Active and Passive Attacks against TPM 2.0-based Full Disk Encryption

Sven Freud[1], Lennart Hein[1], Dominik Klein [1]

Federal Office for Information Security (BSI), Bonn, Germany
{firstname.lastname}@bsi.bund.de

Abstract.
Trusted Platform Module (TPM) 2.0-based full-disk encryption (FDE) has gained renewed importance with the rise of bring-your-own-device environments. Aside from Windows Bitlocker, which uses the TPM since the days of Windows Vista, FDE has also been recently adopted by several Linux distributions. We revisit tamper attacks against TPM 2.0-based FDE systems and analyze the security of current Windows and Linux implementations. We examine TPM 2.0 sealing mechanisms and demonstrate passive and active attacks against systems using discrete TPMs (dTPMs) connected via SPI, using a mini PC and a notebook platform as case studies. While dTPMs are highly tamper-resistant, they remain vulnerable to attacks on the communication bus. We further show that active attacks are not limited to dTPM bus interception but can also affect firmware TPMs (fTPMs). Specifically, we present a prototype attack in a virtualized QEMU environment targeting UEFI firmware rather than bus communication, impacting both Windows and modern Linux systems. These attacks enable us to break not only BitLocker-based full-disk encryption on Windows, but also all current Linux implementations, including those that utilize parameter encryption to secure the bus communication. Our findings expose weaknesses in current TPM 2.0 FDE implementations and motivate stronger specification-level protections.

Keywords: TPM · trusted computing · SPI · LPC · BitLocker · LUKS · Full Disk Encryption

1 Introduction

In this report, we investigate full-disk encryption implementations utilizing a Trusted Platform Module (TPM). We examine the Windows 11 BitLocker implementation, as well as several Linux Unified Key Setup (LUKS) variants. We evaluate these implementations at a conceptual level, analyzing their security design as well as various attack paths. Our scope encompasses both logical security and physical tamper resistance.

The TPM specification by the Trusted Computing Group is a specification on a functional level. It does not define threat models or attackers, as these depend on the specific TPM implementation. In our investigation we assume a *physical attacker* with moderate expertise and having typical low-cost probing equipment at his disposal; in particular a soldering station and a logic analyzer. Most importantly, all attacks also assume *physical access* to the device-under-attack.

In our taxonomy, we distinguish—akin to the Trusted Computing Group's (TCG) recommendations—between passive and active attacks. In a passive attack, an attacker merely observes the host system, the TPM, and their communication with each other without modifying anything. An active attacker, however, logically or physically interferes

with any of these three components. We map these various attacks and their applicability to Windows BitLocker and the various Linux implementations.

Passive attacks, especially in the context of the Microsoft BitLocker implementation, have been extensively described online as well as at various security conferences. However, they came back into focus with the mandatory requirement of a TPM in any computer that runs Windows 11. Recently, significant simplification of the tools and hardware required to implement such physical passive attacks [30, 42] have been introduced, thereby lowering the barrier to entry for attackers and posing a risk that should not be underestimated.

One proposed countermeasure is to encrypt the bus communication between the host system and the TPM, preventing such passive attacks via a mechanism currently available in TPMs known as *parameter encryption*. Here, we pose the question of to what extent parameter encryption prevents passive and active attackers, and, if attacks are possible, the practical effort required to execute them.

In particular, we show that parameter encryption alone is not sufficient to prevent active attackers. In fact, our results show that active attacks are often not significantly more difficult—and can even be easier—than passive attacks to execute. Moreover, we make deliberate design choices transparent. These design choices reveal that certain attacks were deliberately excluded from the threat model, potentially leading one to question whether they should be categorized as attacks at all. However, our analysis also shows that these design decisions may contradict typical user expectations.

TPMs come in different form factors. These include firmware TPMs (fTPMs), which are software implementations running in a dedicated part of the CPU and/or on firmware controllers, and discrete TPMs (dTPMs), which consist of dedicated silicon. The latter come with explicit protection against physical tampering by attackers with high attack potential, making them the more secure option from a tamper-resistance perspective. The drawback is that they leave the bus connecting the host system and the TPM exposed. We show, however, that active attacks apply to both fTPMs and dTPMs alike, even though the implementation of the attacks varies.

Finally, we propose countermeasures to prevent such attacks. These include measures compatible with the current TPM specification, such as dedicated user authentication with a PIN value in combination with parameter encryption, both of which are not enabled by default or are not available on Windows 11 and several Linux distributions. Regarding dTPMs, we highlight that tamper-resistant hardware for the TPM alone does not suffice to make the platform secure. While some vendors have recently introduced proprietary solutions to combat active and passive attacks, our work highlights some gaps in the TPM specification, namely, that an additional bus-encryption method is required alongside parameter encryption.

This study is structured as follows: In Section 2, we illustrate TPM 2.0-based full-disk encryption on a conceptual level as well as concrete implementations in Windows and several Linux variants. This serves as the basis for Section 3, where we describe both passive bus-sniffing and active state forgery attack paths. In Section 5, we present our practical experimental results. First, we revisit passive bus-sniffing attacks and estimate their difficulty in practice. Next, we implement active attacks on a notebook computer, breaking not only BitLocker-based full-disk encryption on Windows, but also all current Linux implementations, including those that utilize parameter encryption. We also prototypically show that active attacks apply to fTPMs as well by modifying the TianoCore UEFI implementation in a virtualized QEMU environment.

Following these experimental demonstrations, Section 4 systematically analyzes countermeasures, distinguishing between physical and bus-level protections and firmware-level platform roots of trust.

In Section 6, we discuss related work and show which attacks have been discussed in the literature or online before, both on a logical level and in terms of practical implementation;

Host System		**TPM**
		reset $\mathsf{PCR}:= 0$
$m_0 = \mathcal{H}(d_0)$	$\xrightarrow{m_0}$	$\mathsf{PCR} := \mathcal{H}(\mathsf{PCR} \parallel m_0)$
$m_1 = \mathcal{H}(d_1)$	$\xrightarrow{m_1}$	$\mathsf{PCR} := \mathcal{H}(\mathsf{PCR} \parallel m_1)$
$\vdots$	$\vdots$	$\vdots$
$m_i = \mathcal{H}(d_i)$	$\xrightarrow{m_i}$	$\mathsf{PCR} := \mathcal{H}(\mathsf{PCR} \parallel m_i)$

Protocol 1: Hashing into TPM PCRs

some of these works serve as inspiration for our own. In Section 7, we conclude our presentation and mention future work both in terms of research and standardization.

2 TPM 2.0-based Full Disk Encryption

Central to the TPM 2.0 specification are Platform Configuration Registers (PCRs). They are designed to store information about the current system state in such a way that the boot state of the system can be measured. Using these measurements, certain policies can be defined. For example, one can define a measurement as a reference for a known good state of the system. In that state, an action can be taken, e.g. releasing a secret key value from the TPM. If, however, the measurement shows that the system's state is different from that known good state, the key must not be released to the outside world but remain safely inside the TPM.

Technically, the TPM 2.0 specification defines 24 PCRs that track boot components and the system state. They are designed to be volatile so that the measurement chain starts fresh with every boot. During the boot process, the initial stage is evaluated by computing a cryptographic hash over the UEFI code components (d_0). The resulting hash constitutes the measurement $m_0 = \mathcal{H}(d_0)$. This measurement m_0 is then sent to the TPM to be extended into a defined PCR.

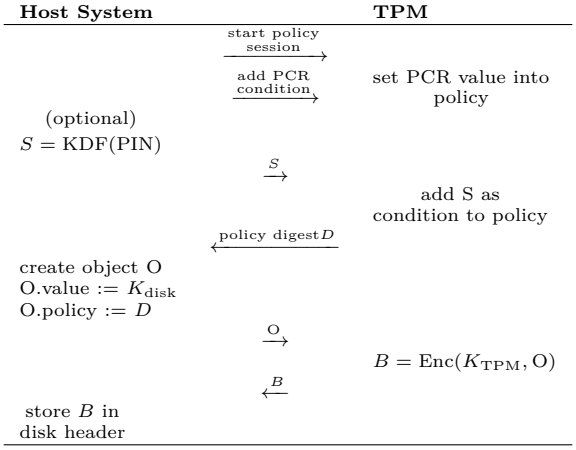
While the TPM is powered on, PCR values cannot be reset; they can only be *extended* as follows: Given a current value PCR and a new measurement m_i, the new PCR value is computed as $\mathsf{PCR} := \mathcal{H}(\mathsf{PCR} \parallel m_i)$. This process is depicted in Protocol 1.

By creating such hash chains, any modification in a boot stage results in a different measurement and, hence, a different PCR value. While the boot process is not stopped in case of a modification, any change of the system state can be reliably detected, as the PCRs themselves are protected against tampering and do not allow a reset. However, while the discrete TPM chip internally protects its registers, the underlying platform threat model does not enforce tamper-proofing for the external motherboard traces, reset lines, or power supply. This leaves the system vulnerable to adversaries who manipulate signals outside the physical boundary of the TPM silicon.

Full-disk encryption encrypts an entire logical volume with a high-entropy symmetric key, e.g. an AES key. The concept of TPM 2.0-based full-disk encryption is to securely store the disk encryption key inside the TPM and only release it if the system measures into the known good state. The initial key provisioning is called *sealing* in the context of TPM and is depicted in Protocol 2.

2.1 Sealing and Unsealing

Here, we assume that the symmetric disk encryption key K_{disk} has been generated, the volume has been encrypted, and the system has been booted into a good state. The caller, i.e. the operating system, now starts a policy session with the TPM. The policy defines the conditions under which the TPM will release the disk encryption key for the operating



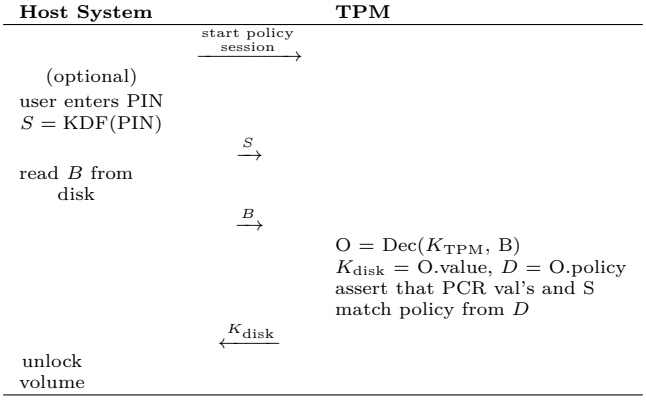
Protocol 2: Sealing

system to decrypt the volume after the initial boot stage has been finished. The caller then adds PCR values as a condition to that policy; these can reflect the current state or precalculated expected values. Optionally, an additional authentication value S can be used, i.e. a value derived with a key derivation function from a (potentially low-entropy) PIN. This value is then added as a condition to the policy. The PIN then has to be entered by the user during each boot. The TPM generates the policy and sends back a policy digest D to the host-system. Next, an object O is created by the host-system. The class member *value* is set to the disk encryption key K_{disk}, and the class member *policy* is set to the policy digest; and the object O is sent to the TPM. As storage on a TPM is limited, the TPM itself does not store the disk encryption key directly, but encrypts it with a secret key (K_{TPM}) known only to the TPM and sends the sealed blob B back to the host-system. This sealed blob, containing the encrypted disk encryption key as well as the policy digest, is then stored on the hard disk. From now on, the TPM will release (unseal) this object only if the policy conditions defined in the digest are met.

At this stage, a backup key is usually also generated: the disk encryption key is encrypted with a password and stored on the disk as well. The password can then be backed up by the user, either offline or in the cloud. TPMs have anti-hammering protection that is designed to prevent brute force attacks, or more complex dictionary attacks. It is therefore safe to use a low-entropy PIN for the optional authentication value during sealing. As the backup key is stored completely unrelated to the TPM one must ensure that a high-entropy password — difficult to remember for the user, but safe against dictionary attacks — is generated.

After the initial sealing, on each bootup the disk encryption key must be unsealed from the TPM, depicted in Protocol 3. During the initial stages of bootup, the firmware measures system states and reports hash values to the TPM as before. Next the host system reads out the encrypted object B from the disk and sends it to the TPM. The TPM then extracts the policy digest D, and compares the current state of the PCRs to the known "good" values of the system. If they match, the system is in a known, "good" state and the policy condition is met. If a PIN was set during sealing, this PIN is also checked if it meets the policy from the digest D. Next, the host system requests to unseal the disk encryption key, and if all policy rules match, the TPM sends the disk encryption key K_{disk} back to the host system.

Without going into details regarding potential attacks, there are two observations: First, if no PIN is used for sealing the disk encryption key, unsealing entirely depends on the current values of the PCRs. We can assume that the TPM as a trust anchor



Protocol 3: Unsealing

provides tamper protection against logical and physical attacks — this is why we store the disk encryption key inside the TPM in the first place. Therefore we cannot assume the same level of tamper protection for the outside world, i.e., the firmware components that measure the system state and compute the hash values — otherwise there would be no need for a TPM and we could store the key K_{disk} directly outside from the TPM, e.g. in the firmware.

This means we want to protect the host system's firmware by measuring if it is in a good state by using the TPM, yet the process of *measuring* whether the system is in a good state relies on that very firmware. If firmware is modified, its hash values change. However, if it is modified in such a way that the known good hash values are sent to the TPM, this modification will go unnoticed. Second, as shown in Protocol 3, the communication between the host system and the TPM via the bus is in plaintext.

The TPM specification however allows certain command parameters — such as those of *unseal* — to be encrypted; this is dubbed *parameter encryption* [34]. Conceptually, we can distinguish between two session key establishment functions, a *bound session* and a *salted session* [33].

- In a bound session, the session is bound to the *authValue* of a specific TPM object. Knowledge of this value is required by the caller to successfully authorize operations involving the object. Both the host system and the TPM generate a random nonce each, and exchange this information. Conceptually, the session key is then derived as

$$\text{KDF}(\mathit{authValue}, \mathit{nonce}_{\mathit{caller}}, \mathit{nonce}_{\mathit{tpm}}).$$

- In a *salted* session, a random salt value is generated by the host system, encrypted with the public key of a TPM object capable of decrypting the salt. After receiving this encrypted salt, the TPM decrypts it with the corresponding private key, and, together with additional random nonces of the host system and the TPM, the session key is conceptually derived as

$$\text{KDF}(\mathit{salt}, \mathit{nonce}_{\mathit{caller}}, \mathit{nonce}_{\mathit{tpm}}).$$

  Using a salted session usually comes with the drawback of increased resource usage, as asymmetric crypto operations can become costly on resource-constrained devices, which the early boot stages of a computer system often depend on.

Regarding the security of parameter encryption for full-disk-encryption: In both cases, cryptographic session keys are established that enable confidentiality and integrity

protection of selected TPM command parameters between the host system and the TPM. For a bound session this implies that the authValue is either stored securely on the host-system, or the authValue is entered by the user (i.e. derived from PIN) whenever a bound session is established. In case of a salted session, the host-system must ensure that it uses the genuine public key of the TPM. Replacing the public key on the host-system by the attacker would enable a man-in-the-middle attack.

As mentioned, a bound TPM session derives its parameter encryption keys from the object's authValue. If an attacker obtains a complete session transcript and the authValue has low entropy, the attacker may be able to perform offline guessing by deriving candidate session keys and validating decrypted parameters. If we assume that an attacker is not able to obtain a complete session transcript, storing the disk encryption key inside the TPM sealed with a (low entropy) PIN provides a significant security advantage compared to systems without a TPM: If a user wants to protect a (high entropy) disk encryption key by e.g. key-encryption with a password, then either the user is required to enter a high-entropy password during each bootup, or using a low entropy PIN, the disk-encryption key would be susceptible to brute force attacks. TPMs implement anti-hammering techniques (Dictionary Attack protection), where failed authentication attempts result in exponentially increasing time-delays or trigger a hardware lockout state; because these lockout counters are maintained in non-volatile memory (NVRAM) within the TPM chip, the anti-hammering protection survives reboots and power cycles. Therefore, even an easy-to-remember low entropy PIN suffices to protect the disk-encryption key securely. Windows BitLocker as well as current-generation Linux distributions do not employ a PIN to seal the disk-encryption key by default, however.

While users have to enter a PIN or password during login for the operating system, this is not tied to the disk encryption key. A typical user expectation, however, might be that the encrypted drive is strictly tied to the login credentials, and that the data is safe if, e.g., a computer or hard disk is stolen. This expectation is not fulfilled in practice, as our attacks described in Section 3 illustrate.

2.2 Windows BitLocker Implementation

BitLocker is Microsoft's built-in FDE, available as *Device Encryption* on all current Windows editions. [1] BitLocker supports multiple *protectors* for unlocking drives, including TPM-only, TPM+PIN (an optional pre-boot authentication protector), and password-based recovery keys. [2] By default, the TPM-only configuration is used as the primary protector alongside a recovery key.

Every BitLocker-encrypted drive stores a Full Volume Encryption Key (FVEK) encrypted by a Volume Master Key (VMK). Each protector holds an encrypted copy of the VMK, enabling multiple unlock methods while allowing efficient revocation. Re-encrypting the FVEK with a new VMK is computationally far less expensive than re-encrypting the whole data.

The TPM-VMK is encrypted by the TPM and stored in the protector metadata as public and private blobs, alongside a PCR selection and PCR digest. The PCR selection indicates which registers are used for sealing and unsealing, by default `PCR#7` (Secure Boot measurements) and `PCR#11`; the latter being extended during boot to lock the TPM, preventing unsealing from a fully booted environment and thereby protecting the decryption key from running programs. The PCR digest is a hash over the selected PCR values, representing the expected system state.

[1] https://support.microsoft.com/en-us/windows/device-encryption-in-windows-cf7e2b6f-3e70-4882-9532-18633605b7df

[2] https://learn.microsoft.com/en-us/windows/win32/secprov/getkeyprotectortype-win32-encryptablevolume

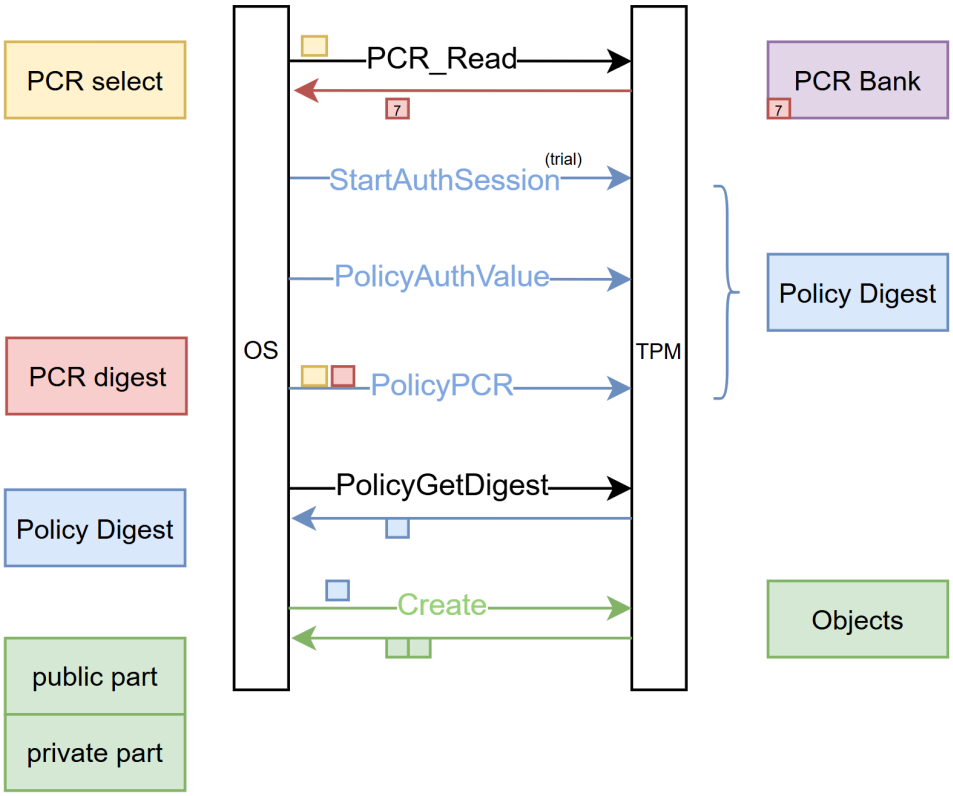


Figure 1: Sealing sequence of BitLocker

When setting up encryption with TPM, adding a TPM protector, or changing the configuration for a TPM protector, BitLocker has to seal the VMK with the TPM. Figure 1 shows the sealing process. First, BitLocker requests the current PCR values for the PCR selection, by default `PCR#7` via the `PCR_Read` command. The response is the base for the "trusted" state, and the respective value(s) must be matched during unsealing. In the default configuration, `PCR#11` is not fetched from the TPM, because during BitLocker setup it was already "locked". An empty `PCR#11` is required for unsealing. Then, a trial session is created using `StartAuthSession`; this session builds the *policyDigest* that reflects the trusted state. `PolicyAuthValue` is significant when setting up an optional TPM+PIN protector (see Section 4.1.3); in default TPM-only setups, the *authValue* is empty. Nonetheless, the `PolicyAuthValue` modifies the *policyDigest* of the Trial Session either way and must be included in the unsealing process. Finally, the expected PCR values are baked into the *policyDigest* by calling `PolicyPCR`. `PolicyPCR` receives the PCR Selection, here `PCR#7` and `PCR#11`, and the PCR Digest, which is a hash over the value of `PCR#7` which was obtained by `PCR_Read`, and a zeroed value for `PCR#11`, representing the pre-locked state. With the *policyDigest* now fully reflecting the trusted state, including the optional PIN and the PCRs, it is fetched by BitLocker using `PolicyGetDigest`. Finally, this *policyDigest* is sent to the TPM to seal the VMK; `Create` then returns the public and private blobs which are stored in the metadata of the protector.

Figure 2 shows the unsealing sequence, prior to which the trusted state was reached by measuring with `PCR_Extend`. Initially, the private and public blobs are loaded into the TPM via `Load`. Next, the *policyDigest* must be recreated; `StartAuthSession` starts

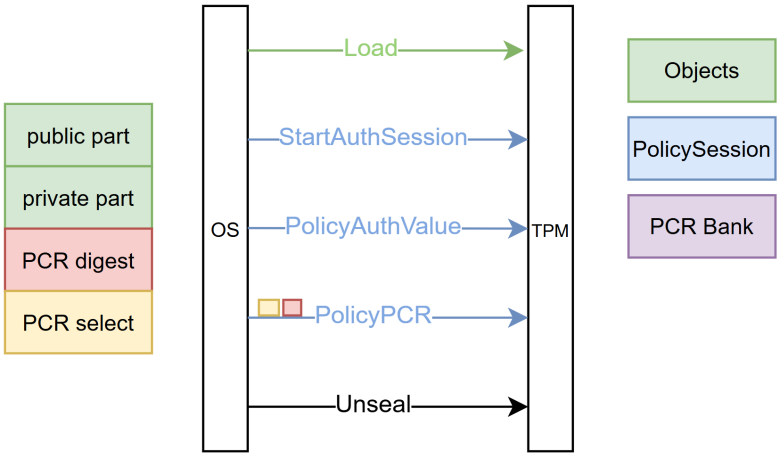


Figure 2: Unsealing sequence of BitLocker

a policy session, the empty *authValue* is provided by `PolicyAuthValue`, and `PolicyPCR` ensures the PCR values are reflected in the *policyDigest*. Should the system be in the trusted state, `Unseal` will return the VMK.

It is not possible for an attacker to simply send the expected PCR Digest to the TPM; during the unsealing sequence, a policy session is created rather than a trial session, which is required in order to unseal via `Unseal`. `PolicyPCR` works slightly differently; rather than extending the *policyDigest* with the **provided** PCR Digest, it calculates its own PCR Digest based on the provided PCR Selection and the current PCR Values in the PCR Bank. Should the provided and TPM-calculated PCR Digests differ, `PolicyPCR` will fail. If the provided PCR Digest matches the current state, but the current state is not the trusted state, `Unseal` will fail due to a mismatched *policyDigest*. The same happens if the PCR Selection is changed; this would alter the calculated PCR Digest, and consequently the *policyDigest*.

2.3 Linux Implementations

In recent years, Linux distributions have made significant progress toward integrating TPM-based disk encryption, drawing inspiration from Windows BitLocker. In the Linux ecosystem, Full-Disk Encryption is traditionally built on LUKS (Linux Unified Key Setup) and `dm-crypt`. Several frameworks and tools have evolved to handle TPM-based key management, ranging from distribution-specific tools to standardized implementations within `systemd`.

Below, we detail three key Linux implementations: `Clevis`, `systemd-cryptenroll`, and custom setups using `tpm2-tools` in the `initramfs`.

2.3.1 Clevis

Clevis is a pluggable framework for automated decryption, widely used in enterprise Linux distributions like Fedora, RHEL, and Ubuntu. Clevis binds a LUKS volume to a TPM 2.0 chip using the `clevis-luks` plugin via the `tpm2` pin, which relies on `tpm2-tools` under the hood. During binding, Clevis generates a high-entropy secret, seals it with the TPM under a specified PCR policy (typically `PCR#7`), and stores the resulting encrypted object

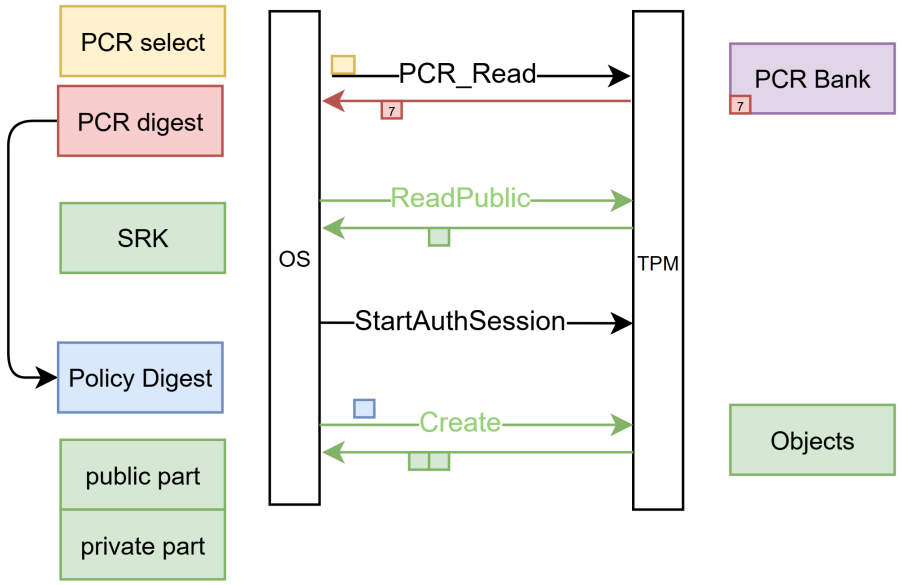


Figure 3: Sealing sequence of systemd-cryptenroll

(a JSON Web Encryption blob) in a LUKS header token slot. The secret itself is configured to unlock a dedicated LUKS keyslot holding the volume master key. During boot, an `initramfs` hook calls Clevis to unseal the secret from the TPM when the live PCR policy matches, automatically unlocking the LUKS volume.

2.3.2 systemd-cryptenroll

With full support for TPM+PIN and parameter encryption enabled by default, `systemd-cryptenroll` implements a modern TPM-based full-disk encryption. Setting up pure TPM-based FDE using the default PCR selection of {`PCR#7`} can be achieved by the following commands:

```
sudo apt install -y tpm2-tools dracut
sudo systemd-cryptenroll --tpm2-device=auto /dev/vda3
sudo dracut -f
```

During sealing (cf. Figure 3), `systemd-cryptenroll` first reads the current PCR values via `PCR_Read` and computes the *policyDigest* entirely in software, unlike BitLocker, which uses a trial session on the TPM for this purpose. Since the *policyDigest* is a deterministic hash chain, it can be calculated on the CPU without TPM interaction. Next, the Storage Root Key (SRK) is read via `ReadPublic`, and a salted HMAC session is established via `StartAuthSession` using the SRK's public key for the salt exchange. This session provides parameter encryption for subsequent commands. The secret (in this case the LUKS volume key) is then sealed via `Create` under the SRK as the parent key, with the pre-computed *policyDigest* embedded in the object's *authPolicy*. The secret is encrypted by the HMAC session in transit, and the TPM returns public and private blobs that are subsequently stored on disk.

Figure 4 shows the unsealing sequence of a default `systemd-cryptenroll` setup. Like with BitLocker (cf. Section 2.2), the public and private parts of the decryption key are loaded into the TPM with `Load`. Unlike BitLocker, `systemd-cryptenroll` uses two sessions during unsealing. An HMAC session, created via `StartAuthSession` and salted with the TPM's asymmetric key, establishes a secure channel for parameter encryption of the returned secret from `Unseal`. Because the random salt is encrypted with the

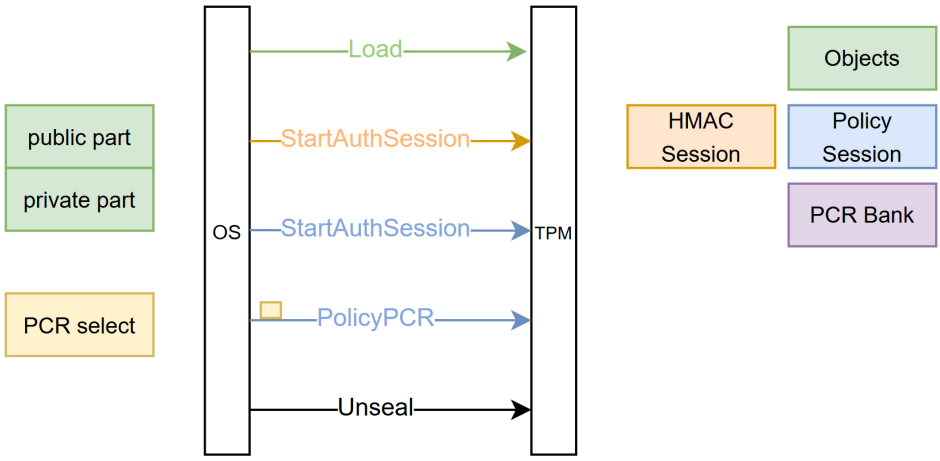


Figure 4: Unsealing sequence of `systemd-cryptenroll` with parameter encryption

TPM's asymmetric public key during session setup, only the TPM (holding the matching private key inside its hardware boundary) can decrypt the salt to derive the session key, establishing a protected transport channel between the TPM and host CPU that shields plaintext key material from passive bus sniffing. A second policy session attests the system state. Without a configured PIN, `PolicyAuthValue` is omitted; only `PolicyPCR` modifies the policy session, extending it with the PCR digest calculated from the live PCR values.

2.3.3 initramfs and `tpm2-tools`

The `tpm2-tools` suite allows for direct interaction with the Trusted Platform Module (TPM). While most automated solutions default to `PCR#7` (the Secure Boot policy) to seal the disk encryption key, `tpm2-tools` enables the selection of a custom combination of Platform Configuration Registers (PCRs) to measure additional boot stages. Once the key is sealed, a custom Bash script can be written to execute `tpm2_unseal` via `tpm2-tools` to retrieve the disk encryption key, unlock the LUKS volume, and proceed with the boot process. This script is subsequently added to `initramfs`. Since `tpm2-tools` supports all TPM 2.0 features, it is also possible to activate parameter encryption with, e.g., a salted session to avoid exposing the disk encryption key in plain on the bus.

Using a custom set of PCRs allows extending protection beyond Secure Boot to cover OS-level components like the kernel command line or the `initramfs` image. However, as discussed in Section 4.2.2, expanding the PCR policy beyond `PCR#7` introduces a significant trade-off between protection against early-boot tampering (cf. Section 3.2.2) and update friction during routine system maintenance.

3 Attacks

In this section, we describe various attack vectors for accessing the disk encryption key. We distinguish between *passive attacks*, where the attacker only monitors bus communication between the host CPU and the TPM without altering signals, and *active attacks*, where the attacker physically tampers with the host platform or bus lines to intercept, modify, or inject custom TPM messages.

3.1 Passive Attacks

If the bus communication is unencrypted and physically accessible, a logic analyzer can be used to capture interactions, extract the disk encryption key, and decrypt the hard disk. This attack has been particularly highlighted in the context of Windows BitLocker. As noted, BitLocker does not require a PIN by default. Consequently, during boot, the unsealing of the disk encryption key relies entirely on the values of the PCRs. If the UEFI or other low-level firmware remains unaltered, the measurements and PCR values will match. Access to the running system is controlled by the operating system's user login interface; however, on Windows 11 with PIN-based login, the login credentials are also protected by the TPM, making them more difficult to bypass.

Once the correct PCR values are provided, the Windows bootloader unseals the VMK using the `TPM2_Unseal` command. The VMK then appears in clear text within the `TPM2B_SENSITIVE_DATA` structure of the response. In BitLocker setups, the VMK is a 256-bit AES key starting at offset `0x10`. The VMK can then be used to decrypt the FVEK, which in turn decrypts the volume. It is important to note that this attack vector is not unique to BitLocker but applies to any system—including Linux—where unsealing depends solely on the correctness of the PCRs, parameter encryption is not used, and bus communication is physically accessible. Several countermeasures exist to mitigate this attack. The bus can be protected physically, proprietary encryption mechanisms can be used, or a design can require a PIN to be entered during boot to allow TPM unsealing. The most straightforward approach is to use parameter encryption [34, 33], which encrypts sensitive parameters (such as the unsealed key bytes) during transit over the bus using a session key. As mentioned, `systemd-cryptenroll` activates parameter encryption by default, providing a significant security advantage compared to all other implementations.

3.2 Active Attacks

Against systems with protected TPM communication, e.g. through physical hardening or parameter encryption, passive bus sniffing is ineffective. In such cases, active attacks become relevant. Although conceptually more complex, our experiments show that they may even be easier to execute in practice.

3.2.1 State Forgery Attack

The security of TPM-based FDE relies on the assumption that the PCRs truthfully reflect the state of the system; correct PCRs meaning a trustworthy state. State forgery attacks create a seemingly trustworthy state in an untrustworthy environment. Once the trusted state is forged, attacks against e.g. FDE can easily be mounted. The challenging part of the attack is to manipulate the PCRs, due to two characteristics of the hash-based PCRs: First, there is no intended way to directly write to or reset the PCR values after startup. The only way to change a PCR is by measuring and extending the PCR, building a chain of measurements. Second, due to the nature of the hash function used, any extension is generally one-way; finding a measurement such that another can be reversed is computationally infeasible.

Forward Extension and Divergence A known set of measurements that can extend a specific PCR from A to B is called a forward extension from A to B. If there is no known set of measurements to extend from C to B, then we call C divergent from B. If the reverse is also true, then B and C are divergent from each other.

Figure 5 shows this concept with the example of booting different OSs, affecting `PCR#7`. From startup, there is a forward extension to the pre-OS stage (reflecting early firmware and Secure Boot state). From here, both Windows' Bootmgr and Linux's GRUB forward

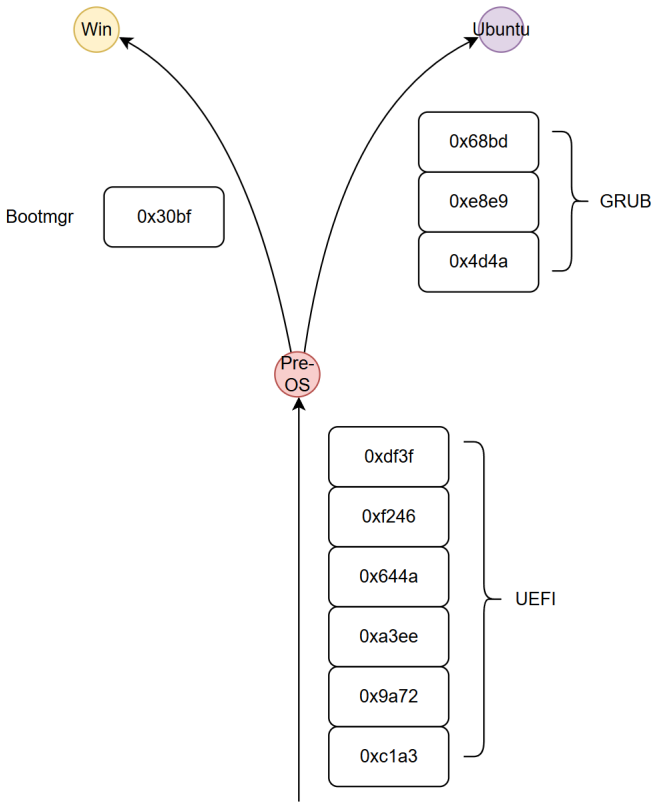


Figure 5: Divergence between Windows and Ubuntu on `PCR#7`

extend with different measurements during their boot process. Windows and Ubuntu finally are divergent from each other, as reaching one state from the other would require reversing the respective measurements during the bootloader's stage. This scenario is exactly what TPM-based FDE intends to secure; an encrypted disk cannot simply be decrypted by booting into a different environment.

Based on Figure 5, booting into a different Windows installation, of which the attacker knows the OS-level password, would be a feasible attack path. This is true for `PCR#7`, as the same version of Windows, and the same Secure Boot configuration of the platform, will measure the same values for `PCR#7`. As mentioned in Section 2.2, the default configuration of BitLocker defines the truthful state through `PCR#7` and `PCR#11`, with the latter being required to be zero, but being "locked" during boot. Once a different Windows environment is booted, `PCR#11` is already fully divergent from the zeroed state BitLocker requires. Presumably, this is why `PCR#11` is always measured into by Windows Bootmgr, even on systems where BitLocker is not enabled; to prevent simply mounting an attack from an attacker controlled Windows installation.

Zero State To forward extend to a trusted state, all required PCRs must be forward extended. As such, the difficulty of a state forgery attack depends heavily on the PCR selection by the FDE implementation. In order to reliably mount an attack against any target, especially considering the range of implementations and configurations possible on Linux, one requires a state that can be forward extended into any other state, across all PCRs; the **zero state**, i.e., zeroed values for every PCR index.

Trivially, there's always a zero state at initial hardware power-on when starting up

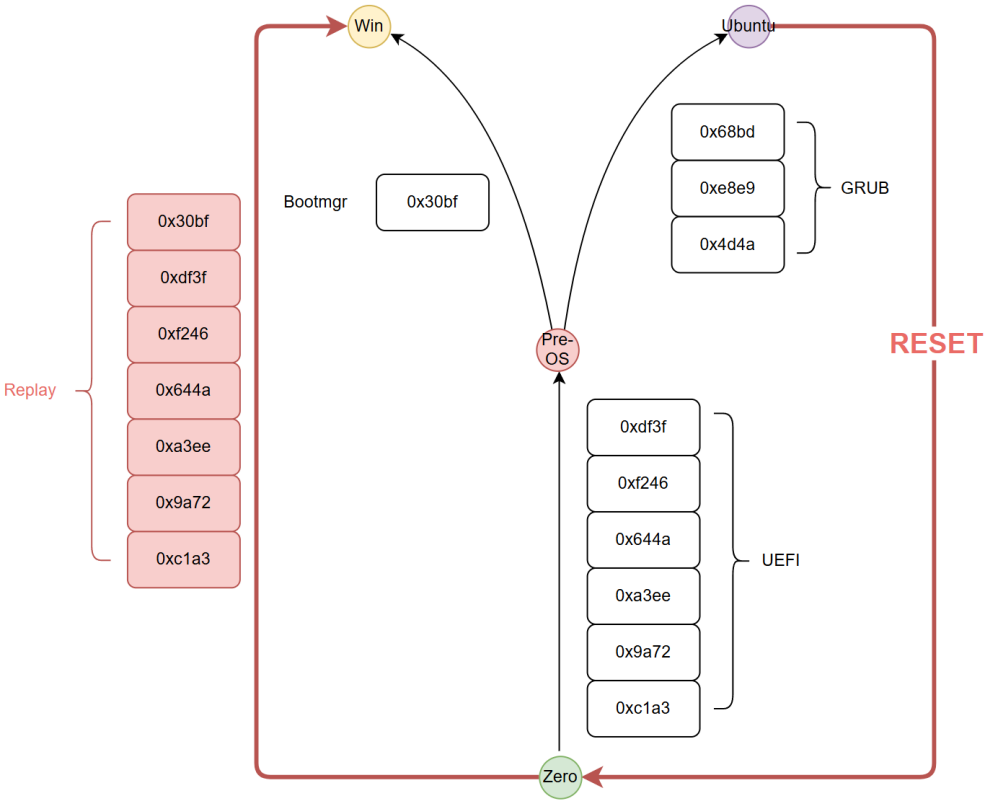


Figure 6: State Forgery Attack against Windows BitLocker using a reset attack induced zero state (PCR#7)

the TPM. In the context of active attacks, the zero state refers specifically to an attacker-controlled environment (e.g., live Linux) initialized with zeroed PCRs while maintaining a fully functional interface to the TPM chip. There are conceptually two ways to achieve a zero state at that moment; resetting the PCRs after divergence, or preventing divergence altogether, i.e. preventing any kind of measurement. Against the design principle of the TPM, exploits that lead to a reset of the TPM have been demonstrated, notably [19], which abused a faulty restore of PCR values after an S3 sleep state. Further, we demonstrate multiple ways to physically reset the TPM while retaining system functionality and the interface to the TPM in Section 5. Alternatively, Section 5.2.3 describes a firmware manipulation attack against TianoCore EDK2.

State Forgery Attack against BitLocker By mounting a reset attack (cf. Section 5) an attacker Linux system first resets the TPM, and further can replay the respective PCR measurements to forward extend the PCRs to the trusted state of the target Windows BitLocker system, as seen in Figure 6.

In order to abuse the zero state achieved by a reset or firmware manipulation attack, an attack needs a few key pieces of data; the set of measurements to forward extend the PCR from zero state to the trusted state, the private and public parts of the key, as well as the PCR selection. While the PCR digest is not required for the unsealing sequence, it can be used to calculate the subset of PCR measurements that is fulfilling the forward extension.

The parts of the key, as well as the PCR selection and PCR digest, are stored in the

```
→ sudo mmls /dev/nbd0
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors

     Slot      Start        End          Length       Description
000:  Meta     0000000000   0000000000   0000000001   Safety Table
001:  -------  0000000000   0000002047   0000002048   Unallocated
002:  Meta     0000000001   0000000001   0000000001   GPT Header
003:  Meta     0000000002   0000000033   0000000032   Partition Table
004:  000      0000002048   0000206847   0000204800   EFI system partition
005:  001      0000206848   0000239615   0000032768   Microsoft reserved partition
006:  002      0000239616   0082151423   0081911808   Basic data partition
007:  004      0082151424   0132962303   0050810880
008:  003      0132962304   0134213631   0001251328
009:  -------  0134213632   0134217727   0000004096   Unallocated
```

Figure 7: Finding the sector of the BitLocker partition.

BitLocker metadata. The utility `dislocker` [5] can be used to parse the metadata. For this, the offset of the BitLocker partition has to be calculated first. Using `mmls` from `sleuthkit` [10], the partitions can be listed; the BitLocker partition is labeled "Basic Data Partition" (see Figure 7). The start sector must be multiplied by 512 to get the offset. With the offset, `dislocker` can be run:

```
sudo dislocker-metadata -V /dev/nbd0 -o offset > output.txt
```

Figure 8 is the relevant part of the `dislocker` output; the "TPM_ENCODED" VMK. The payload contains two bytes representing the size of the private keyblob, followed by the keyblob, then size of and public key. Again, two bytes for the size of the PCR digest followed by the digest, and finally one byte for the size of the PCR Selection, followed by the PCR selection bitmask.

In order to obtain the PCR measurements needed, all PCR measurements must first be logged. If available, they can be measured with the Windows program `tpmtool` [23] using the `gatherlogs` subcommand. The resulting file can be parsed by the `tpm2_eventlog` utility of the `tpm2-tools` [31] suite. This will output a list of all measurements recorded during the current power cycle.

In order to calculate the exact subset of extend commands needed, all measurements into the PCRs that are in the PCR selection need to be evaluated in sequence. Replaying all logged events blindly is often insufficient because event logs record both pre-OS firmware events and OS-stage events; for instance, replaying past BitLocker's sealing checkpoint into the OS boot phase extends `PCR#11` past its zeroed state, locking the TPM and causing unsealing to fail. Therefore, a simulation of a PCR bank is created, extending, one by one, every measurement in the list. After every measurement, the PCR values from the PCR selection are hashed to compute the simulated PCR digest. This PCR digest is compared to the expected PCR digest retrieved from the BitLocker metadata. Once they match, the exact cutoff point for the required measurement subset is identified, as shown in Figure 9. In case of the current state **not** being a zeroed state, but another state that can be forward extended to the trusted state, an analogous calculation has to be done for the start of the subset of measurements: first, after every measurement into the simulation, the PCR values of the selection must be compared to the live PCR values. Once the simulator is in the equal state, the start of the subset of measurements is determined.

Finally, all necessary information is available, and the unsealing sequence can be executed, after the PCR values are forward extended with the subset of measurements. The sequence is identical to the sequence that BitLocker implements (see Figure 2). For this, the relevant TPM 2.0 commands have been re-implemented in binary, as an example, see

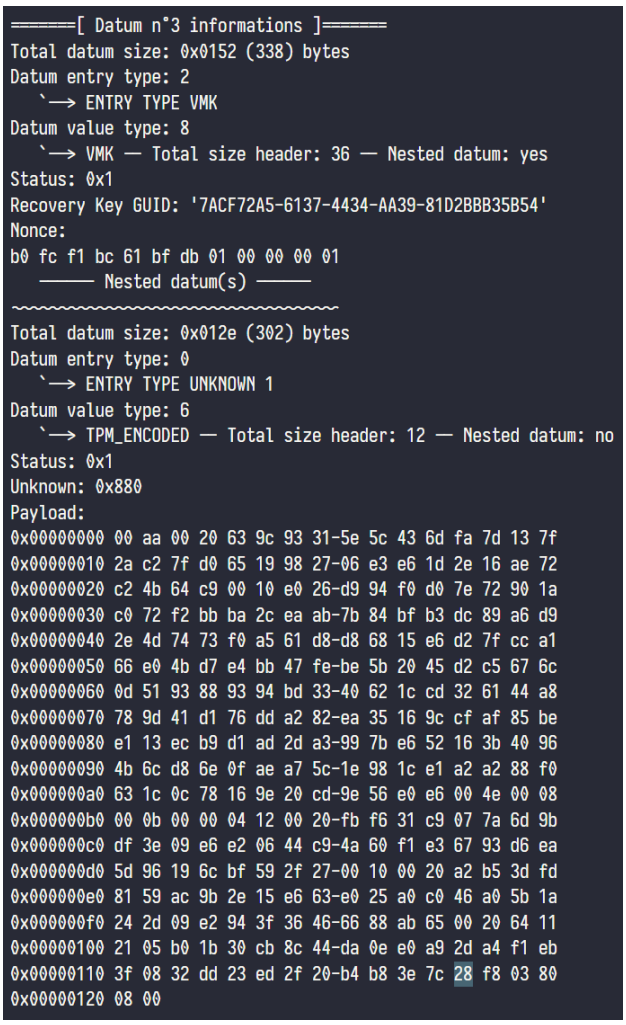


Figure 8: Dislocker Output, the payload contains the keyparts, the PCR Digest, and the PCR Selection.

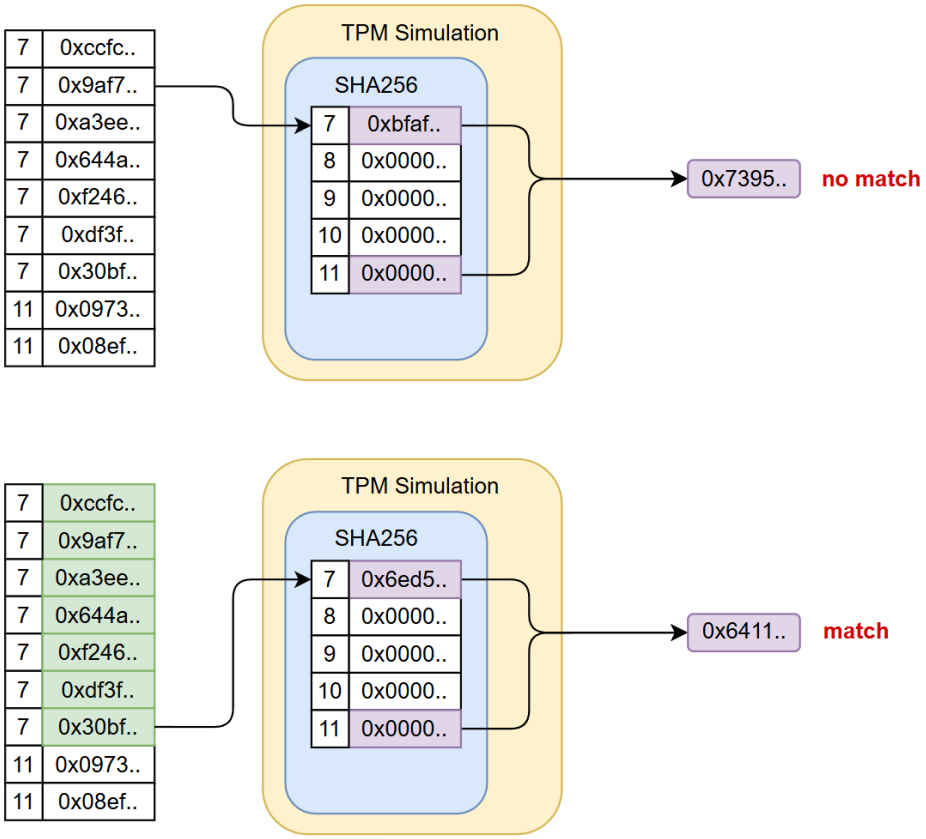


Figure 9: Calculating the required extends by comparing the simulated PCR digest with the trusted state PCR digest obtained from the metadata.

```python
def policy_pcr(pcr_digest_hex, pcr_selection_bitmap="150800", session_handle=0x03000000, hash_alg="SHA256"):
    pcr_digest_size = len(pcr_digest_hex) // 2
    builder = [
        "8001",                                                                  # TPM_ST_NO_SESSIONS
        (14 + 2 + pcr_digest_size + 4 + 2 + 1 + 3).to_bytes(4, byteorder='big').hex(),  # total size
        "0000017f",                                                              # TPM_CC_PolicyPCR
        session_handle.to_bytes(4, byteorder='big').hex(),                       # sessionHandle
        pcr_digest_size.to_bytes(2, byteorder='big').hex(),                      # pcrDigest size
        pcr_digest_hex,                                                          # pcrDigest
        "00000001",                                                              # TPML_PCR_SELECTION.count = 1
        hash_alg_ids[hash_alg].to_bytes(2, byteorder='big').hex(),               # TPMI_ALG_HASH (SHA256)
        "03",                                                                    # sizeofSelect = 3 bytes
        pcr_selection_bitmap                                                     # 3 bytes = 6 hex digits
    ]
    return "".join(builder)
```

Figure 10: Implementation of the PolicyPCR command

Figure 10. Importantly, the TPM will generate fresh nonces, which have to be considered. For caller nonces, static data can be used. For the `Unseal` command, an HMAC is required; the HMAC is calculated as follows:

$$\begin{aligned}&\mathbf{HMAC}(\emptyset, \mathrm{SHA256}(\mathrm{commandCode} \parallel \mathrm{name}) \parallel \\ &\mathrm{nonce}_{\mathrm{caller}} \parallel \mathrm{nonce}_{\mathrm{tpm}} \parallel \mathrm{sessionAttributes})\end{aligned}$$

Because the session is unsalted and unbound (since no *authValue* PIN is configured), the HMAC key is $\emptyset$ (denoting a zero-length key), but notably the *sessionAttributes* structure is not empty, but contains a single zero byte (`0x00`).

The unsealing sequence will result in a returned VMK, which can be used by `dislocker` to decrypt the BitLocker volume. It also should be noted that depending on the chosen Linux distribution, the kernel might interfere with the unsealing sequence, by closing handles as a form of garbage collection.

State Forgery Attack against `systemd-cryptenroll` Although `systemd-cryptenroll` uses parameter encryption to prevent passive bus sniffing attacks, parameter encryption alone does not protect against active state forgery attacks. Because the attacker initiates the unsealing session from their own software, they obtain the ephemeral session keys used to decrypt the returned payload on the bus. Furthermore, because parameter encryption is negotiated per-session by the caller, an attacker mounting a state forgery attack can simply choose not to request parameter encryption during session setup.

Similar to BitLocker, `systemd-cryptenroll` creates a LUKS header including the blob (combined), the PCR selection (`PCR#7` by default), and the policy digest. Note that, in order to extract the PCR subset of measurements for the forward extension, the calculated PCR digest must be used to calculate a would-be *policyDigest*, which needs to be compared to the *policyDigest* from the header. For the default configuration of `PCR#7`, this is not a concern since that PCR is assigned under the purview of the UEFI, and all measurements should be completed by the time `systemd-cryptenroll` is executed, and therefore included in the subset:

```
sudo cryptsetup luksDump --dump-json-metadata /dev/sdb3 \
    | jq '.tokens[] | select(.type=="systemd-tpm2" \
    or .type=="tpm2")'
```

Since `systemd-cryptenroll` internally uses `tpm2-tools`, using `tpm2-tools` for the attack is convenient. An attack can be mounted as simple as this:

```
tpm2_pcrextend 7:sha256=e8e9...1016
```

```
...
tpm2_pcrextend 7:sha256=68bd...89cd
tpm2_load -C 0x81000001 -u sealed.pub -r sealed.priv \
    -c sealed.ctx
tpm2_unseal -c sealed.ctx -p pcr:sha256:7 | xxd
```

First, the PCRs need to be forward extended to the trusted state, using `tpm2_pcrextend`. Then, using the blobs, a context handle is created using `tpm2_load`, using the SRK as a parent key, here `0x81000001`. Finally, `tpm2_unseal` implicitly creates a policy session to satisfy the requirements for the unseal by adding the `-p pcr:sha256:7` flag, to indicate that the state of `PCR#7` with the hash algorithm SHA256 must be trusted. `xxd` then displays the binary secret in human-readable form. Unlike BitLocker, this secret is not the decryption key. In order to maintain compatibility with `homed`, the binary secret is base64-encoded and then used as a passphrase [28]. With this passphrase, the volume can be decrypted and mounted:

```
base64 key.bin > key.b64
lsblk -f /dev/sdb0
cat key.b64 | sudo cryptsetup luksOpen /dev/sdb0p3 \
    cryptroot
sudo vgscan
sudo vgchange -ay
sudo lvs
sudo mount /dev/mapper/ubuntu--vg-ubuntu--lv /mnt
```

State forgery attack against other implementations A state forgery attack involving a replay attack against disk encryption with `tpm2-tools` and `initramfs` is conceptually the same as against `systemd-cryptenroll`: obtain the desired hash values and PCR extensions, reset the TPM into the zero state, and then replay the extensions such that the desired PCR values are reached. Once the policy matches, the attacker can obtain the disk encryption key from the TPM. Conceptually this also works against Clevis, but as Clevis does not implement parameter encryption, a passive attack works as well in that scenario.

Man-in-the-Middle (MitM) Attacks In contrast to passive attacks, the attacker is not limited to merely monitoring communication. As mentioned, to mitigate bus sniffing, the TPM 2.0 specification supports parameter encryption using session-based encryption. This ensures that sensitive data (like the unsealed key) is encrypted before it is transmitted over the bus. However, this protection is susceptible to a sophisticated MitM attack if the session is not properly authenticated via a trusted, pre-established key.

In a targeted attack against a salted session, the attacker interposes themselves between the host system and the TPM. When the host attempts to establish a secure session, it generates a session key based on a salt and a public key provided by the TPM. If the attacker can intercept this exchange, they can provide a *fake public key* to the host system. The host, believing it is communicating with a legitimate TPM, encrypts the salt using the attacker's key. The attacker then decrypts the salt, re-encrypts it using the actual TPM's public key, and forwards it to the chip. This allows the attacker to derive the same session key, maintaining the illusion of a secure, encrypted session while successfully performing a transparent interception of the disk encryption secrets upon unsealing. To prevent this, the host system would need to verify the public key, e.g., using the public key infrastructure of the TPM manufacturer, which is difficult to securely implement at this early boot stage.

3.2.2 Initramfs Attacks

A critical vulnerability in TPM 2.0-based disk encryption arises when the sealing policy of the disk encryption key fails to encompass the integrity of the initial RAM filesystem (*initramfs*). This attack was reported as `CVE-2016-4484`. While UEFI Secure Boot ensures

the authenticity of the bootloader and the kernel (typically measured in `PCR#7`), many standard configurations neglect to include `PCR#9`, which captures the measurement of the *initramfs* image.

An attacker with brief physical access can exploit this gap by intentionally triggering a boot failure—such as entering an incorrect decryption passphrase—to drop into a root-level debug shell. From this shell, or by mounting the `/boot` partition via external media, the attacker can unpack *initramfs* using tools like `unmkinitramfs`, inject a malicious script designed to exfiltrate the disk encryption key (e.g., printing it to the console or a serial port), and repackage the archive. Because *initramfs* is not part of the sealing policy, the TPM will still unseal the decryption key during the subsequent boot: the Secure Boot state (`PCR#7`) remains valid, and the kernel (`PCR#8`) remains untampered, but the malicious payload within the unmeasured `PCR#9` executes with full privileges immediately after the disk is decrypted. While measuring `PCR#9` effectively mitigates this attack, it introduces significant operational overhead, as any legitimate system update that regenerates *initramfs* will alter its hash and prevent the TPM from unsealing the key, necessitating a complex re-sealing process.

3.2.3 TPM Relocation Attack

Another physical attack involves the relocation of a TPM from a target system to an attacker-controlled platform, a method known as a *TPM Relocation Attack*. In this scenario, the attacker removes the TPM chip from the victim's motherboard and attaches it to a secondary "interposer" host system that is entirely under the attacker's control.

The objective of this attack is to decouple the cryptographic secrets stored within the TPM from the legitimate integrity state of the original hardware. Because the TPM is a passive component that merely responds to commands, it cannot inherently verify whether the measurements (PCR extensions) it receives are coming from a legitimate, untampered CPU or an attacker's emulator. On the controlled host, the attacker can perform one of the previous state forgery attacks of the entire boot sequence. By feeding the TPM the exact sequence of legitimate hashes (PCR values) that would have been generated by the original, untampered system, the attacker satisfies the TPM's sealing policy. Once the TPM unseals the disk encryption key under the false pretense of a healthy boot state, the attacker can extract the key from the controlled host's memory, effectively bypassing all platform-level protections.

4 Countermeasures

While the active and passive attacks described in Section 3 and experimentally demonstrated in Section 5 highlight significant vulnerabilities in TPM-backed Full Disk Encryption (FDE), several countermeasures can mitigate these risks. Defending against these threats requires addressing two distinct layers: securing the physical communication channel between the CPU and the TPM (Section 4.1), and preserving measurement integrity against firmware tampering and state manipulation (Section 4.2). Finally, Section 4.3 examines the fundamental trust asymmetries, certification mismatches, and emerging hardware protections across modern platforms.

4.1 Mitigating Physical and Bus Attacks

4.1.1 Parameter Encryption and Bus Security Protocols

The Trusted Computing Group (TCG) explicitly recommends parameter encryption to mitigate passive bus sniffing. By establishing a secure salted session or bound session, parameter encryption creates an encrypted tunnel between the host CPU and the TPM [33].

However, while parameter encryption is highly effective against passive attacks, it provides **no protection** against active state forgery attacks, where the attacker controls the host environment completely. Because the attacker is the one initiating the `StartAuthSession` command, they can opt to not use parameter encryption. Parameter encryption secures the transport layer, but it does not authenticate the caller.

Parameter encryption also cannot be used to prevent access to the measurements; it can encrypt only commands and responses whose first parameter has an explicit size field. As the payload of `TPM2_PCR_Extend` must be parsed procedurally, the length is not immediately determined, thus making encryption of the measurements using parameter encryption impossible [34].

To counter active bus interposition on discrete TPMs, TCG guidance recommends establishing authenticated sessions bound to a platform-provisioned host secret [35]. Furthermore, efforts are underway to introduce transport-layer payload encryption using the Security Protocol and Data Model (SPDM): while the TCG Server Work Group published a specification for server environments [37], standardization for PC client platforms remains an ongoing effort within the TCG PC Client Work Group. In parallel, commercial implementations such as HP TPM Guard [20] have begun deploying SPDM-based bus encryption with platform-bound keys on selected client hardware. As we analyze in Section 4.3, however, link-layer encryption introduces fundamental architectural challenges regarding session endpoint trust and certification boundaries.

4.1.2 Firmware TPMs (fTPMs)

Mitigating active physical attacks can also be achieved by shifting the root of trust deeper into the hardware [39]. Utilizing a firmware TPM (fTPM), such as AMD's fTPM or Intel's PTT, integrates the TPM into the main CPU silicon. This eliminates the exposed SPI bus (defeating Sniffing Attacks) and removes the physical reset pin (defeating Reset Attacks).

However, migrating to an fTPM does not eliminate the threat of active attacks; it merely shifts the attack surface to host firmware and the processor itself. First, as our own experiments demonstrate in Section 5.2.3, suppressing measurements in UEFI boot code bypasses PCR attestation on fTPMs just as effectively as on dTPMs, because an fTPM relies on the host firmware to honestly submit measurements. Second, discrete TPMs typically feature certified, tamper-resistant packaging, whereas fTPMs share the host processor silicon and lack dedicated physical countermeasures, exposing them to voltage fault injection attacks such as *faulTPM* [21] and microarchitectural side channels [24].

4.1.3 TPM+PIN

As noted in Section 2, default FDE installations rely solely on a TPM-only protector to provide seamless, unattended boot. However, enabling an optional combined TPM+PIN protector serves as a powerful countermeasure against both passive and active attack vectors.

With the `systemd-cryptenroll` tool, enabling the TPM+PIN feature requires the flag `-tpm2-with-pin=yes` for the `systemd-cryptenroll` command, which prompts for a PIN during setup.

Figure 11 shows the unsealing sequence of `systemd-cryptenroll` with a TPM+PIN protector; unlike a sole TPM protector, the command `PolicyAuthValue` is additionally used. This command extends the *policyDigest* and sets the *isAuthValueNeeded* flag to require the *authValue* of the object for the HMAC calculation, creating a bound session.[3] During the `Unseal` command, due to the set flag, the HMAC calculation requires both the ephemeral secret of the parameter encryption, as well as the *authValue*:

[3]Furthermore, the session is salted due to the enabled parameter encryption.

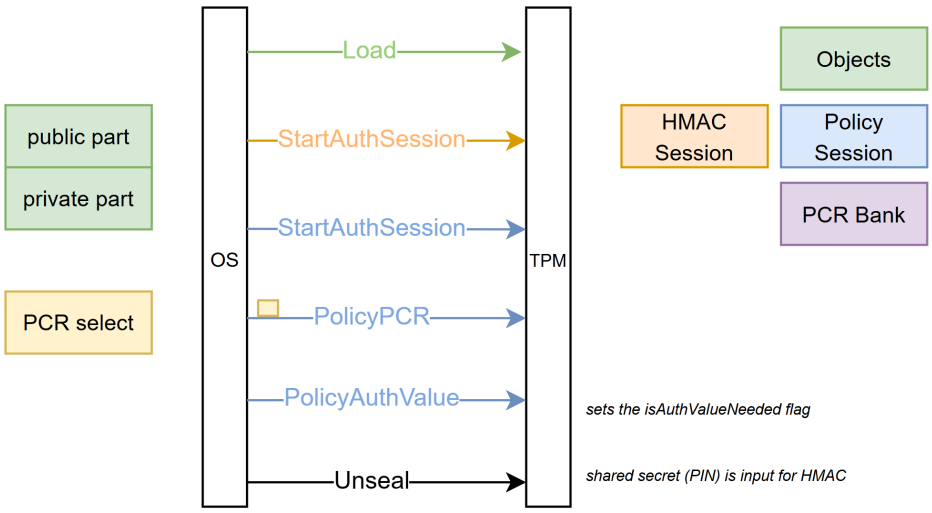


Figure 11: Unsealing sequence of `systemd-cryptenroll` with parameter encryption and TPM+PIN protector enabled

$$\textbf{HMAC}(\text{sessionKey} \parallel \text{authValue}, \text{SHA256}(\text{commandCode} \parallel \text{name}) \parallel \text{nonce}_{\text{caller}} \parallel \text{nonce}_{\text{tpm}} \parallel \text{sessionAttributes})$$

Because `PolicyAuthValue` extends the *policyDigest* by `TPM_CC_PolicyAuthValue`, an attacker may not omit the command, as satisfying the policy will be otherwise impossible.

Parameter encryption must be enabled nonetheless to protect against passive attacks.

Windows BitLocker differs conceptually from this, as well as the TCG guidance (cf. Section 2.2); while parameter encryption is never supported, the TPM sealed secret is, in the case of the TPM+PIN protector, **not** the plain VMK. Alternatively, BitLocker itself encrypts the VMK with a key derived from the user PIN and a nonce, and seals that encrypted VMK blob in the TPM. This way, when using a PIN with BitLocker, passive attacks are also mitigated, as the unsealed VMK is no longer sent in the clear over the bus. As with `systemd-cryptenroll`, the PIN is also used during pre-boot authentication, mitigating active state forgery attacks.

In order to enable the TPM+PIN protector under Windows, the administrator must create a group policy and enable "Require Startup PIN with TPM", and disable all other options, notably set "Do not allow TPM".

Despite its security benefits, TPM+PIN introduces user friction by requiring pre-boot PIN entry during boot before the volume key can be unsealed, followed by standard OS-level user authentication upon reaching the login desktop.

4.2 Mitigating Firmware and Measurement Attacks

The attack on TianoCore EDK2 shown in Section 5 is based on the assumption that boot firmware (i.e. UEFI components) can be modified. Other attacks described in Section 5 exploit the fact that the environment outside the TPM can be manipulated. Such modifications can occur via software attacks or physical tampering, with the latter being often the less explored attack vector. Hardening against physical tampering is known to be

a demanding problem, but often not in scope when designing threat models for firmware attacks. In this section we describe existing technologies to harden against software and physical firmware attacks and how these affect full disk encryption with TPM2.

4.2.1 Platform Roots of Trust: SRTM and DRTM

Technologies like Intel Boot Guard or AMD Hardware Validated Boot (HVB) establish a Hardware Static Root of Trust for Measurement (H-SRTM), distinguishing them from older, firmware-based Core Roots of Trust for Measurement (CRTM). By storing OEM public key hashes in one-time programmable hardware fuses, CPU microcode cryptographically verifies the initial boot block before execution. Attackers cannot trivially patch out the `HashLogExtendEvent` functions (Firmware Tampering Attack), as the CPU halts the boot process if the firmware signature is invalid.

However, the real-world effectiveness of H-SRTM relies on several assumptions that can break down in practice. First, desktop motherboards and embedded systems frequently ship with Boot Guard unprovisioned or in non-enforcing modes, allowing firmware to be rewritten directly via external flash clips. Second, parser vulnerabilities such as *LogoFAIL* [7] allow execution to be hijacked during boot without modifying signed binaries. Finally, compromised signing keys (such as the 2023 MSI data breach [9]) or devices shipped with default vendor test keys (*PKfail* [8]) completely undermine cryptographic verification.

To eliminate reliance on large, complex pre-boot UEFI firmware stacks, the TCG defines the Dynamic Root of Trust for Measurement (DRTM) architecture [32], commonly implemented via Intel Trusted Execution Technology (TXT) and AMD Secure Launch. Rather than trusting the entire pre-boot environment, DRTM establishes a chain of trust dynamically at operating system launch. By executing a hardware-enforced launch mechanism, the CPU isolates its execution environment and resets the dynamic PCRs (`PCR#17`–`PCR#22`) to a hardware-enforced initial state, clearing any prior state from the pre-boot phase. The CPU then measures a minimal, hardware-isolated loader directly into protected cache, bypassing the entire pre-boot UEFI stack.

For Full Disk Encryption, DRTM provides a critical defense against firmware tampering. Modern implementations such as Windows Defender System Guard Secure Launch for BitLocker, as well as Linux DRTM initiatives like TrenchBoot, can seal disk encryption keys to these dynamic PCRs rather than static firmware registers. Because dynamic PCRs can only be reset and populated through this hardware-enforced late launch, an attacker who subverts or suppresses pre-boot UEFI measurements cannot spoof the expected state, effectively preventing unauthorized key unsealing even on platforms with compromised host firmware. Once the keys are unsealed, the operating system relies on continuous hardware protections such as DMA protection via an IOMMU and CPU memory isolation to prevent unauthorized access from peripherals or unmeasured firmware contexts.

Fundamentally, however, DRTM represents a pragmatic concession and a departure from the original design philosophy of measured boot. The foundational vision of SRTM relied on an unbroken chain of transitive trust from platform reset, operating under the assumption that the pre-boot environment could serve as an auditable, verifiable foundation. As modern UEFI firmware expanded into a complex codebase with extensive third-party supply chains, verifying the entire pre-boot stack became practically infeasible. By establishing a late root of trust, DRTM abandons the objective of verifying the complete platform state in favor of isolating specific runtime payloads. While this reliably protects targeted operations such as FDE key unsealing, it provides a narrower security guarantee: it isolates sensitive secrets from compromised firmware rather than ensuring that the underlying host platform itself is in a trusted state.

Furthermore, DRTM does not resolve the underlying vulnerabilities of the TPM implementation itself, while introducing its own attack surface. On platforms equipped with

a discrete TPM, DRTM fails to mitigate physical bus attacks: the `TPM2_Unseal` response still traverses the exposed SPI bus in plaintext, leaving key extraction via bus sniffing viable unless paired with parameter encryption or a PIN. Conversely, when deployed on systems with a firmware TPM, DRTM inherits all the physical and architectural limitations of fTPMs, remaining vulnerable to processor voltage fault injection (*faulTPM* [21]) and microarchitectural side channels. Finally, the DRTM launch mechanism operates as a proprietary black box: it relies on closed-source vendor launch modules and CPU microcode that have historically suffered from implementation vulnerabilities, while facing systemic platform challenges such as persistent System Management Mode (SMM) handlers capable of bypassing CPU memory locks to extract unsealed keys directly from host RAM.

4.2.2 PCR Selection

Because State Forgery Attacks rely on manipulating the environment to match a required set of PCRs, the difficulty of the attack entirely depends on the FDE's PCR selection. However, selecting the correct PCRs is a highly complex balancing act between security and system stability.

Different PCRs fall under distinct operational purviews across the boot sequence:

- **Firmware & Platform State (`PCR#0`–`PCR#7`):** `PCR#0` measures core UEFI code, `PCR#1` holds host platform configuration, `PCR#4` tracks bootloader executables, and `PCR#7` attests the Secure Boot policy and certificates.
- **OS & Kernel State (`PCR#8`–`PCR#11`):** `PCR#8` captures kernel command line parameters, `PCR#9` measures the Linux kernel image and *initramfs*, and `PCR#11` is used by OS boot managers to explicitly lock key access post-boot.

Relying solely on firmware-level registers (`PCR#0`–`PCR#7`) leaves the TPM in an "unlocked" state post-boot, allowing an attacker who manages to boot an alternative OS environment to request unsealing. To mitigate this attack surface, FDE policies must incorporate OS-level measurements or explicit locking mechanisms. Windows BitLocker demonstrates this with its use of `PCR#11`. Windows Bootmgr extends `PCR#11` early in the boot process to explicitly lock out the TPM. Even if BitLocker is disabled, this ensures that a fully booted, attacker-controlled Windows environment cannot prompt the TPM to unseal a key using a pre-boot state. Further, sealing against OS-level measurements like `PCR#9` can also provide protection against other attacks, such as malicious *initramfs* modifications (cf. Section 3.2.2).[4]

However, binding to OS-level PCRs (`PCR#8` or `PCR#9`) can introduce severe system fragility. Because these registers measure exact hashes of the bootloader, kernel, and *initramfs*, routine system updates will likely change the hashes and cause the PCR state to diverge. If the FDE policy is not carefully updated *before* the reboot, the system will fail to unseal the drive, locking the user out.[5] FDE implementations must therefore carefully select PCRs that provide OS-level security guarantees without breaking the system during standard patch cycles.

4.3 Certification Scope and Residual Risk

Legislators and certification schemes such as Common Criteria [11] have identified the importance of taking physical tampering into account. For example, the Cyber Resilience Act of the European Union requires secure elements [14] to be *designed to provide protection of at least AVA_VAN.4, as set out in the Common Criteria or the Common Evaluation Methodology* and acknowledges the important role of a TPM serving as a platform secure

[4] https://blog.securityinnovation.com/preventing-initramfs-attacks-tpm
[5] https://0pointer.net/blog/brave-new-trusted-boot-world.html

element, i.e. the implementing act states that *this category includes but is not limited to Trusted Platform Modules (TPMs)*. It also acknowledges different implementation forms, i.e. that they *can be discrete silicon or can be integrated into systems on chip (SoC)*. All the countermeasures discussed so far however operate in an environment where independent, formal security evaluation against physical tampering covers only isolated components.

Leaving other platform components than the TPM out of scope creates a critical architectural asymmetry: while the TPM must be rigorously certified against invasive laboratory tampering, the remainder of the platform, including the physical communication bus and the host firmware, is not. Consequently, the active and passive attacks demonstrated in Section 5 remain feasible on commercial systems not because they defeat the certified security chip, but because they exploit the unprotected boundaries around it.

4.3.1 Attack Feasibility on Commercial Platforms

The experimental findings presented in Section 5 show that both passive key extraction and active reset attacks are practically feasible against commodity hardware using low-cost analyzers and microcontrollers [26, 27].

Even where advanced firmware integrity mechanisms like Intel Boot Guard or AMD Hardware Validated Boot are enabled, they often do not prevent physical manipulation of the boot pipeline. In practice, platforms frequently ship with Intel Boot Guard in non-enforcing configurations, leaving SPI flash rewriting viable via external clips. Furthermore, the aforementioned implementation flaws, key leaks [9], parser vulnerabilities like *LogoFAIL* [7], and untrusted platform keys [8] allow execution to be hijacked before cryptographic measurement invariants can be enforced. When an attacker possesses physical access to the device, the physical firmware storage and the bus connecting the CPU to peripheral security chips represent accessible attack vectors that verified boot alone cannot protect.

4.3.2 Trust Asymmetry at the Protection Boundary

The residual gaps across the discussed countermeasures highlight a fundamental architectural limitation: establishing link-layer encryption between a physically hardened security component and an untrusted host processor is inherently constrained by the weaker endpoint.

In a discrete TPM architecture, the security model suffers from a circular trust dependency. Measured boot and TPM-backed full disk encryption exist precisely because the host CPU, early firmware, and operating system cannot be trusted implicitly; their integrity must be externally measured and attested. However, if link-layer encryption (such as parameter encryption or SPDM) is deployed between the host CPU and the discrete TPM, the untrusted host processor becomes the terminating cryptographic peer of the session:

- **Host-Controlled Negotiation:** In TPM parameter encryption [33], session encryption is initiated by the host caller via `StartAuthSession`. An active attacker can simply opt out of session encryption and issue plaintext unseal requests.

- **Unhardened Session Endpoints:** Authenticated bus encryption must terminate directly on the host CPU, which is an untrusted entity prior to attestation and lacks the dedicated physical tamper resistance of a secure element. During early pre-OS boot, session keys and state stored in host memory or processor registers remain vulnerable to fault injection or firmware execution hijacking before integrity guarantees are established.

- **Relocation of the Attack Surface:** Encrypting the bus does not eliminate physical vulnerability; it merely relocates the point of compromise from the exposed PCB traces to the host silicon and its early firmware stack.

Consequently, the security of the encrypted channel remains bounded by the physical and architectural resilience of the unhardened host endpoint rather than the certified guarantees of the TPM.

4.3.3 Certification Scope versus Attack Surface

This trust asymmetry is directly reflected in the certification landscape. Discrete TPMs achieve Common Criteria certification at high assurance levels (e.g., EAL4+ or EAL5+ with AVA_VAN.5) [11, 15], guaranteeing resilience against invasive silicon microprobing, decapsulation, and laser fault injection. In contrast, only a few commercial platforms take components that store and enforce firmware security into account. Technically, one architectural approach is to use a dedicated platform security controller; one example for such an approach is the HP Endpoint Security Controller (EpSC) evaluated under the French ANSSI CSPN scheme (ANSSI-CSPN-2023/08 and ANSSI-CSPN-2025/01) [1, 2], where countermeasures such as firmware signature verification, CPU reset gating, and private golden-image flash recovery are within scope of certification.

A different approach is to directly prevent offline firmware manipulation by replacing commodity SPI NOR flash with security-certified secure flash ICs. Examples for such secure flash ICs include the Winbond TrustME W75F (certified at Common Criteria EAL5+ with AVA_VAN.5) [40] or W77Q (Common Criteria EAL2+, SESIP Level 2). These components integrate on-die cryptographic engines, physical tamper shields, and authenticated SPI command sets to prevent unauthorized readouts and arbitrary firmware overwrites at the silicon level. Such security-hardened flash ICs however have not been deployed in high-volume commercial PCs, but are usually only found in specialized embedded and IoT systems.

Bus Encryption and Standardization: TCG SPDM Another critical frontier is establishing authenticated, encrypted communication across the discrete TPM bus.

At the standards level, the TCG Server Work Group published *TPM Communication over SPDM Secure Session Version 1.0* in December 2025 [37], establishing DMTF SPDM [13] as the transport wrapper for TPM commands. However, this specification defines requirements only for the TPM, leaving host requirements open; mutual authentication remains caller-selected (`MUT_AUTH_CAP`), and identity keys are exchanged in-band during the session handshake rather than pre-provisioned to cryptographically bind the TPM to a specific platform. Furthermore, corresponding standardization for PC client laptops and desktops remains an ongoing internal draft within the TCG PC Client Work Group.

As of now, the only deployed commercial platform implementing such an approach is TPM Guard [20]. Operating within a platform architecture where an end-point security controller (EpSC) anchors firmware integrity, TPM Guard utilizes DMTF SPDM transport over the SPI bus. During manufacturing, a pre-shared hardware key binds the discrete TPM cryptographically to the specific host CPU, enforcing authenticated session encryption across the bus and mitigating passive sniffing, active reset, and interposition attacks. Crucially, the EpSC provides the prerequisite platform trust anchor: by validating firmware signatures and controlling host CPU reset lines before execution begins, it prevents an attacker from tampering with early boot firmware to hijack the host-side SPDM session. Conceptually, such an approach therefore prevents the presented attacks without requiring a dedicated PIN for FDE.

A detailed security evaluation of such approaches is outside the scope of this work and remains an avenue for future research. Nevertheless, its commercial deployment demonstrates that a secure implementation of the TCG's SPDM specification is feasible on modern PC client architectures.

5 Experimental Results

We carried out the attacks on real hardware to assess their feasibility. As targets, we used a mini PC with a pluggable TPM, where bus communication was easily accessible, and a business-class laptop that proved to be more challenging. We performed passive bus sniffing against Windows 10 with BitLocker and Linux with Clevis to extract the disk encryption keys. For active state forgery attacks, we demonstrated reset attacks against BitLocker and Linux with enabled parameter encryption using `systemd-cryptenroll`. Additionally, we implemented a firmware manipulation attack using a modified TianoCore EDK2 firmware in a QEMU environment.

5.1 Passive Attacks

In both devices, the TPM communicates with the CPU over a Serial Peripheral Interface (SPI). SPI is considered to be a rather "low-speed" bus, which is true compared e.g. to USB 3.2 which offers data rates of up to 20 GBit/s. Nevertheless SPI can still easily reach data rates of several Mbit/s. The TCG's "PC Client Platform TPM Profile Specification" defines a default clock frequency of 24 MHz, while noting that future implementations may operate at higher speeds [36].

We confirmed that those higher speeds are already in use when we tested a TPM on a Raspberry Pi extension board. We measured a clock rate of 28.4 MHz. The data sheet for the chip even stated a maximum bus speed of 43 MHz.

At bus speeds of around 30 MHz simple bus sniffing attempts, that use e.g. jumper wires to connect to the bus, often do not succeed due to the parasitic capacitances of the cables which smooth the signals and make them unrecognizable. This holds not only for the sniffed signals read by the attacker, but also for those received by the rightful communication partners (TPM and CPU). So the sniffing attempt can also lead to a breakdown of the normal communication. Moreover, such speeds push a lot of sniffing devices based on simple microcontrollers to their limits, because their General Purpose Input/Output (GPIO) pins simply cannot follow such fast signal changes.

Yet there are still a lot of cheap logic analyzers on the market which can cope with bus speeds of up to 100 MHz[6]. The price range starts at about 40 EUR for custom PCBs using a Raspberry Pi Pico 2, which has a maximum sample rate of 400 MHz [3],[30],[42], over projects based on cheap FPGA boards like the Lattice ice40 stick [4], up to about a few hundred Euro for commercial devices providing 400 or 500 MS/s[7]. Some of these even come with custom software that has an integrated decoder for the TPM protocol[8].

Therefore, some of these devices might not be able to monitor the necessary number of channels for SPI with the needed sample rate for a sufficient time range.

In our experiments we used a Saleae "Logic Pro 16" logic analyzer which costs about 1.900,- EUR and has a maximum sample rate of 500 MS/s on 16 channels. As software, we used the open source application "*pulseview*". Here we additionally installed a dedicated TPM protocol decoder [22].

Then we took two different approaches to connect the logic analyzer to our targets.

[6] Note that the sample rate should be about four times the bus speed in order to get reliable results.

[7] For example a Kingst LA5016 that offers 500 MS/s for 230,- EUR or the DreamSourceLab DSLogic Plus which provides 400 MS/s for about 130,- EUR

[8] [30], [42] and [4] for instance are even designed as special TPM sniffing devices.

5.1.1 Passive Bus Sniffing Attack on a mini PC

The pluggable TPM on the mini PC mainboard was easily accessible. So we were able to use a "Sensepeek PCBite" set that allows measurements with small needle probes. We used the standard SQ10 probe tips.

We placed the needles on the four SPI bus lines (CS, CLK, MOSI and MISO). For this we had two possibilities:

1. Placing the needles directly at the pins of the TPM. This was a little uncomfortable, because the SPI pins are located on the same side of the chip package (Pins 19, 20, 21 and 24) and therefore almost all directly adjacent. Nevertheless, this was still absolutely feasible.

2. Placing the needles at the serial termination resistors of the SPI bus. These resistors are placed a little further apart from each other, so here the needle placement was a little easier (cf. Figure 12 and Figure 13).

After connecting the probes to the resistors, we recorded the SPI communication while the PC booted into Windows. We chose a sample rate of 50 MHz, because we expected a bus speed of about 24 MHz (see above) and this was the next higher available setting. As mentioned above, the sampling rate usually should be three to four times the bus speed, but we wanted to test which is the lowest possible rate for a successful read-out.

We then applied the TPM protocol decoder to the measurement data. The decoder directly prompted the Volume Master Key (VMK) in the communication. So the rather low sampling rate we applied was already sufficient. The reason was that the clock speed of the TPM was only about 14.39 MHz. The extracted key can then be applied to unlock the encrypted drive as described e.g. in [22] or [29]. The whole process took about 20 minutes.

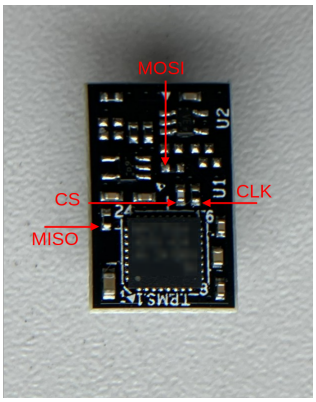


Figure 12: Serial termination resistors for the respective SPI bus lines.

5.1.2 Passive Bus Sniffing Attack on a Laptop

Other than the connection of the TPM on the mini PC mainboard, tapping the TPM on the laptop mainboard required some preparation steps.

We removed the back cover of the laptop. In order to locate the TPM chip we detached all field replaceable units and stripped off the black protective covers. Because we could not find the TPM on this side of the PCB, which we considered the "front side", we removed the mainboard completely and continued our search on the back side, where we finally discovered the TPM (see Figure 14). So tapping the bus using probing needles as described above was not possible because the laptop would then not have been functional.

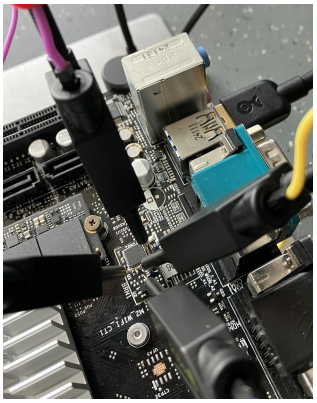


Figure 13: Probing needles placed at the serial termination resistors.

Therefore we contacted the SPI bus lines by soldering enameled copper wire to pads that were available in the respective traces (see Figure 15) and connected these wires to the logic analyzer (Figure 16).

With this setup we could read out the VMK of the laptop following the same procedure as described for the mini PC. We again used a sample rate of 50 MHz which was again sufficient because the bus speed in this case was only about 17.15 MHz. This also explains why we were able to measure clear signals even though the copper wires were several centimeters long. Due to the preparation steps this method took about two hours.

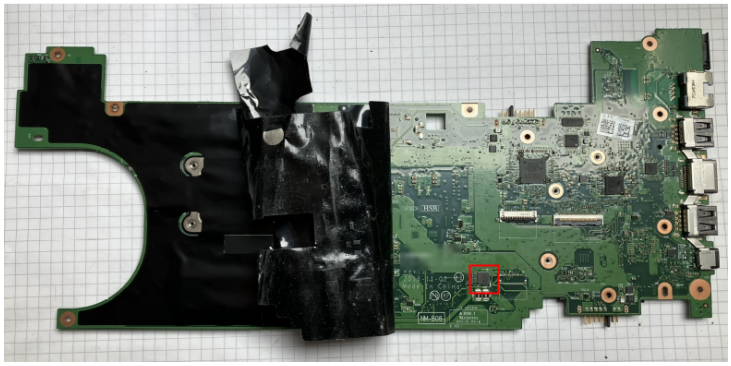


Figure 14: Disassembled mainboard PCB from the back side. The position of the TPM is indicated by the red rectangle.

Similar attacks like this have already been documented in various blogs, see e.g. [12], [16] and [29] or, for a (partial) overview of several attempts, [25]. In some of these approaches the authors made use of the fact that the BIOS flash chip and the TPM share the same SPI bus lines (cf., e.g., [17]). So it is possible to contact the pins of the flash chip instead of the TPM which is a little more convenient, because flash chips usually come in a SOIC-8 package unlike TPM chips which are lately mostly cased in QFN32 packages as was the case for both our targets. In that case soldering becomes a little easier or can be completely omitted when a dedicated SOIC-8 test clip is used (see, e.g., [42]).

This attempt would have also been possible with the laptop we investigated, because here the BIOS flash chip was rather easily accessible. It was located near the SSD on the front side of the PCB, so an attacker would only have to remove the back cover of the laptop and could connect to the SPI bus with a SOIC-8 clip which reduces the preparation steps to a minimum. However we decided to contact the TPM bus lines directly to showcase

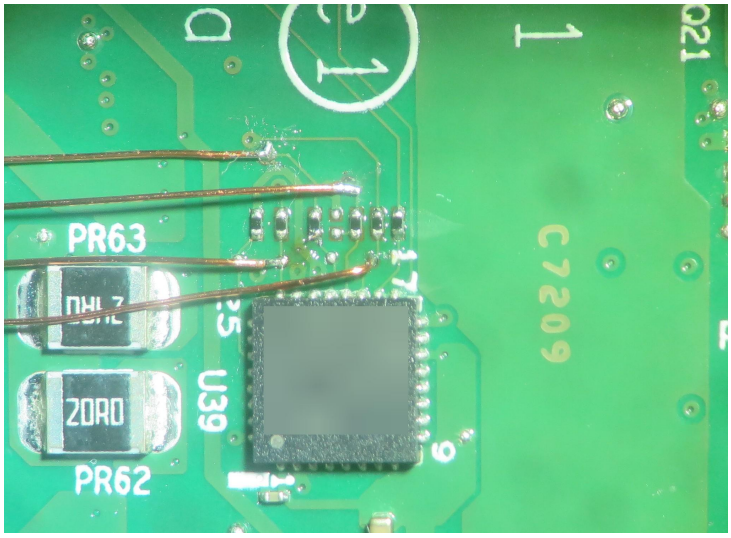


Figure 15: Copper wires connected to pads of the SPI bus on the mainboard of the laptop.

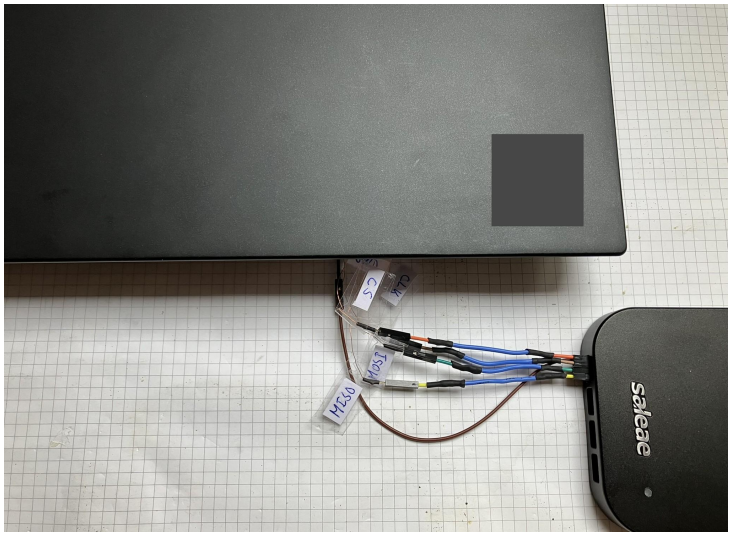


Figure 16: Laptop with the copper wires connected to the logic analyzer.

that the attack is also feasible when the PCB would feature a dedicated SPI connection for the TPM.

To validate our results we used the extracted key to mount the BitLocker partition under a Ubuntu 24.04.2 LTS USB live stick. We could access the file system and all stored files which proved that our recordings contained the correct data.

5.2 Active Attacks

An active attack that resets the TPM after boot does not require access to bus communication. We demonstrated this on the mini PC against Windows BitLocker, a custom `initramfs` script with parameter encryption, and `systemd-cryptenroll`, which enables parameter encryption by default.

5.2.1 Active Attack on a mini PC

Since the mini PC provided a dedicated pinheader for the TPM chip, we decided to utilize this as an access point for the attack. The pinheader was easily accessible, so the most comfortable way to contact the reset pin was to attach a mini grabber clip to the pinheader before the TPM was plugged in.

Figure 17 illustrates the pinout of the header on the mainboard, while Figure 18 shows a mini grabber connected to this pin on the board with the TPM plugged in. The TPM did not fit perfectly when the grabber was attached, but the contact was still sufficient for it to work properly. We connected a jumper wire to the mini grabber clip to establish a connection to a ground (GND) pin on the mainboard in order to carry out the reset of the TPM. After this preparation, we could boot the PC and reset the TPM while the system was running.

If a mini grabber is not at hand, it is also possible to use a simple male-to-female jumper wire to touch the reset pin of the TPM chip directly (pin 3). As a "tool-free" option, the attacker can even unplug the TPM without powering off the PC first. By replugging it, a reset of the TPM is carried out, which clears all PCR values stored before.

Briefly connecting the reset line to GND after boot reset the TPM and cleared all PCRs. We then reconstructed the trusted state by extending the PCRs with the measurements extracted from the event logs and successfully unsealed the BitLocker VMK and the Linux LUKS volume keys, respectively.

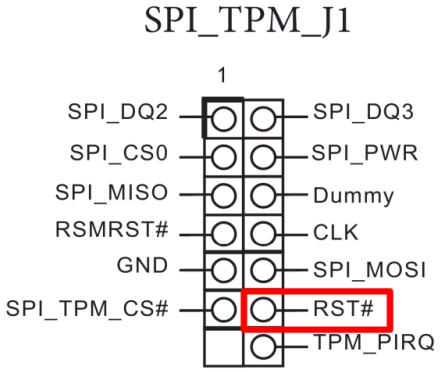


Figure 17: Pinout of the mainboard TPM pinheader on the N100 mainboard.

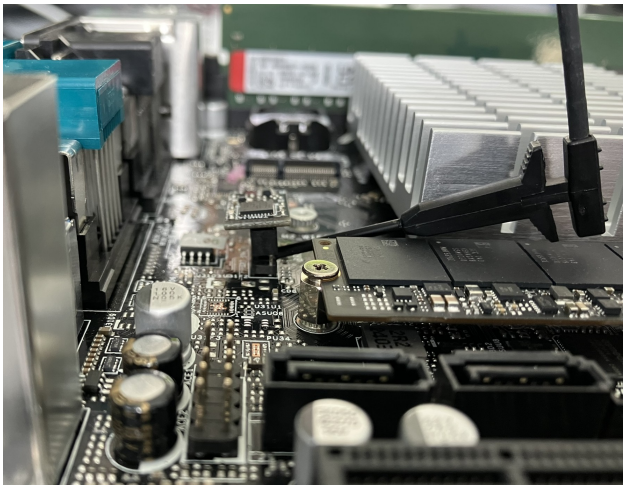


Figure 18: Mini grabber connected to the reset pin.

5.2.2 Active attack on a Laptop

For this attack we again removed the mainboard of the laptop. As visible in Figure 19 all communication pins of the chip are connected to vias that we conjectured to be accessible also from the front side of the PCB. This proved to be true in any case for the reset pin. The respective via leads to a point on the front side that is located under the SSD slot.

So we were able to solder enameled copper wire to the via from the front side of the PCB without having to remove any component but the SSD. For the connection of a wire to GND there were a lot of possibilities, e.g. at the numerous copper pads around the screw holes (cf. Figure 14). This method took, once identified, about 20 minutes to pursue.

However, when we prepared the PCB respectively and tried to reset the TPM, the complete system was freezing. Windows initiated a restart, while Ubuntu just did not react anymore. In both cases we especially noticed that the keyboard was dysfunctional. A check of the mainboard schematic, which could be found on the internet, showed, that the via did not just belong to the reset pin of the TPM but rather to a reset net that connects also to the IC which controls among other things the keyboard of the laptop, which explained the behaviour described above[9]. Therefore it was impossible to specifically reset the TPM with this setup.

Nevertheless, the rather long reset line (see again Figure 19) of the TPM provided enough space to allow for manual manipulation. We removed the solder mask of the respective trace in two places and connected enameled copper wire to these newly created pads (Figure 20). Afterwards we cut the trace at a point between the new connections. We connected the two wires, as well as the GND connection we already established above, to a switch which we placed outside of the laptop casing (see Figure 21). With this switch we could deliberately connect the reset line of the TPM either to the reset net of the mainboard (normal operation state) or to GND (TPM reset state). Using this switch we could reset the TPM to a state where all PCRs only contained 0x00 or 0xFF respectively as shown in the screenshots in Figure 22 and Figure 23.

Due to the fact that for this method the mainboard has to be completely removed and built back in and that the soldering requires slightly more effort, this preparation took about two hours.

[9]This observation is also in accordance with the results of the authors of [18]

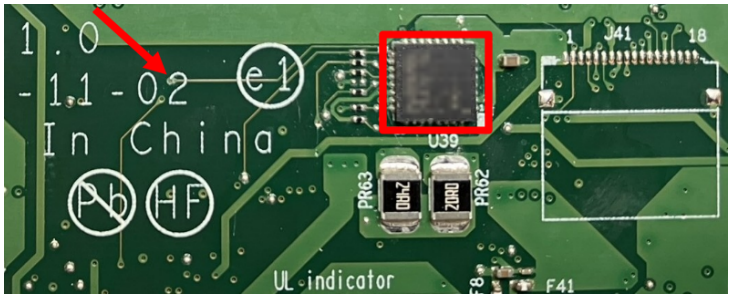


Figure 19: Zoom view of the TPM (red rectangle). Like all other communication pins the reset pin is connected to a via (red arrow). The respective trace is rather long and allows for manual manipulation.



Figure 20: Enameled copper wire soldered to the two new connections.

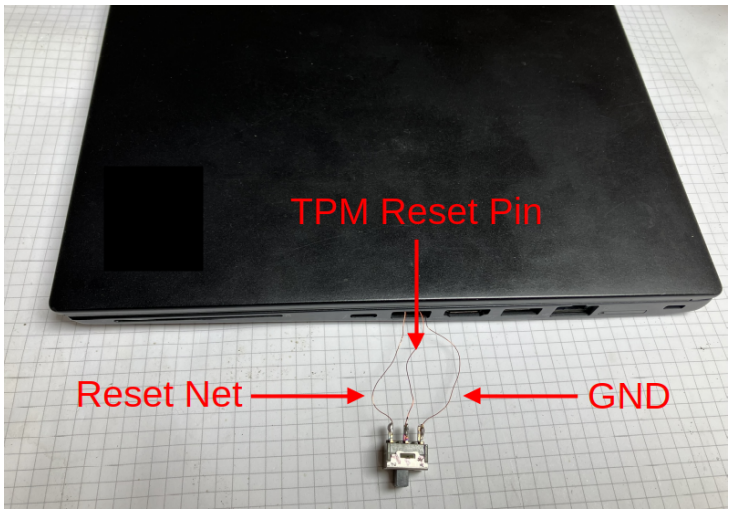


Figure 21: Laptop with external switch to reset the TPM.

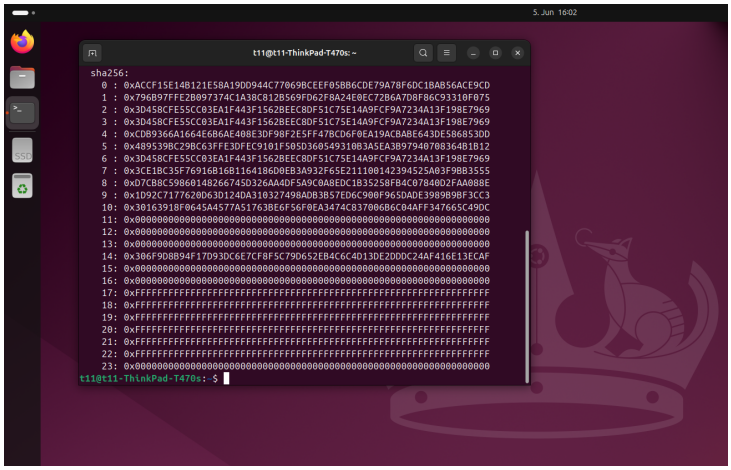


Figure 22: PCR values of the laptop before the reset of the TPM.

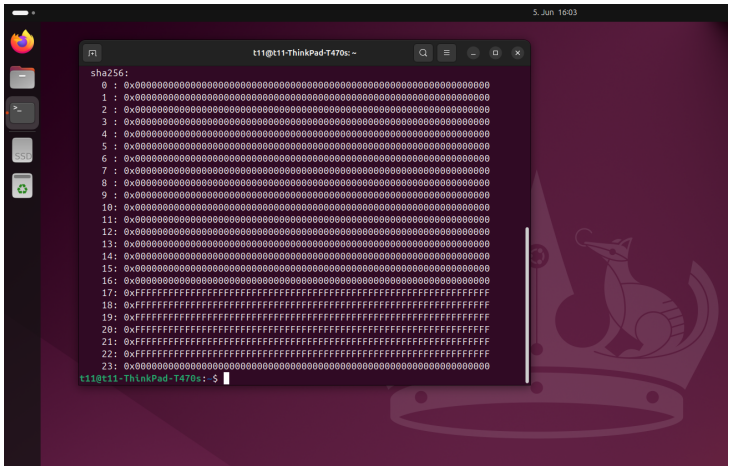


Figure 23: PCR values of the laptop after the reset of the TPM.

5.2.3 Active attack on TianoCore EDK2

Conceptually, reset attacks allow the UEFI to perform valid measurements during boot before the TPM state is cleared. Alternatively, measurements can be prevented entirely by modifying the UEFI code responsible for generating them. This shifts the attack vector from TPM bus communication to firmware integrity and therefore affects both dTPMs and fTPMs. It should be emphasized that deploying modified firmware on modern physical hardware is not trivial when hardware roots of trust (such as Intel Boot Guard or AMD Hardware Validated Boot) are strictly enforced. However, in practice, firmware integrity can be subverted through multiple realistic avenues: misconfigured or unprovisioned hardware (common on desktop and workstation motherboards where Boot Guard is left in non-enforcing modes), early boot software vulnerabilities (such as *LogoFAIL* [7], which hijacks execution without altering signed binaries), and supply-chain key compromises (exemplified by the 2023 leak of MSI Intel Boot Guard private signing keys [9] and the deployment of default vendor test keys like *PKfail* [8]). To demonstrate the technical impact on TPM measurements, we set up QEMU with a virtual TPM and modified

TianoCore EDK2[10] to suppress PCR measurements. Booting with the modified firmware resulted in an empty PCR bank that enabled state forgery attacks.

The UEFI boot process is split into multiple stages, notably the Pre-EFI Initialization (PEI) and Driver Execution Environment (DXE) phases. During the PEI phase, system hardware such as memory is initialized; furthermore, `PCR#0` is populated with the first measurements, including firmware strings and firmware blobs. These measurements can be prevented by patching out the `HashLogExtendEvent` function from the TPM 2.0 PEI driver (`SecurityPkg/Tcg/Tcg2Pei/Tcg2Pei.c`):

```
HashLogExtendEvent (
    IN      EDKII_TCG_PPI      *This,
    IN      UINT64             Flags,
    IN      UINT8              *HashData,
    IN      UINTN              HashDataLen,
    IN      TCG_PCR_EVENT_HDR  *NewEventHdr,
    IN      UINT8              *NewEventData
    )
{
    return EFI_SUCCESS;
}
```

During the following DXE phase, two mechanisms measure into the PCR allocation. For one, whenever an executable is loaded from an external source such as a disk, that executable is measured into `PCR#2` or `PCR#4`. Option ROMs and third-party drivers are typically measured into `PCR#2`, while bootloaders such as GRUB and Windows Bootmgr are measured into `PCR#4`. The second mechanism is the DXE TPM 2.0 driver measuring system configuration and boot state at specific points during the boot process. Due to the different triggers for these measurements, the TPM 2.0 DXE driver (`SecurityPkg/Tcg/Tcg2Dxe/Tcg2Dxe.c`) must have two functions disabled: `TcgDxeHashLogExtendEvent` and `Tcg2HashLogExtendEvent`:

```
TcgDxeHashLogExtendEvent (
    IN      UINT64             Flags,
    IN      UINT8              *HashData,
    IN      UINT64             HashDataLen,
    IN OUT  TCG_PCR_EVENT_HDR  *NewEventHdr,
    IN      UINT8              *NewEventData
    )
{

    return EFI_SUCCESS;
}

...

Tcg2HashLogExtendEvent (
    IN EFI_TCG2_PROTOCOL     *This,
    IN UINT64                Flags,
    IN EFI_PHYSICAL_ADDRESS  DataToHash,
    IN UINT64                DataToHashLen,
    IN EFI_TCG2_EVENT        *Event
    )
{
    return EFI_SUCCESS;
}
```

In order to achieve a full zero-state, preventing measurements during the UEFI boot process alone is not sufficient, as `PCRs#0` through `7` only cover firmware-related measurements; some operating systems extend additional PCRs during their own boot stages. Modern versions of Windows notably use `PCR#11` to lock BitLocker decryption, even when BitLocker is disabled. Meanwhile, the Linux Integrity Measurement Architecture (IMA) kernel subsystem measures hashes of files when opened; this will taint `PCR#10` during OS

[10] TianoCore EDK2 (EFI Development Kit II) is the official open-source reference implementation and cross-platform toolkit for building Unified Extensible Firmware Interface (UEFI) and Platform Initialization (PI) firmware, cf. https://github.com/tianocore/edk2

initialization. A zero-state can most easily be reached by booting a Linux distribution that does not measure into any PCR during boot, such as Arch Linux or Alpine Linux.

Booting the system using the patched EDK2, here using Arch Linux inside QEMU, results in a root environment with full TPM access and zeroed PCRs, as shown in Figure 24. From this unmeasured zero state, the attacker can execute the replay stage of the state forgery attack (cf. Section 3.2.1). By calling `tpm2_pcrextend` with the target system's measurement sequence extracted from event logs, the attacker extends the zeroed PCR bank to match the exact target policy. Once the trusted state is forged, `tpm2_unseal` successfully releases the sealed volume key, allowing the attacker to decrypt and mount the target BitLocker or LUKS encrypted partition. Having demonstrated the feasibility of both physical and firmware-level active attacks, we systematically analyze architectural countermeasures and sealing policies in Section 4.

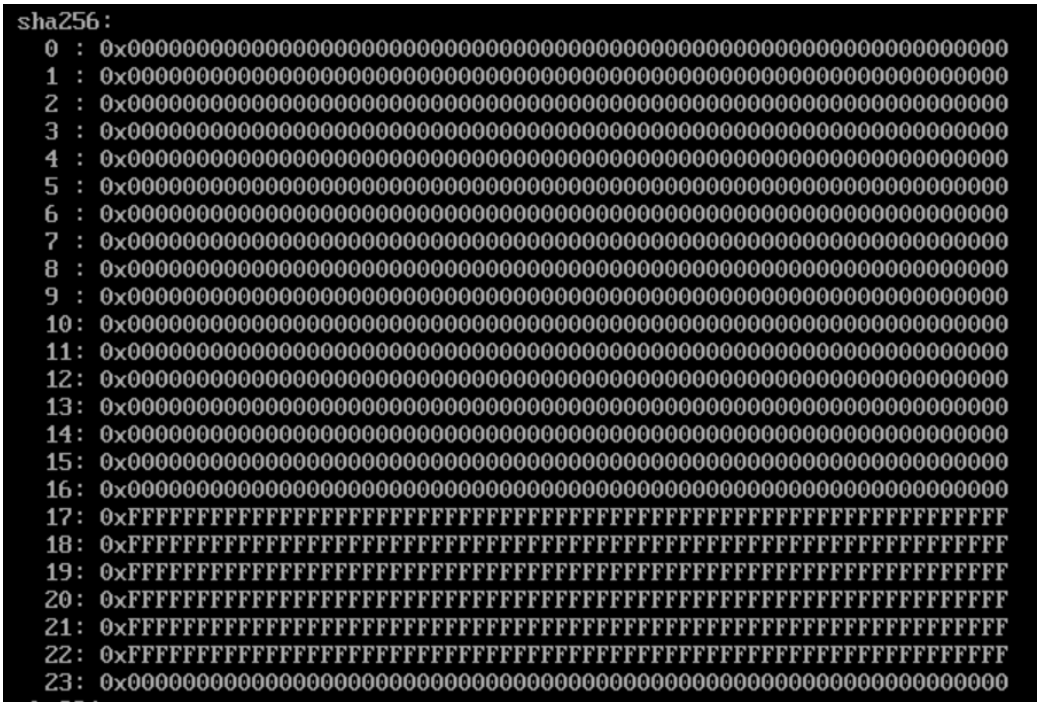


Figure 24: Full Zero State after booting a patched EDK2 firmware.

6 Related Work

Active and passive bus attacks have been discussed for more than 15 years, with one of the earliest references being [38]. These works focus on the Windows BitLocker implementation and discuss several attack scenarios, specifically highlighting the lack of a trusted path to the user as well as reset attacks. However, their work is limited to theoretical considerations, and none of the described attacks were implemented in practice. Winter et al. [41] later refined these abstract considerations, focusing more specifically on the LPC and I2C communication buses between the host system and the TPM. They likewise concentrated on theoretical aspects and did not experimentally implement the attacks.

The TCG reacted to such attacks by publishing guidance documents addressing active [35] and passive attacks [33]. Unfortunately, BitLocker does not follow these recommendations regarding passive attacks, and most OEMs do not harden their systems against active attacks.

With the widespread adoption of Windows BitLocker, attacks based on sniffing the SPI bus between the CPU and the TPM have become increasingly common and have been reported in numerous publications over several years. These reports often appeared as blog posts or instructions hosted in Git repositories, many of them published under aliases. One approach simplified SPI bus sniffing by implementing the decoding logic on a Raspberry Pi Pico [30], thereby reducing both the cost and the attack window from the attacker's perspective.

Active attacks, including reset attacks, have also been demonstrated in practice. Security researchers [18] reset a dTPM on a Dell mainboard by shorting `LRESET#` via the LPC bus. On other systems, however, this triggered a processor reset. Very recently, independent of this work, Werling et al. [39] demonstrated active reset attacks on an Asrock B850 desktop mainboard and a Lenovo ThinkPad T480 equipped with dTPMs. They successfully reset the TPM through the `PERST#` pin on the M.2 connector to compromise Windows BitLocker encryption, arguing that because dTPMs expose physical communication and reset signals, measured boot designs should favor fTPMs over dTPMs.

Our independently conducted experiments also target both a desktop mainboard and a notebook computer. However, for our notebook target, neither `PERST#` nor `LRESET#` allowed isolated TPM resets, requiring physical manipulation of the TPM's reset line. Furthermore, we consider the current state of Linux implementations using parameter encryption. Moreover, we prototypically implement an active attack that modifies UEFI boot code, thereby demonstrating that practical active attacks are not limited to bus communication but also affect the integrity of UEFI and low-level firmware, which equally impacts both dTPMs and fTPMs.

A variant of such an active attack was discovered in [6]. The authors were able to activate a debug interface on certain Intel processor chips without triggering modifications of the TPM PCR values. This debug interface was then used to extract the BitLocker VMK from RAM. Additionally, Han et al. used ACPI sleep states to reset PCRs [19].

dTPMs are typically hardened against physical tampering attacks and often achieve high-level Common Criteria certifications. Firmware TPMs, by contrast, are often not hardened against tampering; one possible reason is the difficulty of integrating a hardened component into a general-purpose CPU design without introducing side channels. Several severe vulnerabilities in fTPMs have been discovered over the years, including a recent result involving voltage fault attacks [21]. It should be noted, however, that advanced side-channel attacks can affect both fTPMs and dTPMs alike [24].

7 Conclusion

Our work and experiments highlight that passive attacks are not the only source of concern. The most straightforward countermeasure is to implement parameter encryption, a mechanism that has been available since the introduction of TPM 2.0. Unfortunately, parameter encryption is not implemented in Windows BitLocker and some Linux frameworks.

Our results also show that active attacks are not significantly more difficult to execute in practice when the bus is easily accessible. In such cases, parameter encryption alone does not prevent the attack.

One might assume that not exposing the bus by using a firmware TPM (fTPM) could mitigate active attacks. However, this does not fully eliminate the threat. First, our experiments demonstrate that fTPMs are also susceptible to active attacks; the target of the attack then changes from the bus to the host system firmware. Second, our analysis does not account for physical tampering attacks on the TPM itself. Typically, fTPMs are not hardened against tampering—for example, they often lack Common Criteria certification—and several severe vulnerabilities have been discovered in the past[11].

[11] As a few examples we mention CVE-2023-1017, CVE-2023-1018, CVE-2020-12954, CVE-2017-16837;

An ideal security setup would combine a hardened TPM (e.g., Common Criteria certified), a fully encrypted bus that extends beyond TPM 2.0's parameter encryption, and proper protection of the secret used to set up bus encryption, for instance on a tamper-protected UEFI or firmware controller. UEFI integrity protection is also required for components responsible for TPM measurements during boot.

Incorporating a PIN into the sealing process provides protection against all the attack vectors discussed. Entering two separate PINs during boot—one to unlock the disk encryption key and one for OS login—is, however, inconvenient from a user perspective. For typical single-user systems, future OS designs could tie both the login credentials and the user password to a single PIN, allowing both secrets to be unsealed in one step. This approach would simultaneously improve user experience and strengthen security.

8 Responsible Disclosure

Aside from the lack of parameter encryption in BitLocker, active and passive attacks on TPM 2.0-based FDE do not stem from implementation errors, but rather from a threat model that does not take physical tampering fully into account. We reached out to the Trusted Computing Group (TCG) with our findings, which welcomed our contribution and highlighted that standardization work is ongoing to secure bus communication [37].

References

[1] Agence nationale de la sécurité des systèmes d'information (ANSSI): Certification Report ANSSI-CSPN-2023/08: HP Sure Start Hardware Root of Trust (2023)

[2] Agence nationale de la sécurité des systèmes d'information (ANSSI): Certification Report ANSSI-CSPN-2025/01: HP Endpoint Security Controller (EpSC) Platform Hardware Root of Trust (2025)

[3] Agustín Gimenez Bernad: github.com/gusmanb/logicanalyzer (2022-2026)

[4] Alexander Couzens: https://github.com/lynxis/lpc_sniffer/ (2017)

[5] Aon Cyber Labs: dislocker: decrypt BitLocker volumes on Linux (2024), https://github.com/Aorimn/dislocker

[6] de Assumpção, M.B., dos Reis, M.A., Marcondes, M.R., da Silva Eleutério, P.M., Vieira, V.H.: Forensic method for decrypting TPM-protected BitLocker volumes using Intel DCI. Forensic Science International: Digital Investigation **44** (2023)

[7] Binarly Research: The Far-Reaching Consequences of LogoFAIL. https://www.binarly.io/blog/far-reaching-consequences-of-logofail (2023)

[8] Binarly Research: PKfail: Untrusted Platform Keys Undermine Secure Boot. https://www.binarly.io/blog/pkfail-untrusted-platform-keys-undermine-secure-boot (2024)

[9] BleepingComputer: Intel Investigating Leak of Intel Boot Guard Private Keys After MSI Breach. https://www.bleepingcomputer.com/news/security/intel-investigating-leak-of-intel-boot-guard-private-keys-after-msi-breach/ (2023)

[10] Carrier, B.: The Sleuth Kit (TSK) (2024), https://github.com/sleuthkit/sleuthkit

note that this is neither an exhaustive nor a representative list

[11] Common Criteria Recognition Arrangement: Common Criteria for Information Technology Security Evaluation (ISO/IEC 15408) (2022)

[12] Compass Security: https://blog.compass-security.com/2024/02/microsoft-bitlocker-bypasses-are-practical/ (2024)

[13] Distributed Management Task Force (DMTF): Security Protocol and Data Model (SPDM) Specification, Version 1.3. https://www.dmtf.org/standards/spdm (2024)

[14] European Union: Commission Implementing Regulation (EU) 2025/2392 of 28 November 2025 on the technical description of the categories of important and critical products with digital elements pursuant to Regulation (EU) 2024/2847 of the European Parliament and of the Council (2025)

[15] Federal Office for Information Security (BSI): Certification Report BSI-DSZ-CC-1113-V3-2022: Infineon Technologies AG Security Target OPTIGA TPM SLB 9672 v15.xx / v16.xx. https://www.bsi.bund.de (2022)

[16] Giggi0x00: https://github.com/giggi0x00/Bitlocker-SPI-TPM-Key-sniffing (2024)

[17] Guillaume Quéré: https://www.errno.fr/BypassingBitlocker.html (2024)

[18] Hacky Solutions: https://hacky.solutions/blog/2024/02/tpm-attack (2024)

[19] Han, S., Shin, W., Park, J.H., Kim, H.: A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. In: Proc. 27th USENIX. pp. 1229–1246 (2018)

[20] HP Threat Research: TPM-Guard: Advancing Endpoint Protection Against Physical Attacks. https://threatresearch.ext.hp.com/tpm-guard-advancing-endpoint-protection-against-physical-attacks/ (2026)

[21] Jacob, H.N., Werling, C., Buhren, R., Seifert, J.P.: faulTPM: Exposing AMD fTPMs' Deepest Secrets. In: Proc. 8th IEEE EuroS&P (2023)

[22] Lucas Teske: https://lucasteske.dev/2024/01/tpm2-bitlocker-keys (2024)

[23] Microsoft Corporation: TPM Tool (2024), https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/tpmtool

[24] Moghimi, D., Sunar, B., Eisenbarth, T., Heninger, N.: TPM-FAIL: TPM meets Timing and Lattice Attacks. In: Proc. 29th USENIX (2020)

[25] NoobieDog: https://github.com/NoobieDog/TPM-Sniffing (2024)

[26] Nurmi, H.: Sniff, there leaks my BitLocker key. https://web.archive.org/web/20201221115038/https://labs.f-secure.com/blog/sniff-there-leaks-my-bitlocker-key/ (2020)

[27] Roth, T.: Breaking BitLocker in 43 seconds with a $10 Raspberry Pi Pico. stacksmashing (2024)

[28] systemd developers: systemd-cryptenroll: Enroll TPM2/FIDO2/PKCS11 tokens to LUKS2 partitions (2024), https://github.com/systemd/systemd

[29] Theophile Wemaere: https://github.com/Theophile-Wemaere/TPM-sniffing/tree/main (2024)

[30] Thomas Roth: https://github.com/stacksmashing/pico-tpmsniffer (2024)

[31] tpm2-software: tpm2-tools: TPM 2.0 tools for Linux (2024), https://github.com/tpm2-software/tpm2-tools

[32] Trusted Computing Group: TCG D-RTM Architecture Specification, Version 1.0. https://trustedcomputinggroup.org/resource/d-rtm-architecture-specification/ (2013)

[33] Trusted Computing Group: CPU to TPM Bus Protection Guidance — Passive Attack Mitigations, Version 1.0 (2015)

[34] Trusted Computing Group: TCG TPM Library Specification, Parts 1-4, Version 2.0 (2015)

[35] Trusted Computing Group: CPU to TPM Bus Protection Guidance — Active Attack Mitigations, Version 1.0 (2023)

[36] Trusted Computing Group: TCG PC Client Platform TPM Profile (PTP) Specification, Version 1.05 (2024)

[37] Trusted Computing Group: TPM Communication over SPDM Secure Session, Version 1.0 (2025)

[38] Türpe, S., Poller, A., Steffan, J., Stotz, J.P., Trukenmüller, J.: Attacking the BitLocker Boot Process. In: Proc. Trust. LNCS, vol. 5471 (2009)

[39] Werling, C., Zahin, T., Seifert, J.P.: Not Discrete Enough: On the Inherent Insecurity of dTPMs for Measured Boot. In: Proc. Cyber Security Experimentation and Test (CSET) (2025)

[40] Winbond Electronics Corp.: Security Target Lite: Winbond W75F Secure Memory, Version 2.4 (2024)

[41] Winter, J., Dietrich, K.: A hijacker's guide to communication interfaces of the trusted platform module. Computers & Mathematics with Applications **65**(5), 748–761 (2013)

[42] Zaphoxx: github.com/zaphoxx/pico-tpmsniffer-spi (2024)