multiple cloud providers, Attestation Fragility combined with the complexities of reproducible builds is a significant burden to adopt CC in practice.

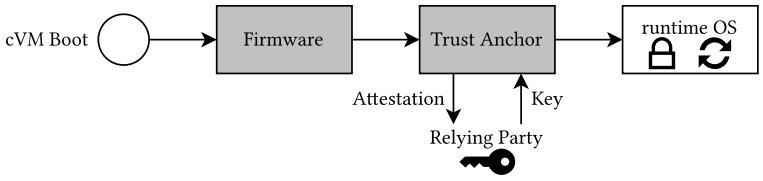

Fig. 1. Boot process of a cVM with ANCHOR. Components that are included in the attestation measurements are displayed in gray. The runtime OS receives frequent updates, but is not included in the attestation measurement. The relying party provisions the key to decrypt the runtime OS to the Trust Anchor only after successful attestation.

1.1 Contributions

To overcome Attestation Fragility and the complexities of reproducible builds, we propose ANCHOR to shift the measurement boundary away from the volatile runtime OS kernel towards a persistent and minimal *Trust Anchor*. We argue that shifting the attestation boundary from a volatile runtime kernel to a minimal Trust Anchor is not a security degradation, since hardware attestation also only captures the software stack at the exact moment of the boot measurement. ANCHOR utilizes a multi-stage boot process powered by `kexec` [21] to build up a chain of trust, originating in the hardware-attested Trust Anchor. The Trust Anchor acts as a gatekeeper and as loader for the OS kernel used during runtime (Figure 1). By decoupling the attestation from the runtime OS, ANCHOR even allows the runtime kernel to be *arbitrarily complex*, e.g., non-reproducibly buildable, without increasing the operational hurdle.

Once the hardware attests that the Trust Anchor is untampered, it establishes a secure channel to a relying party to receive a disk-decryption key. This key is then used to decrypt the actual workload of the cVM, the runtime kernel and root filesystem. Because the runtime kernel is encrypted and managed by ANCHOR, its integrity and confidentiality are cryptographically bound to a successful attestation.

In summary, we make the following contributions:

- We design ANCHOR, a system architecture to solve the attestation fragility problem.
- We implement ANCHOR on Intel TDX and conduct a performance evaluation to assess the boot time overhead of ANCHOR and show its real-world applicability. We show that this is possible only with local modification of the cVM image.
- With ANCHOR, we extend the data in use protection of the guest's OS kernel to data at rest.
- We discuss the security implications of ANCHOR.
- We release ANCHOR's source code as open-source [4].

[4] `https://github.com/ANCHOR-TDX/ANCHOR`

1.2 Outline

Section 2 gives an overview of Intel Trust Domain Extensions (TDX). In Section 3, we present the design of ANCHOR, followed by a performance evaluation in Section 4. We discuss security in Section 5 and present limitations and future work in Section 6. Section 7 deals with related work before we conclude in Section 8.

2 Background

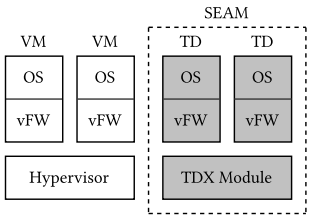

Fig. 2. Intel TDX system overview. Parts that are protected by Intel TDX and run in the SEAM mode are marked gray. Normal VMs are managed by the hypervisor, TDs by the TDX module. VMs and TDs consist of virtual firmware and OS.

This section provides the necessary background on Intel TDX, which ANCHOR targets in its current implementation, including a system overview, remote attestation details, and a threat model.

Intel TDX Overview. Intel TDX is an architectural extension for Intel Xeon processors to enable VM-based CC [9]. Figure 2 shows the general structure of Intel TDX. TDX introduces a new CPU mode, the Secure Arbitration Mode (SEAM) mode, which can be used to deploy cVMs, so-called Trust Domains (TDs). The hypervisor cannot access resources assigned to the SEAM-mode. In addition to the newly introduced TDs, the Intel TDX Module also runs in the SEAM mode. To protect the confidentiality and integrity of TDs from the hypervisor, CPU-level isolation and memory encryption are used. If the hypervisor wants to interact with a TD during runtime, e.g., for scheduling, it needs to request the TDX Module to do so. Inside the TD, besides the kernel, a virtual firmware (vFW), e.g., OVMF, runs. On commercial platforms, the end-user usually has no control over the vFW [15]. Intel TDX provides a remote attestation scheme, which is detailed in the next paragraph.

Remote Attestation. Intel TDX offers a remote attestation scheme to enable a remote party to verify the integrity of a TD [40]. The attestation report includes various hashes of the Trusted Computing Base (TCB), including a measurement of the TDX Module. Besides that, multiple components of the TD are measured and included in the attestation report. Typically, these are the vFW

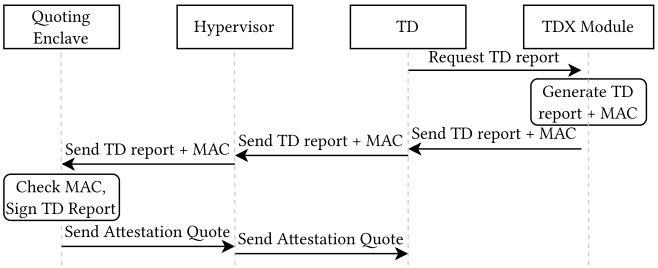

Fig. 3. Generation of an attestation quote with Intel TDX.

and its configuration, the OS kernel, the initramfs, and boot parameters. In addition to the measurements, user-provided data, e.g., a nonce, can be included in the attestation report. The generation of an attestation quote is illustrated in Figure 3. Upon request from the TD, the TDX Module generates an attestation report, which is protected using a Message Authentication Code (MAC). The report is sent to the hypervisor, which invokes an SGX enclave, the so-called quoting enclave. This enclave signs the attestation request if the MAC is valid and sends it back to the hypervisor, which forwards it to the TD. The signed attestation report forms the attestation quote. Because of this attestation scheme, the hypervisor can access the attestation quote before it reaches the TD. The quote can be sent to a remote party, which subsequently can reason about the integrity and decide if the TD is running on genuine TDX hardware, before supplying confidential data to the TD. To perform this step, the verifier needs to calculate the exact same measurements, which can be difficult to achieve.

Threat Model. We require that the Intel TDX hardware and firmware work as specified, including the remote attestation scheme. Likewise, we assume that the cryptographic primitives and protocols, e.g., TLS and Linux Unified Key Setup (LUKS), that we base ANCHOR on, are secure. We do not consider denial-of-service attacks against the TD, following common practice in CC-based systems. Further, we assume that the implementation of the Trust Anchor is secure and does not introduce bugs. In line with the standard Intel TDX threat model [9, 29], we assume that the hypervisor is untrusted and malicious. The hypervisor can access all data that is stored on the host-level persistent memory as well as intercept and modify all network traffic. The attacker can tamper with the boot configuration, e.g., start the TD with debug configuration, or modify and read the image of a TD before booting it. We assume that the attacker's ultimate goal is to break the integrity and confidentiality of the TD's data through accessing it on persistent storage.

3 ANCHOR

In this section, we first define the requirements for ANCHOR, followed by an exploration of the solution space for implementing ANCHOR and the used cryp-

tographic primitives to protect the data at rest. After this, we provide a detailed presentation of the protocol for receiving the disk encryption key over the network and the partition layout. Finally, we sketch the design of the relying party.

3.1 Requirements

Based on insights of related work [15], we formulate requirements, such that ANCHOR is applicable in real-world cloud scenarios.

Recent research has shown that market-leading cloud providers often retain control over critical parts of the software stack inside cVMs. Focusing on the major cloud providers, i.e., Azure, AWS, and GCP, we observe that each cloud provider uses their own, custom vFW implementation that runs alongside the guest OS inside the TD. Although Microsoft Azure has decided to transition to the open-source firmware openHCL [34], it is currently not possible to bit perfectly reproduce the build process [45, 16]. On the other hand, while AWS offers reproducible firmware [2], it does not offer Intel TDX based cVMs. In contrast, GCP offers Intel TDX, but the OVMF-based firmware is closed-source [13]. None of the providers allows to provide custom firmware for cVMs at the time of the writing. Therefore, we conclude that ANCHOR must not rely on firmware modifications (**R-1**).

Further, we require ANCHOR to provide for a stable attestation verification process. Thus, when the runtime kernel of a TD gets updated, the measurements in the attestation quote must not change (**R-2**). Without this property, large scale CC deployments face costly policy updates on the verification side, as updates should be applied frequently.

As we follow the common threat model for CC, which assumes that the hypervisor might be compromised and malicious, ANCHOR needs to ensure that the kernel image was not tampered with or rolled back by the hypervisor. Thus, only untampered kernel images should be allowed to be launched by ANCHOR (**R-3**).

While the focus of CC in general is on the integrity and confidentiality of data in use, data at rest is by default exposed unencrypted to the hypervisor. This means, every time the TD gets restarted, the hypervisor can extract secrets or Intellectual Property (IP). We argue that for a real-world applicable design, ANCHOR should provide options to protect the confidentiality at rest. While past research [48, 7, 19, 11] focused on the root filesystem, we also argue that the runtime OS kernel must be kept confidential at rest. Kernels may include proprietary drivers, performance optimizations, or other modifications containing sensitive IP that must be protected from a potentially compromised hypervisor. As a result, we argue that ANCHOR should be capable of ensuring the confidentiality of the data at rest of the TD's kernel and filesystem (**R-4**).

ANCHOR should only induce a minimal performance overhead in regard to boot time and runtime in order to stay applicable to real-world workloads (**R-5**).

In summary, we have the following requirements for ANCHOR:

R-1 ANCHOR must not rely on modifications of the TD's vFW

R-2 After updating the runtime OS kernel, the measurements must not change
R-3 The integrity of the runtime kernel is guaranteed
R-4 ANCHOR can ensure the confidentiality of the data at rest
R-5 ANCHOR induces minimal performance overhead

3.2 Architectural Considerations

In this subsection, we explore multiple options for implementing ANCHOR.

A simplistic approach for assuring the integrity of the guest OS kernel is using secure boot [4] inside the vFW of the TD. With secure boot, the vFW in the TD only allows cryptographically signed kernels to boot. However, in commercial cloud scenarios with CC, secure boot faces multiple challenges [15]: First, the attestation evidence, which can be used to verify the integrity of the cVM, does not reflect whether secure boot is enabled or not. Second, on multiple hyperscalers it is not possible to add custom signatures or TD-owner provided public keys.

Moreover, replacing or modifying the vFW is typically not possible in the cloud setting. Finally, secure boot provides integrity but does not offer confidentiality guarantees (**R-4**). Consequently, in the current cloud landscape, secure boot is insufficient as a general and deployable solution.

An alternative approach is to employ a minimal kernel as Trust Anchor. An advantage of this approach is a flexible environment for the implementation. Possible options for kernels include Asterinas [33], seL4 [22], or Linux. While the formally verified seL4 microkernel provides strong security guarantees, it lacks support for `kexec`. Furthermore, although there have been efforts to port seL4 to AMD SEV [47], support for modern CC architectures remains incomplete. In contrast, the Asterinas framekernel can run in TDX environments and aims to provide Linux ABI compatibility; however, it also lacks `kexec` support. While both seL4 and Asterinas are promising candidates, their current limitations make them unsuitable for a proof-of-concept implementation of ANCHOR.

Linux, on the other hand, supports both `kexec` and TDX, making it a practical choice for a proof-of-concept implementation. Additionally, Linux provides well-audited networking and cryptography stacks. Therefore, we base our implementation of ANCHOR on the Linux kernel with a minimal configuration, while designing the system to remain adaptable to alternative kernels such as Asterinas or seL4. We note that we only use Linux' virtio network driver and cryptography stack, which both are well-audited. This approach is consistent with related work in the realm of CC [51].

3.3 Protection at Rest

In contrast to the data in use, the confidentiality of data at rest is not protected by CC technologies like Intel TDX. Thus, to fulfill **R-4**, we need to employ additional mechanisms. Without ANCHOR, the runtime OS kernel is included in the attestation report such that a relying party could theoretically reason about its integrity. Due to Attestation Fragility, we argue that this is impractical in

real-world scenarios. Therefore, ANCHOR relies on cryptography to ensure the confidentiality and integrity of the runtime kernel without including a measurement of it in the attestation report.

In this subsection, we sketch the solution space and analyze the respective advantages and disadvantages of symmetric and asymmetric cryptography approaches in this context.

Asymmetric Cryptography. To guarantee both, integrity and authenticity of the runtime OS kernel, ANCHOR could use asymmetric signatures. Thus, every runtime OS needs to be signed by a relying party and ANCHOR only loads kernels which contain a valid signature. On the one hand, this improves the integrity and authenticity guarantees significantly. On the other hand, as the TD itself cannot sign updated kernel images, it is not able to independently update the kernel without assistance of the relying party, which performs the signing operations. Similar to signatures, the same issues arise for asymmetric encryption to ensure the confidentiality of the kernel. As this dependency on the relying party introduces a new operational effort, we deem asymmetric cryptography not suitable for our use-case.

Symmetric Cryptography. With symmetric cryptography, both the TD as well as the relying party share the same key. The key can be provisioned to the TD after successful remote attestation, i.e., when the TD is in a trusted state (Section 3.4). In particular, the availability of the key in the TD allows kernel self-updates, without additional efforts by a relying party.

To guarantee the confidentiality, widely-used encryption schemes like AES-XTS-plain64 can be used. For example, LUKS uses AES-XTS-plain64 by default. From a cryptographic perspective, an attacker might still tamper with the integrity, therefore violating **R-3**. In contrast to only using AES-XTS-plain64, Authenticated Encryption (AE) compositions can be used to protect both, integrity and confidentiality. An AE composition can be implemented by first encrypting data and then applying a MAC scheme [5, 24]. Thus, to fulfill both **R-3** and **R-4**, we propose to use the AES-XTS-plain64 together with HMAC-SHA256.

3.4 Boot Flow

With ANCHOR, we introduce a multi-stage boot process for TDs. Each stage relies on a corresponding file system. While in Stage 1 (L_1, iFS_1) the Trust Anchor is executed, Stage 2 (L_2, iFS_2) and Stage 3 (L_2, rFS) use the volatile runtime kernel that may contain secrets. The first stage is stored unencrypted on disk, while the other stages are encrypted. To transition between the stages, `kexec` [21] and `switch_root` [46] are used. `kexec` is a Linux system call that takes the new kernel to boot, together with an initial file system in RAM, often referred to as initramfs. `switch_root` is used to switch to a root file system (rootfs) on Linux.

With ANCHOR, the TD can update the runtime kernel seamlessly on its own, after it has proven its integrity to the relying party. To transform a TD image into an ANCHOR image, all steps can be done without access to TDX hardware.

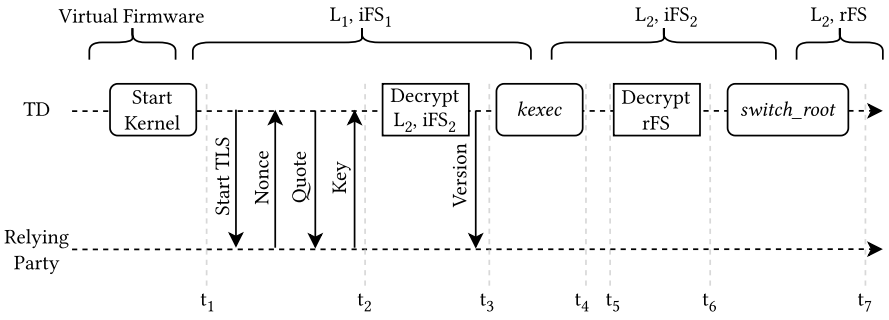

Fig. 4. Boot flow of the TD with communication between relying party and TD. t_i notes the point i in time. The TD starts at t_0 and is fully booted at t_7.

The boot flow of an ANCHOR TD is visible in Figure 4. The TD boots into the vFW, which passes the execution to the first stage consisting of the Trust Anchor L_1, together with a minimal initramfs iFS_1. We assume that a compromised hypervisor can read or modify both before starting the TD. Directly after booting the Trust Anchor, the relying party and the TD build up an encrypted TLS channel (t_1). The TLS certificate is stored in the initramfs iFS_1. To prove its integrity, the TD sends an attestation quote (Section 2) to the relying party, which contains the nonce provided by the relying party, a measurement of the kernel (L_1) and the initramfs (iFS_1), including the TLS certificate. We further include a nonce provided by the relying party for freshness and a hash of the current session handshake material to bind the attestation quote to the current TLS session. This attestation quote is sent to the relying party. Based on this attestation quote, the relying party can decide if the TD is trusted and depending on that decision provide the TD with the key to decrypt the second stage (t_2).

The second stage is encrypted on disk and contains the runtime kernel L_2 and the filesystem iFS_2. However, only encryption does not protect against rollback attacks, violating **R-3**. To protect the second stage against rollback attacks, we additionally employ a counter in the second stage. After decrypting it, the Trust Anchor reports the value of the counter to the relying party (Section 3.6). If the relying party reports that the counter value is outdated, i.e., a rollback attack is performed, the Trust Anchor aborts the execution. In other cases, we use `kexec` with the unlocked runtime kernel L_2 and the initramfs iFS_2 (t_3) to transition to the second stage. After the runtime kernel L_2 is updated, the counter is increased. By using a second initramfs, we can include the key for unlocking the rootfs rFS in iFS_2, making further communication with the relying party unnecessary. As iFS_2 is encrypted on the disk, the key for rFS cannot be accessed at rest. All data at rest used after the start of L_2 at t_4 is encrypted and therefore not readable by the hypervisor. Using the key stored in iFS_2, the TD can unlock the rootfs rFS (t_5). Finally, at t_6, the TD can use `switch_root` to switch to the now unlocked filesystem rFS. After t_7, the boot process of the TD is completed.

3.5 Partition Layout

Table 1. ANCHOR's partition layout.

Partition Number	Partition	Encrypted at Rest	Stable	In Attestation Report
1	rootfs (rFS)	✓	✗	✗
2	initramfs(iFS_2)+kernel(L_2)+counter	✓	✗	✗
3	BIOS	✗	✓	✓
4	EFI System Partition	✗	✓	✓
5	Boot Partition (iFS_1, L_1)	✗	✓	✓

Table 1 shows the partition layout of an ANCHOR image. Partition 1 holds the encrypted rootfs. With ANCHOR, partition 2 is introduced to hold the image of the second stage, i.e., the runtime kernel (L_2) and the second initramfs (iFS_2), together with a counter. L_2 is the kernel that the TD uses during runtime, and iFS_2 holds the key to partition 1. Partition 2 is encrypted on the disk. Partitions 3–5 are unencrypted at rest, and their confidentiality is therefore not protected against a compromised hypervisor. We design those partitions, such that they do not hold any secrets and do not change upon updates of the runtime OS kernel (**R-2**). Since they are included in the attestation report of the TD, they are directly attestable and a verifying party can reason about their integrity. Ideally, these components should be reproducibly buildable to enable independent verification of their integrity. In partition 5, we place our Trust Anchor and a minimal initramfs (iFS_1). By keeping these two components as minimal as possible, we reduce the need of updating these components and limit the boot time overhead. They are not active after the TD has switched to the runtime kernel and rootfs.

3.6 Relying Party

To communicate with the TD, the relying party, e.g., the TD owner, provides a TLS server and waits for connections. After a connection is established, the relying party generates a random nonce, sends it to the TD and awaits the attestation quote. Without ANCHOR, the verification process is cumbersome as the frequently updated, usually non-reproducible buildable OS kernel is included in the attestation quote. With ANCHOR, the attestation quote is expected to be rather stable, since instead of the runtime kernel, the attestation quote contains the more minimal and stable Trust Anchor. Thus, ANCHOR simplifies the operational hurdle of the relying party significantly, making attestation quote verification manageable and consequently mitigating Attestation Fragility.

After receiving the attestation quote, the relying party compares the measurement values against the pre-calculated ones. Similarly, it checks if the handshake material of the current TLS session matches to the one that is included in

the attestation quote to bind the quote cryptographically to the session. If no inconsistencies are detected and the signature of the attestation quote is valid, the decryption key is transmitted to the TD over the TLS connection. After transmitting the key, the relying party waits for the Trust Anchor to transmit the counter value stored in Partition 2, which it compares to its own reference counter. If the counter's value is bigger than the reference counter, it permits the Trust Anchor to start the runtime kernel.

4 Performance Evaluation

In this section, we evaluate ANCHOR. We outline the experimental setup before presenting the results.

4.1 Setup

For the evaluation, we build ANCHOR on top of Canonical's tooling to create a TD [8], as suggested by Intel's official TDX guide [20]. The TDs used in the evaluation are based on Ubuntu 24.04 with 4 vCPUs and 8 GB of RAM. Since ANCHOR only modifies the boot process of TDs, this is the only aspect of a TD's lifecycle that ANCHOR impacts. Therefore, we measure the time between TD boot and reaching the login prompt 100 times. To obtain timing measurements, we sample the host's Time Stamp Counter (TSC) register at multiple points during the boot process and derive elapsed times from the differences between consecutive samples.

For the test setup, we use an Intel Xeon Silver 4509Y with 128 GB of DDR5 RAM and a 1 TB WD Green S350 SSD running Ubuntu 25.10 as the hypervisor OS. To reduce network effects, we run the relying party directly on the hypervisor.

Currently, Intel TDX guests support `kexec` usage only on single-core TDs [43, 44]. Further analysis showed that the main problem is that ACPI MADT does not support to shut down CPU cores after they have been started [42]. We modify the Trust Anchor to only bring up one core. However, this does not pose a limitation for the cores of the runtime kernel, which brings up secondary cores seamlessly. Thus, during runtime, all available vCPU cores are usable.

We vary the software stack in the following variations:

- Baseline: Unmodified Ubuntu 24.04 TD without encrypted rootfs
- ANCHOR: Implementation of ANCHOR in the TD without encrypted rootfs
- ANCHOR-rFS: Implementation of ANCHOR in the TD with encrypted rootfs

4.2 Results

Figure 5 shows the measurements of the three configurations: Baseline, ANCHOR, and ANCHOR-rFS. The standard deviation is below 1.0 s for each measurement point. In the baseline configuration, we measure 10.5 seconds to reach the beginning of `switch_root`, and in total 15.7 seconds to reach the login prompt. The

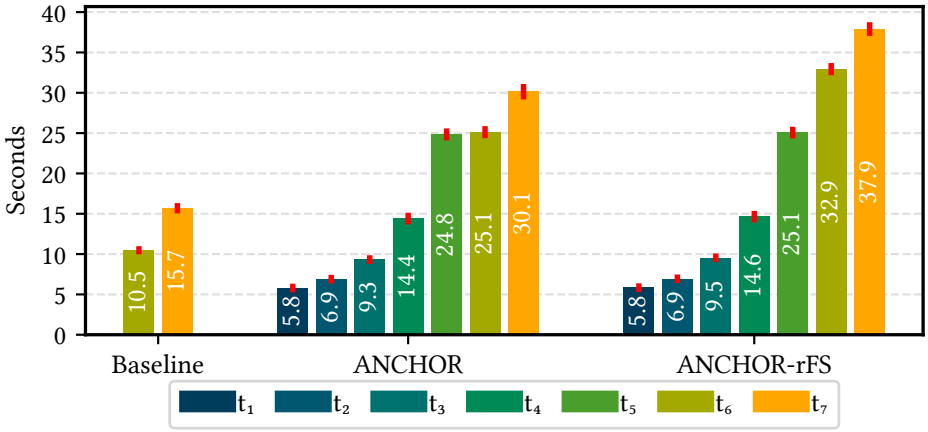

Fig. 5. Cumulative average boot time across three configurations with standard deviation: Baseline, ANCHOR without encrypted rootfs, and ANCHOR with encrypted rootfs. Consistent with Figure 4, the markers t_i denote the following events: t_1 boot of L_1, t_2 key transfer to the TD, t_3 start of `kexec`, t_4 boot of L_2, t_5 decryption (if applicable) of rFS, t_6 start of `switch_root`, and t_7 completion of the L_2 boot process.

measurements of ANCHOR and ANCHOR-rFS are divided into seven time periods, as shown in Figure 4 (Section 3.4). For both, ANCHOR and ANCHOR-rFS, t_1 to t_5 are almost identical with differences of at most 0.25 seconds. This matches our expectations, as the executed software and execution flow until t_5 is identical. After t_5, ANCHOR-rFS decrypts the rootfs, which adds 7.8 seconds of boot time to reach the login prompt.

Compared to the baseline, ANCHOR increases total boot time by 14.4 seconds, corresponding to a 1.92× overhead. To quantify the costs of ANCHOR's boot time at a fine granular basis, we further decompose the overhead of ANCHOR into four components and quantify each component's average contribution. The boot of the Trust Anchor takes 5.8 seconds (40.1% of ANCHOR's boot time overhead). This time span, however, depends on the kernel used for the Trust Anchor. We expect differing times to be measured if another kernel, e.g., Asterinas [33], is used instead. Subsequently, the communication with the relying party takes place, which accounts for 1.1 seconds (7.6%). We expect this value to depend on the characteristics of the underlying network communication between the host and the relying party. In default cVM scenarios without ANCHOR, the cVM would also request an attestation quote after booting and supply it to a relying party [48, 28]. Thus, any overhead that would occur on different network configuration between host and relying party is not induced by ANCHOR, but inherited from the default CC model. The decryption of the next stage (L_2, iFS_2) takes 2.4 seconds (16.8%). Finally, the execution of `kexec` accounts for 5.1 seconds (35.2%).

ANCHOR's core does not affect the overhead during the runtime of the system, i.e., after the TD switched to the runtime kernel, and only adds a one-time boot

overhead of 14.4 seconds. We consider the one-time overhead justified for most real world cVMs scenarios, e.g., for machine learning [30] or database systems [36], which usually run for longer time periods. Although ANCHOR-rFS encrypted rootfs can induce runtime overhead, encryption of the root filesystem in general is ubiquitous in the realm of CC, even without ANCHOR [7, 19, 11]. Therefore, we do not consider the runtime overhead of an encrypted filesystem as closely linked to ANCHOR itself. For evaluating the performance of disk encryption in general, we refer to related work [23, 10]. In summary, we consider ANCHOR practical and broadly applicable, fulfilling **R-5**.

5 Security Discussion

We assume the primary goal of an adversary is to break the integrity and confidentiality of the runtime kernel used for a TD. In line with the general CC and Intel TDX threat model (Section 2), we assume that an attacker has full control over the hypervisor. Further, they can intercept and modify network packets. During runtime, we assume that kernel and filesystem are protected by Intel TDX mechanisms (Section 2). Shielding the TD's interfaces to the hypervisor during runtime is orthogonal to our approach. Similarly, we assume that the Trust Anchor is implemented correctly and bug free.

The data at rest of the kernel, i.e., when the TD is not running, is encrypted using an AE scheme. Thus, to break the confidentiality or integrity, the attacker needs to break the underlying cryptographic scheme or obtain the key that was used to encrypt the kernel. As we assume that the cryptographic schemes are secure, the attacker needs to obtain the decryption key, in order to tamper with the integrity or confidentiality unnoticed. In the following, we explore existing attack surfaces that an attacker can use and explain how ANCHOR defends against such attacks.

Boot Attacks. The attacker can tamper with the boot configuration of the TD, i.e., start the TD with debug features or start the provided image as normal VM without TDX protection. Furthermore, the unencrypted Stage 1 image can be tampered with, for example by injecting a small script that extracts the decryption key during runtime.

Before provisioning the key to the TD, the relying party waits for an attestation quote from the TD. The attestation quote contains measurements of both hardware and software (including kernel and initramfs) inside the TD. Therefore, if either the TD or its environment is tampered with, the relying party would detect inconsistencies in the attestation quote, assuming the underlying hardware platform works according to its specification. Thus, the relying party does not provide the decryption key in this scenario to the TD. Since with ANCHOR, the measurements inside the attestation quote are not supposed to change often due to the stable Trust Anchor, the verification process is manageable even in large scale cVM deployments.

Man-in-the-Middle Attacks. We assume that the hypervisor-based or network-based attacker can intercept all messages that are sent or received by a

TD and conduct Man in the Middle (MitM) attacks. Moreover, an attacker can completely block the network traffic. However, CC in general does not provide any availability guarantees, as a compromised cloud provider can just power off the server. ANCHOR protects data in transit by using the TLS protocol. As the decryption key is transmitted over the TLS channel, the confidentiality and integrity of it is bound to the security of the TLS protocol. To extract the key, an attacker can try to mount a MitM attack. To achieve this, they can extract the TLS certificate of the relying party and build a connection to the relying party. Furthermore, they can try to replace the TLS certificate of the TD used for the communication with a certificate of their own to be able to read the traffic. As the TLS certificate is stored in the unencrypted partition of the stage, the modification of the certificate is initially possible. Finally, by tampering with the network configuration, all traffic can be routed through an attacker, who can read all transmitted data in clear text. However, before the relying party transmits the decryption key over the TLS channel, it verifies the attestation quote. As the attestation quote contains a measurement of the initramfs, where the TLS certificate is stored, MitM attacks are recognized and no decryption key is sent.

Impersonation Attacks. As introduced in Section 2, on an Intel TDX-enabled system, the hypervisor is responsible for forwarding the attestation quote from the quoting enclave back to the TD for the signing process. On transit, the integrity before the attestation quote is signed, is guaranteed by a MAC. Due to this design, an attacker controlling the hypervisor can access and capture the attestation quote in clear text before it is accessible to the TD itself. In addition, the hypervisor can freeze or terminate the execution of any TD. Simultaneously, as the Trust Anchor and its initramfs are not encrypted, an attacker controlling the hypervisor can extract the TLS certificate before the TD is started.

Combining these abilities, the attacker can mount an impersonation attack: Instead of forwarding the attestation quote to the TD, they save it and freeze the TD. Subsequently, the attacker builds up a TLS connection to the relying party using the TD's TLS certificate. Over this malicious TLS channel, the attacker then can relay the intercepted attestation quote to the relying party to impersonate a legitimate TD and obtain the decryption key. ANCHOR detects such attacks by including a hash of the current session information in the attestation quote. As this session information is derived of handshake information and the master secret, it is unique per session. Before provisioning the decryption key, the relying party recomputes the session information on its side and checks if it matches to the one in the attestation quote. In the TLS session between the hypervisor and the relying party, these values do not match, so no decryption key is sent.

Rollback Attacks on the Kernel. A compromised cloud provider can restore a previously valid version of a TD image at a later point in time. While such an attack does not directly violate integrity or confidentiality, it can cause the system to boot in an outdated state. To tackle this attack vector, the Trust Anchor reports the counter value of the second stage to the relying party. If the

relying party classifies the version as outdated, `kexec` is not executed and the system aborts execution. An attacker might try to tamper with the counter by attempting to overwrite it on disk or during runtime of the TD. Since the kernel is encrypted with an AE composition on disk, this is not possible, if the underlying cryptography holds. During runtime, overwriting the counter is not possible due to Intel TDX' protection, which prevents the hypervisor from accessing the VM's contents. Thus, ANCHOR can effectively detect rollback attacks and prevent a compromised cloud provider from starting outdated runtime kernels.

6 Discussion

In this section, we discuss limitations of ANCHOR and future work.

Alternative Approaches. Section 3.2 explores multiple approaches for the implementation of the Trust Anchor, each with distinct trade-offs. While our implementation deliberately prioritizes deployability on current cloud platforms, future work may revisit alternative approaches as their ecosystems mature. In particular, formally verified or specialized kernels could provide stronger security guarantees once they offer the required functionality (i.e., `kexec` and compatibility with CC platforms). This is in line with prior work on CC [51].

Shift of the Attestation Boundary. With ANCHOR, we shift the attestation boundary away from the runtime kernel to a Trust Anchor. Rather than attesting the runtime kernel, which we deem impractical under frequent updates, hardware measurements are bound to a minimal Trust Anchor. Trust in the runtime kernel is then established transitively: the cVM owner, acting as the relying party, releases the decryption key only to the hardware-attested Trust Anchor. As ANCHOR exclusively decrypts and loads authenticated images, it guarantees that the runtime kernel has not been tampered with by a third party, such as the hypervisor. While the Trust Anchor itself may occasionally require updates, its minimal nature means that such events are rare and tractable.

Other Confidential Computing Architectures. Similar to Intel TDX, other CPU vendors also have proposed VM-based CC architectures, including AMD SEV-SNP [3] and ARM CCA [26]. While this work implements ANCHOR for Intel TDX, the concepts are also transferable to other CC designs. ANCHOR heavily relies on remote attestation, which is implemented by all comparable CC architectures. For example, although AMD SEV-SNP does not provide runtime measurements like Intel TDX, related work [18, 17] showed that measured direct boot can give the same integrity assurances. We observe that AMD SEV-SNP with measured direct boot is also affected by the Attestation Fragility, as every OS kernel update alters the launch measurement. Despite this, with ANCHOR, the integrity of the Trust Anchor and its initramfs are attestable to a remote relying party. The following stages are independent of the architecture, so they can be applied in a similar way like on Intel TDX. Thus, we conclude that the concepts of ANCHOR can be applied to other CC architectures. Future work may explore implementations on other CC platforms.

7 Related Work

Remote Attestation. Eisoldt et al. [15] analyze CC offerings by popular cloud providers for both Intel TDX and AMD SEV-SNP. The focus of their examination is on the setup of the cVMs and the remote attestation. Their analysis shows that sometimes, cloud providers retain control over critical components running inside cVMs. Crucially, the vFW used in cVMs is often not customizable in these environments. SNPGuard [48] implements a protocol for full disk encryption on AMD SEV-SNP. However, to achieve this, they modify the vFW of the cVM, which usually cannot be done on commercial platforms, as shown by Eisoldt et al. [15]. Similarly, multiple commercially oriented and academic tools exist [7, 19, 11] to enhance the confidentiality of the data at rest for cVMs. While they enable encryption of the filesystem for different use-cases and CC architectures, the confidentiality of the kernel is not covered. Additionally, none of these approaches tackle the highlighted problem of Attestation Fragility.

Revelio [17] offers an attestation approach to ensure that a web-facing workload is running in a cVM. While Revelio highlights the problem of reproducible builds, the update frequency of modern systems is not considered. Preibsch et al. [35] extend Revelio with a remote attestation scheme inside web browsers.

Apart from CC architectures like AMD SEV-SNP and Intel TDX, past research explored remote attestation on TEE architectures based on ARM [38, 27] and RISC-V [6, 41].

Reproducible Builds. The fundamental promise of CC, verifying that untrusted hosts run the exact software requested, rests on exact binary reproducibility: if a user cannot recreate a cVM image bit-for-bit from source, the measurement quote is a black box that necessitates blind trust in the infrastructure provider. Achieving this in practice is a significant engineering hurdle. Even from identical sources, the Reproducible Builds project [37] reports that binaries are frequently altered by *environmental volatility*, i.e., embedded metadata such as timestamps, build paths, or the build machine's user and hostname, and by *toolchain sensitivity*, i.e., discrepancies in compiler minor versions. Projects like *NixOS* [14] and *Yocto* [50] have made reproducibility a core design feature, but only through years of radical re-engineering of entire build pipelines, which is significantly harder to replicate for mainstream distributions like Debian [12]. ANCHOR acknowledges that fully reproducible builds are currently out of reach for mainstream, full-blown OSs.

8 Conclusion

We present ANCHOR, an architecture that transforms the attestation of cVMs from a fragile, cumbersome process to a simple, stable workflow. It provides a stable Trust Anchor in the volatile cloud landscape, ensuring that security updates no longer come at the cost of broken trust relationships. ANCHOR achieves this by decoupling the attestation measurement from the runtime OS kernel, while still ensuring its integrity. Additionally, ANCHOR extends the data in use

confidentiality protection of CC to data at rest. We measure the overhead and conclude that it is practical in real-world scenarios. Therefore, we believe that ANCHOR lowers the adoption barrier for a secure usage of CC significantly.

Acknowledgments. We thank the reviewers for their helpful feedback. This research was partly supported by the German Federal Ministry of Research, Technology, and Space (BMFTR) as part of the CELTIC-NEXT project SUSTAINET-inNOvAte ("Frictionless, secure, and resilient communication networks for the dynamic digital world", Förderkennzeichen "16KIS2258" and "16KIS2262").

Author Contribution Statement This section uses the CRediT taxonomy (`https://credit.niso.org/`). **Julian Funk:** Conceptualization, Methodology, Validation, Visualization, Software, Writing – Original Draft, Writing – Review & Editing. **Matti Schulze:** Writing – Review & Editing. **Patrick Sieber:** Software. **Manuel Vögele:** Writing – Review & Editing. **Jonas Röckl:** Conceptualization, Writing – Review & Editing. **Tilo Müller:** Supervision, Writing – Review & Editing.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ali, M., Khan, S.U., Vasilakos, A.V.: Security in cloud computing: Opportunities and challenges. Inf. Sci. (2015). `https://doi.org/10.1016/J.INS.2015.01.025`
2. Amazon Web Services, AWS UEFI Firmware, GitHub repository (2026). `https://github.com/aws/uefi`. Accessed: 2026-08-14.
3. AMD, AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. White Paper, Advanced Micro Devices, Inc. (2020)
4. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A Secure and Reliable Bootstrap Architecture. In: 18th IEEE Symposium on Security and Privacy, IEEE S&P 1997, Oakland, CA, USA, May 4-7 (1997). `https://doi.org/10.1109/SECPRI.1997.601317`
5. Bellare, M., Namprempre, C.: Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In: 6th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT 2000, Kyoto, Japan, December 3-7. LNCS, Springer, Heidelberg (2000). `https://doi.org/10.1007/3-540-44448-3_41`
6. Bove, D., Funk, J.: Basic secure services for standard RISC-V architectures. Comput. Secur. (2023). `https://doi.org/10.1016/J.COSE.2023.103415`
7. Brescia, L., Colonnelli, I., Schiavoni, V., Felber, P., Aldinucci, M.: End-To-End Confidentiality with Sev-Snp Leveraging in-Memory Storage. In: IEEE European Symposium on Security and Privacy Workshops, EuroS&PW 2025, Venice, Italy, June 30 - July 4 (2025). `https://doi.org/10.1109/EUROSPW67616.2025.00054`
8. Canonical Ltd., Intel Trust Domain Extensions (TDX) on Ubuntu, GitHub repository (2025). `https://github.com/canonical/tdx`. Accessed: 2026-02-12.
9. Cheng, P., Ozga, W., Valdez, E., Ahmed, S., Gu, Z., Jamjoom, H., Franke, H., Bottomley, J.: Intel TDX Demystified: A Top-Down Approach. ACM Comput. Surv. (2024). `https://doi.org/10.1145/3652597`

10. Chrapek, M., Orenbach, M., Atamli, A., Copik, M., Alder, F., Hoefler, T.: sNVMeoF: Secure and Efficient Disaggregated Storage. CoRR (2025). https://doi.org/10.48550/ARXIV.2510.18756. arXiv: 2510.18756
11. Confidential Computing API Group, TDX Full Disk Encryption, GitHub repository (2025). https://github.com/cc-api/full-disk-encryption. Accessed: 2026-02-12.
12. Debian Project, Debian Reproducible Builds, Online service (2026). https://reproduce.debian.net/. Accessed: 2026-04-12.
13. DionnaGlaze, Open source OVMF binaries, Google Developer Forums post (2025). https://discuss.google.dev/t/open-source-ovmf-binaries/173731/4. Accessed: 2026-04-12.
14. Dolstra, E., Löh, A., Pierron, N.: NixOS: A purely functional Linux distribution. J. Funct. Program. (2010). https://doi.org/10.1017/S0956796810000195
15. Eisoldt, J., Galanou, A., Ruzhanskiy, A., Küchenmeister, N., Baburkin, Y., Dai, T., Gudymenko, I., Köpsell, S., Kapitza, R.: A Cloudy View on Trust Relationships of CVMs: How Confidential Virtual Machines are Falling Short in Public Cloud. In: 41st Annual Computer Security Applications Conference, ACSAC 2025, Honolulu, HI, USA, December 8-12 (2025). https://doi.org/10.1109/ACSAC67867.2025.00046
16. fnerdman, Support for Reproducible Builds and MRTD Measurement Reproduction in Azure TDX VMs, GitHub issue (2024). https://github.com/microsoft/openvmm/issues/181. Accessed: 2026-08-14.
17. Galanou, A., Bindlish, K., Preibsch, L., Pignolet, Y., Fetzer, C., Kapitza, R.: Trustworthy confidential virtual machines for the masses. In: 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15 (2023). https://doi.org/10.1145/3590140.3629124
18. Galanou, A., Lubitz, F., Jeon, H., Fetzer, C., Kapitza, R.: Full Trust Alchemist: Reforging Attestation for Cloud-based Confidential Workloads. In: 26th International Middleware Conference, Middleware 2025, Nashville, TN, USA, December 15-19 (2025). https://doi.org/10.1145/3721462.3770778
19. Intel Corporation, Full Disk Encryption with Intel Trust Domain and Intel Trust Authority, Online technical documentation (2026). https://www.intel.com/content/www/us/en/developer/articles/technical/disk-encryption-intel-trust-domain-trust-authority.html. Accessed: 2026-02-12.
20. Intel Corporation, Guest OS Setup – Intel TDX Enabling Guide, Online technical documentation (2024). https://cc-enabling.trustedservices.intel.com/intel-tdx-enabling-guide/06/guest_os_setup/. Accessed: 2026-02-12.
21. kexec(8): directly boot into a new kernel, Linux man page (2006). https://man7.org/linux/man-pages/man8/kexec.8.html. Accessed: 2026-02-12.
22. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM (2010). https://doi.org/10.1145/1743546.1743574
23. Korchagin, I.: Speeding Up Linux Disk Encryption. In: 2020 Linux Storage and Filesystems Conference, Vault 2020, Santa Clara, CA, USA, February 24-25 (2020)
24. Krawczyk, H.: The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?) In: 21st Annual International Cryptology Conference, CRYPTO 2001, Santa Barbara, CA, USA, August 19-23. LNCS, Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44647-8_19

25. Lamb, C., Zacchiroli, S.: Reproducible Builds: Increasing the Integrity of Software Supply Chains. IEEE Softw. (2022). https://doi.org/10.1109/MS.2021.3073045
26. Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., Stockwell, G.: Design and Verification of the Arm Confidential Compute Architecture. In: 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13 (2022). https://www.usenix.org/conference/osdi22/presentation/li
27. Ling, Z., Yan, H., Shao, X., Luo, J., Xu, Y., Pearson, B., Fu, X.: Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes. J. Syst. Archit. (2021). https://doi.org/10.1016/J.SYSARC.2021.102240
28. Ménétrey, J., Göttel, C., Khurshid, A., Pasin, M., Felber, P., Schiavoni, V., Raza, S.: Attestation Mechanisms for Trusted Execution Environments Demystified. In: 22nd IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, DAIS 2022, Lucca, Italy, June 13-17. LNCS, Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-16092-9_7
29. Misono, M., Stavrakakis, D., Santos, N., Bhatotia, P.: Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. Proc. ACM Meas. Anal. Comput. Syst. (2024). https://doi.org/10.1145/3700418
30. Mo, F., Tarkhani, Z., Haddadi, H.: Machine Learning with Confidential Computing: A Systematization of Knowledge. ACM Comput. Surv. (2024). https://doi.org/10.1145/3670007
31. Mulligan, D.P., Petri, G., Spinale, N., Stockwell, G., Vincent, H.J.M.: Confidential Computing - a brave new world. In: 1st International Symposium on Secure and Private Execution Environment Design, SEED 2021, Washington, DC, USA, September 20-21 (2021). https://doi.org/10.1109/SEED51797.2021.00025
32. Parast, F.K., Sindhav, C., Nikam, S., Yekta, H.I., Kent, K.B., Hakak, S.: Cloud computing security: A survey of service-based models. Comput. Secur. (2022). https://doi.org/10.1016/J.COSE.2021.102580
33. Peng, Y., Tian, H., Zhang, J., Li, R., Chen, C., Jiang, J., Xian, J., Wang, X., Xu, C., Zhou, D., Luo, Y., Yan, S., Zhang, Y.: ASTERINAS: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB. In: 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9 (2025). https://www.usenix.org/conference/atc25/presentation/peng-yuke
34. Perezvargas, C.: OpenHCL: the new, open source paravisor, Microsoft Community Hub blog post (2024). https://techcommunity.microsoft.com/blog/windowsosplatform/openhcl-the-new-open-source-paravisor/4273172. Accessed: 2026-04-12.
35. Preibsch, L., von Onciul, M.R., Kapitza, R.: Site Attestation: Browser-based Remote Attestation. In: 18th European Workshop on Systems Security, EuroSec 2025, Rotterdam, The Netherlands, March 30 - April 3 (2025). https://doi.org/10.1145/3722041.3723095
36. Qiu, L., Taft, R., Shraer, A., Kollios, G.: The Price of Privacy: A Performance Study of Confidential Virtual Machines for Database Systems. In: 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, June 10 (2024). https://doi.org/10.1145/3662010.3663440
37. Reproducible Builds, Reproducible Builds Project, Project website (2026). https://reproducible-builds.org/. Accessed: 2026-08-14.
38. Röckl, J., Schulze, M., Funk, J., Bernsdorf, N., Müller, T.: Safeguarding Data from Deprecated System Software: A Resilient Boot Architecture for Edge Devices Leveraging TEEs. In: 9th International Conference on Information Systems

Security and Privacy, ICISSP 2023, Lisbon, Portugal, February 22-24. Communications in Computer and Information Science. Springer (2023). https://doi.org/10.1007/978-3-031-89518-0_4
39. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted Execution Environment: What It is, and What It is Not. In: 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015, Helsinki, Finland, August 20-22 (2015). https://doi.org/10.1109/TRUSTCOM.2015.357
40. Sardar, M.U., Musaev, S., Fetzer, C.: Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification. IEEE Access (2021). https://doi.org/10.1109/ACCESS.2021.3087421
41. Shepherd, C., Markantonakis, K., Jaloyan, G.: LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices. In: IEEE Security and Privacy Workshops, SPW 2021, San Francisco, CA, USA, May 27 (2021). https://doi.org/10.1109/SPW53761.2021.00036
42. Shutemov, K.A.: [PATCH 12/13] x86/acpi: Do not attempt to bring up secondary CPUs in kexec case, kexec mailing list post (2023). https://lists.infradead.org/pipermail/kexec/2023-October/028254.html. Accessed: 2026-02-12.
43. Shutemov, K.A.: [PATCH] x86: Disable kexec for TDX guests, kexec mailing list post (2023). https://lists.infradead.org/pipermail/kexec/2023-March/026825.html. Accessed: 2026-02-12.
44. Shutemov, K.A.: [PATCHv3 00/14] x86/tdx: Add kexec support, Patchew.org Linux kernel patch archive (2023). https://patchew.org/linux/20231115120044.8034-1-kirill.shutemov@linux.intel.com/20231115120044.8034-14-kirill.shutemov@linux.intel.com/. Accessed: 2026-02-12.
45. smo4201, openhcl: tdx mrtd precomputation, GitHub issue (2025). https://github.com/microsoft/openvmm/issues/2247. Accessed: 2026-08-14.
46. switch_root(8): switch to another filesystem as the root of the mount tree, Linux man page (2015). https://man7.org/linux/man-pages/man8/switch_root.8.html. Accessed: 2026-02-12.
47. Weidinger, A.: Improving Confidential Computing with seL4: A Promising Guest OS Solution. In: seL4 Summit 2025, Prague, Czech Republic, September 3-5 (2025). https://sel4.org/Summit/2025/slides/improving-confidential.pdf
48. Wilke, L., Scopelliti, G.: SNPGuard: Remote Attestation of SEV-SNP VMs Using Open Source Tools. In: IEEE European Symposium on Security and Privacy Workshops, EuroS&PW 2024, Vienna, Austria, July 8-12 (2024). https://doi.org/10.1109/EUROSPW61312.2024.00026
49. Xiao, Z., Xiao, Y.: Security and Privacy in Cloud Computing. IEEE Commun. Surv. Tutorials (2013). https://doi.org/10.1109/SURV.2012.060912.00182
50. Yocto Project, The Yocto Project Documentation, Project documentation (2026). https://www.yoctoproject.org/. Accessed: 2026-08-14.
51. Zhao, S., Li, M., Zhang, Y., Lin, Z.: vSGX: Virtualizing SGX Enclaves on AMD SEV. In: 43rd IEEE Symposium on Security and Privacy, IEEE S&P 2022, San Francisco, CA, USA, May 22-26 (2022). https://doi.org/10.1109/SP46214.2022.9833694