



This work is distributed
under the CC BY-SA license,
unless stated otherwise.

Hardware Security

4. TPMs, Trusted Boot, Disk encryption

Ján Jančár

Summer 2026

Today

Trusted Platform Modules

What are they? Where are they used? How do they fail?

This is going to be dense. Sorry...
It will help you with the first homework assignment

TPM

Trusted Platform Module

*Its design is so flexible that in most contexts the term “TPM” is largely replaceable with “**magic security pixie dust**.” –Eric Chiang*

- Cryptographic processor
- Not programmable, fixed set of functions
- Low-level, generic API, command-response
- Specified by the [Trusted Computing Group](#) (TCG)
- **Functions**
 - Secure storage
 - Cryptographic operations
 - Root of trust
 - Remote attestation
- Very complex: thousands of pages of specs

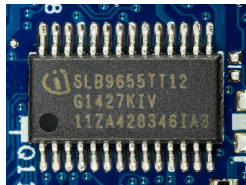


- TPM 1.2, superseded
 - SHA-1, RSA
- **TPM 2.0** (focus today)
 - Algorithmic flexibility
 - ⊕ SHA-256, ECC
 - More hierarchies
 - More root keys

TPM

Types

- **dTPM:** Discrete TPM
 - Separate chip on the motherboard
 - LPC, SPI or I2C connection
- **iTPM:** Integrated TPM
 - Separate area of the CPU
- **fTPM:** Firmware TPM
 - Firmware running on CPU ([paper](#))
- Software TPM
 - Development and debug
 - <https://github.com/microsoft/ms-tpm-20-ref>
 - <https://sourceforge.net/projects/ibmswtpm2/>



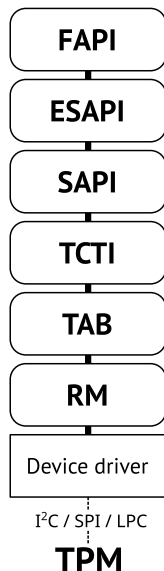
https://en.wikipedia.org/wiki/Trusted_Platform_Module#/media/File:Lenovo_N20_Chrome_-_motherboard_-_Infineon_SLB9655TT12-49890.jpg

TPM

TPM software stack (TSS)

High level -> low level

- FAPI: Feature API
- ESAPI: Enhanced System API
- SAPI: System API
- TCTI: TPM Command Transmission Interface
- TAB: TPM Access Broker
- RM: Resource Manager
- Device driver
- **Implementations**
 - `tpm2-tss`
 - `TSS.MSR`
 - `TrouSerS`



*A TPM 2.0 **entity** is an item in the TPM that can be referenced with a handle.*

*A **handle** is a 32-bit reference to an entity in the TPM.*

*An **entity name** is the hash of its public area.*

- Permanent entities
 - Defined by the TPM spec
 - Can't be created/deleted
 - Persistent hierarchies, ephemeral hierarchy, PCRs, ...
- Nonvolatile indexes
 - NVRAM storage
- **Objects**
 - Either key or data
 - Have a public area and a sensitive area
 - Belong to a **hierarchy**
- Persistent / Non-persistent
 - Whether they persist through a power cycle
 - Limited amount of persistent memory

A stage with red curtains and two speakers. The curtains are a deep red color and hang in vertical folds. The stage floor is dark, and two black speakers are visible at the bottom of the frame. The text "Cryptographic intermezzo" is centered on the curtains in white.

Cryptographic intermezzo

Cryptographic intermezzo

- **HMAC:** Hash-based Message Authentication Code
 - Takes a key and a message
 - Produces authentication tag
 - Security
 - ▶ Without the key the attacker can not produce a valid tag
 - ▶ Thus, the tag is unpredictable to the attacker
- **KDF:** Key Derivation Function
 - Takes a secret (a seed), some additional data
 - Creates one or more keys
 - Example
 - ▶ $\text{KDF}_K(\text{'auth key'})$ and $\text{KDF}_K(\text{'encryption key'})$
 - ▶ You get two keys from one K
 - Security
 - ▶ Indistinguishable from a uniformly random bit-string
 - ▶ Thus, also unpredictable to the attacker
- **Key wrapping**
 - **Encrypt (wrap)** a key under another TPM-only key
 - Wrapped key can be offloaded from the TPM
 - On load, TPM decrypts it
 - **Problem:** wrapped key could be modified
 - **Solution:** also use a MAC

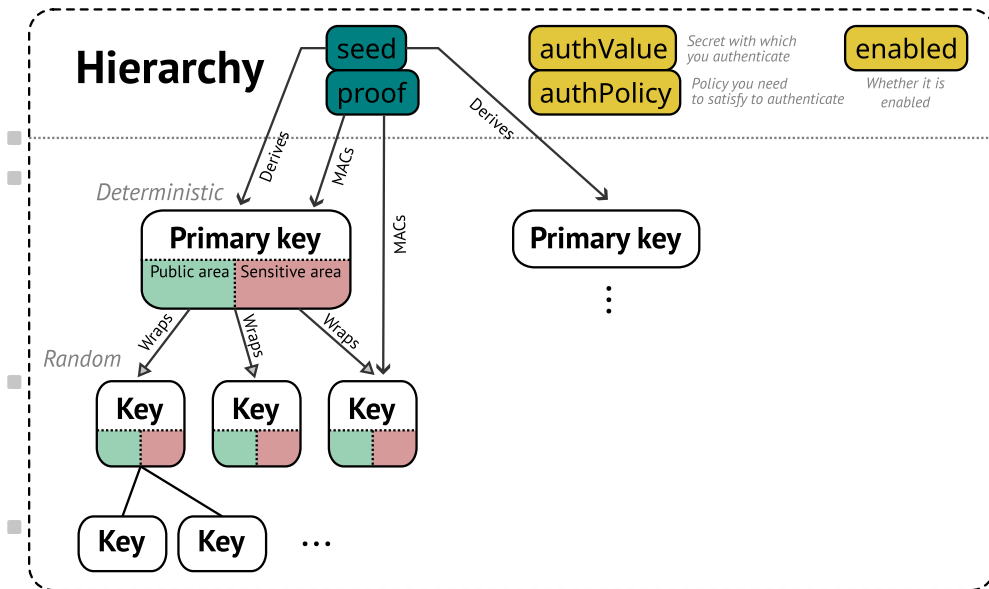
A **hierarchy** is a collection of entities that are related and managed as a group.

- Like a forest (multiple trees)
- Each has
 - Authorization **value** and **policy**: who and when can access it?
 - **Enable** flag: is it enabled?
 - Persistent **seed**: secret from which primary keys are deterministically derived
 - Persistent **proof**: secret which is used to MAC keys
 - **Primary keys**: keys which are the roots of the hierarchy
- Three persistent hierarchies
 - **Platform**: control of the platform manufacturer (BIOS/UEFI)
 - **Storage**: control of the owner/user
 - **Endorsement**: control of the owner/admin
- **NULL**/ephemeral: random seed for every boot



TPM

Hierarchies



TPM

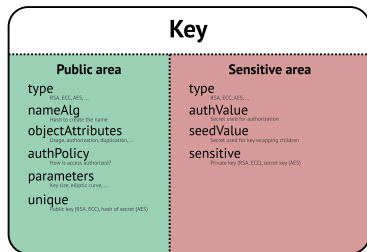
Primary keys

- Roots of the trees
- Derived deterministically from the hierarchy **seed** and key properties (**template**), using the KDF
 - You always get the same key back
 - The template can include some unique data, e.g. "myAppKey"
 - No storage necessary
- **Endorsement Key (EK)**
 - Generated during manufacture
 - Signed by manufacturer's key: EK certificate
 - You can prove that you have a genuine TPM
 - ▶ Sign something with an EK
 - ▶ Usually you will sign another Attestation Identity Key (AIK)
 - **Can not** be exported
- **Storage Root Key (SRK)**
 - Can be regenerated
 - Parent of the keys in the storage hierarchy
 - **Can not** be exported
- Anything else ...

TPM

Keys (non-primary)

- Can be persistent or not
- Have a parent
- Are generated randomly
- Have an `authValue` and `authPolicy`
- Sensitive area never leaves the TPM in the clear
- Can be exported, wrapped under their parent
 - ❓ How is this done if the parent is not a symmetric key? (e.g., an RSA key)
 - ➡ Use the KDF and derive an AES and an HMAC key from the parent's `seedValue`
 - ➡ Then encrypt with the AES key and MAC with the HMAC key
- Access to them can be controlled



Authorizations and sessions

An **authorization** is the process of granting access to a resource based on an authenticated identity.

- Specify `authValue` and `authPolicy` at key creation
- **Password** authorization
 - Send `authValue` in plaintext with the command
 - TPM compares with the expected `authValue`
 - ? Problems?
 - Bus sniffing and MiTM, replay attacks
- **HMAC** session
 - Use `authValue` (if any) and nonces for freshness to derive a session key
 - HMAC all command and response data using that key
 - Optionally, parameter encryption
 - ? How does it help?
 - Prevents active MiTM
 - Prevents replay attacks, nonces provide freshness
 - Prevents some bus sniffing if parameter encryption is enabled
- **Policy** session
 - Can express conditions on the state of the TPM, secrets, passwords, signatures
 - “*authorization Rube Goldberg machines*”

- We protect keys with an `authValue`
- ? What prevents simple brute-forcing?
 - ➔ Native hardware rate-limiting
- TPM tracks failed auth attempts internally
 - Parameters: `maxTries`, `recoveryTime`, `lockoutRecovery`
- **Lockout state**
 - Reach `maxTries` -> DA lockout
 - TPM refuses auth-requiring commands
 - Cleared by timer or lockout hierarchy

Platform Configuration Registers

- 24 registers
- Core operation: $\text{Extend}_i(\text{data})$
 - $\text{PCR}[i] = \text{SHA-256}(\text{PCR}[i] \parallel \text{data})$
 - You can only **add** to the PCRs, and not in a way where you control the outcome
- Can be only reset with reboot
- Used for boot integrity
 - Components in the boot chain measure the next one and extend PCR values with the measurement

```
0 : 0xB0DE51075A45BB075F70F82D51D2B1A497790C2DF8D043B43E765B0290F9D833
1 : 0x91709E01C44A58697E01AA354A0539E8B650B0307FFE9AE364C27F5155791CE2
2 : 0x6F633F15C4273A273367F0FCC3CFBA3B160A4AD735E7A8B941D8A3FDB29BD2FB
3 : 0x3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
4 : 0x1AFDBF65F9EE35C32778CA3C8289E6BF2958FEA4A6BD62D0E497D5DC16A45DD2
5 : 0xC42DF3ED0392D66D72299AFA2B9B26C9FAEBBCF73B398340042CDF19E2A76E22
...
```

What is all of this good for?!?

Right?

- Boot integrity
 - Measured boot, trusted boot, verified boot, secure boot
- Disk encryption
 - Example: [BitLocker](#), [LUKS](#)
- Remote attestation
 - Verify the integrity of a computing environment remotely
- FIDO2/WebAuthn
 - Platform authenticator
 - Example: [Windows Hello](#)
- Hardware-backed keys
 - Never leave the hardware
 - You can attest that they never leave
- Generic crypto processor
 - True random number generator
 - Dictionary-attack protection
- Example: [Chrome OS](#)
 - Preventing SW and FW version rollback
 - Protecting user data encryption keys
 - Attesting keys and device mode

Boot integrity

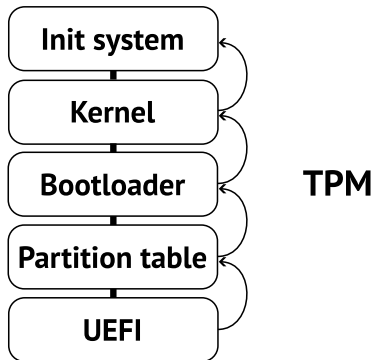
Motivation

- I use full-disk encryption on my laptop using [LUKS](#)
- When it boots, a binary from the non-encrypted /boot partition asks for my password:
Enter passphrase for /dev/nvme1n1p3:
- I enter it, the binary decrypts and mounts my system, including all my data
- What if there was an evil maid attack (described [here](#))?
 - Booted up my laptop using a live-USB
 - Modified the binary on the non-encrypted /boot partition such that it
 - ☑ Asks for my password
 - ☑ Decrypts and mounts my system
 - ❓ Stores the password somewhere hidden on that system
 - ✴ Adds a backdoor that runs on boot and transmits my password, along with any other data out, then deletes itself
- ❓ How do you detect and prevent this kind of attack?

Boot integrity

The setup

- The boot happens in [stages](#)
- We want to detect/prevent any malicious changes to the boot stages
- **Idea:** stage n somehow checks stage $n + 1$, then only stage 0 needs to be trusted



Boot integrity

Root of trust

The base component of a chain of trust that is implicitly trusted.

- For measurement (CRTM)
 - BIOS, UEFI, Intel Boot Guard, AMD PSP
- For storage
 - TPM
- OpenTitan & Google Titan

Root of Trust



Turtles all the way down

Boot integrity

Many words, similar concepts

■ Measured boot

- Each stage measures the next stage and stores in a tamper-proof log
- Tampering with stage n will be detected in measurement from stage $n - 1$
 - ▶ This is why we need a root of trust, what about stage 0?
- Usually, measured boot is about **detection**, the boot process will continue
- Example: TPM PCRs
 - ▶ You have to know what PCR values are **good**
 - ▶ Setup the system in a known good state, measure, then remember

■ Verified boot

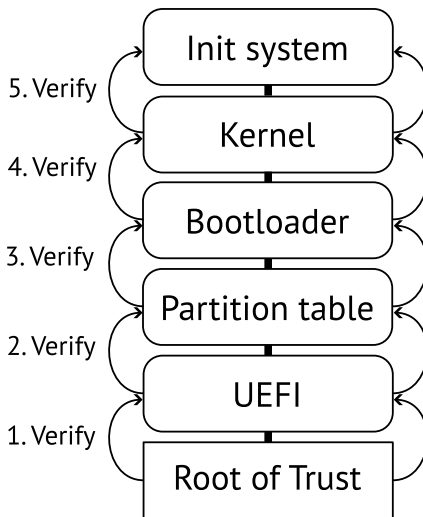
- Each stage verifies a signature of the next stage and runs it only if it is valid
- Tampering with stage n will break signature verified by stage $n - 1$
 - ▶ This is why we need a root of trust, what about stage 0?
- Usually, verified boot is about **prevention**, the boot process will stop
- Example: [Secure Boot](#) (UEFI), [Trusted Boot](#) (Windows)

■ [Explainer by Microsoft](#)

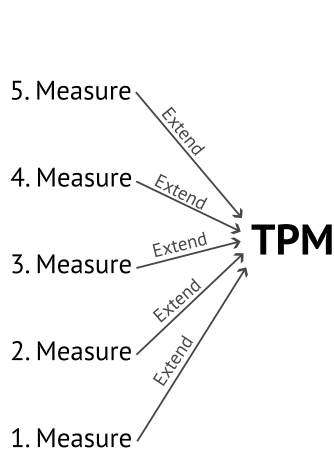
Boot integrity

Many words, similar concepts

Verified boot



Measured boot



TPM

Boot integrity

How would it have helped?

■ Measured boot

- I would see changed PCRs after booting
- But how would I get to them?
 - ▶ System can be compromised, even the PCR read utility
- Needs one more magic element
 - ▶ Storing the key in the TPM and authorizing it by the PCR state

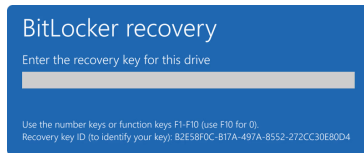
■ Verified boot

- System would fail to boot with modification
- I would be prevented from leaking my password and data

Boot integrity

BitLocker recovery key

- I was setting up a new laptop
- Immediately installed Linux, but kept Windows for occasional use
- Did some firmware updates, changed the partition table
- Booted into Windows, met with this:



- What happened?
 - BitLocker uses the TPM and stores the drive encryption key in it, authorized by the PCR state
 - My changes modified the PCR state
 - Measured boot worked!
- The Windows installation backed up a recovery key to my Microsoft account
 - Microsoft has your BitLocker keys by default
 - You can recover
 - Microsoft can leak/give your key to law enforcement

Disk encryption

- Storage encryption possible at various levels
 - File-system encryption ([Linux fscrypt](#), [Apple fs](#))
 - **Disk (sector) encryption** ([BitLocker](#), [Cryptsetup/LUKS](#), [VeraCrypt](#))
 - HW-based encryption ([Opal2](#), [SED](#))
 - ▶ Beware of [Self-Encrypting Deception](#), many drives are broken
- Disk encryption (FDE)
 - Encrypts sectors of a disk, usually using one per-partition key
 - Modes like [XES](#), [XTS](#) or [Adiantum](#)
- Often, does not provide integrity on its own
 - Rollback to old data is possible
 - Integrity is tricky
 - ▶ needs extra data for tag (not length-preserving)
 - ▶ needs atomicity of data + tag write

Disk encryption

Role of a TPM

- It holds the key and releases (decrypts) based on some policy
 - State of PCRs
 - Correct password
- Properties
 - ✓ Evil maid booting a live USB: PCRs will be different, and the TPM will refuse to release the key
 - ✓ Disk put into different machine: No TPM to load the key from
 - ✗ Disk and TPM put into different machine with a backdoored root of trust: Attacker can replay PCR extension events

Remote attestation

- 🔍 You measured your boot. How do you prove your state to a remote party?
- ➦ **Idea**
 - Take current PCR values (stored inside TPM)
 - Sign it by TPM's attestation private key (AIK), (happens inside TPM)
 - Send signed PCR values to remote party
 - Remote party holds public key and can verify signature => trust in authenticity of PCR values
- Can attest any hardware-backed value this way

Breaking stuff

The fun part

TPM

The reality

- Sounds great, real-world uses are a bit lacking
- **Complexity**
 - Thousands of pages
 - Hard to use
 - Can lead to implementation bugs
- **Brittleness**
 - PCRs are only stable in a very controlled scenario
 - Like the Bitlocker example earlier
- **Low adoption**
 - Browsers and password managers could also use TPMs
 - Instead they use secure enclaves on phones and separate authenticators
- **Who is the TPM protecting?**
 - High goals: protect the user from malware and evil maid attacks
 - Real-world usage: can be used to protect the vendor from the user (DRM, game anti-cheat, OS lock-in)
 - **TPM provides zero practical security**
 - ▶ Some interesting points, but read critically
 - ▶ Also the [discussion](#)
 - **Can you trust your computer?**

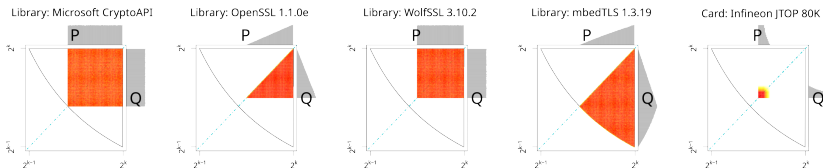
- Bus sniffing and interposing
- ROCA
- TPM-Fail and Minerva
- TPMScan

TPM

Bus sniffing and interposing

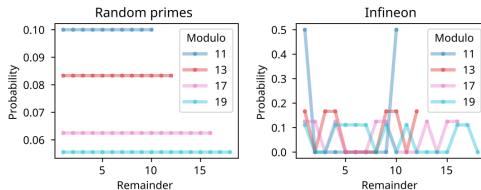
- dTPMs are physically separate from the CPU
 - Communicate over LPC, SPI or I2C buses
 - What if you connect to this bus and have some fun?
- TPMGenie
 - Just an Arduino
 - Can capture, modify and inject TPM commands and responses
- Any secrets transmitted in plaintext can be captured
 - Example: BitLocker key when no PIN/password is set
- HMAC session and policy sessions with parameter encryption prevent some harm

- The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli: Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, Vashek Matyas (CVE-2017-15361)
- They were investigating biases in RSA public keys
 - RSA: public key modulus $n = pq$
 - Looked at most-significant byte of p and q
 - Saw a *weird* distribution for Infineon TPMs



The distribution of the most significant bytes of a pair of RSA primes varies for different cryptographic libraries.

- The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli: Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, Vashek Matyas (CVE-2017-15361)
- They were investigating biases in RSA public keys
 - RSA: public key modulus $n = pq$
 - Looked at most-significant byte of p and q
 - Saw a *weird* distribution for Infineon TPMs
- The distribution looked even weirder modulo small primes



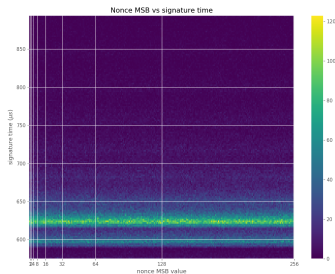
The distribution of RSA keys modulo small primes

- The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli: Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, Vashek Matyas (CVE-2017-15361)
- They were investigating biases in RSA public keys
 - RSA: public key modulus $n = pq$
 - Looked at most-significant byte of p and q
 - Saw a *weird* distribution for Infineon TPMs
- The distribution looked even weirder modulo small primes
- The prime generation was faulty
 - Way less entropy than it should have
 - Practical factorization possible
- **Huge** impact
 - TPMs
 - eIDs: Spain, Slovakia, Estonia, Austria, Bulgaria, Brazil, Italy, Kosovo, Malaysia, Poland, and Taiwan

TPM

TPM-Fail and Minerva

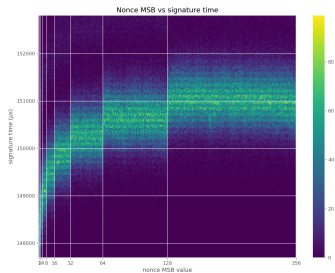
- **TPM-Fail: TPM meets Timing and Lattice Attacks:** Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, Nadia Heninger
- **Minerva: The curse of ECDSA nonces:** Jan Jancar, Vladimir Sedlacek, Petr Svenda, Marek Sys
- We were looking at testing all sorts of ECC primitives on smartcards
 - Invalid curve attacks
 - Small subgroup attacks
 - Domain parameter validation
 - Also looked at timing of ECDSA
- We expected
 - ? What is leaking on this graph?
 - ➔ The bit-length of the nonce
- Is that so bad?



TPM

TPM-Fail and Minerva

- **TPM-Fail: TPM meets Timing and Lattice Attacks:** Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, Nadia Heninger
- **Minerva: The curse of ECDSA nonces:** Jan Jancar, Vladimir Sedlacek, Petr Svenda, Marek Sys
- We were looking at testing all sorts of ECC primitives on smartcards
 - Invalid curve attacks
 - Small subgroup attacks
 - Domain parameter validation
 - Also looked at timing of ECDSA
- We saw
 - ❓ What is leaking on this graph?
 - ➡ The bit-length of the nonce
- Is that so bad?



- In ECDSA, you sample a random nonce during every signature
- Then you do scalar multiplication with that nonce as a scalar
- ? If it leaks
 - ✱ Your private key can be computed from the signature
- ? If parts of several nonces leak
 - ✱ Your private key can be computed from the signatures
- ? If you reuse it for two different messages
 - ✱ Your private key can be computed from the signatures
- ? If you hold it a bit too hard...
 - ✱ You see where I am going with this

- If you leak the bit-length of several nonces
 - Your private key can be computed from the signatures
 - Several hundred to several thousand signatures
 - Hidden Number Problem
 - Lattice reduction
 - ▶ Closest Vector Problem
 - ▶ Shortest Vector Problem
- We found this on a certified smartcard and several libraries
- TPM-Fail: Same thing on TPM chips
- More in lecture 6: **Timing and cache attacks**

- **TPMScan: A wide-scale study of security-relevant properties of TPM 2.0 chips:**
Petr Svenda, Antonin Dufka, Milan Broz, Roman Lacko, Tomas Jaros, Daniel Zatovic, Josef Pospisil
- They were examining the TPM ecosystem
 - What is supported/not-supported?
 - How is the performance?
 - How are vulnerabilities mitigated?
- *“We found a range of additional unreported nonce leakages in ECDSA, ECSCHNORR, and ECDAA algorithms in dTPMs and fTPMs of three vendors.”*
- *“The most serious discovered leakage allows extraction of the private key of certain Intel’s fTPM versions using only nine signatures with no need for any side-channel information, making the vulnerability retrospectively exploitable despite a subsequent firmware update.”*
- The lowest four bytes of nonces of ECDSA and ECSCHNORR algorithms were set to 0x00000001

- My ex-colleagues from the [CRoCS](#) lab are still collecting measurements from TPMs
- Please [measure and submit](#) yours
- The TPMScan paper was only possible with this support

Takeaways

- TPMs are powerful but complex
- How boot integrity works: measured vs verified
- How disk encryption relies on hardware
- TPMs are not unbreakable

Assignment 1

- Out soon™
- You will use your FIDO2 and TPM knowledge

Recommended reading

- For this lecture
 - Read [A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security](#), Chapter 3
 - ▶ A quick tutorial to TPMs and their use cases
- For next lecture
 - Read [SoK: Hardware-supported Trusted Execution Environments](#)
 - ▶ A very thorough overview of TEEs

References

- ☰ [A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security](#)
 - The book
- 🔗 [tpm-js](#)
 - Amazing interactive guide to TPMs
- 📖 [TCG Glossary](#)
 - If you are looking for a term
- 📖 [Trusted Platform Module 2.0 Library Specification, Version 185](#)
 - The spec, if you really need it
 - [Part 0: Introduction](#)
 - [Part 1: Architecture](#)
 - [Part 2: Structures](#)
 - [Part 3: Commands](#)
- 📖 [Simple TPM overview](#)
 - Simple TPM overview
- 📖 [Nice blog post on the hierarchies](#)
 - Nice blog post on the hierarchies
- 📖 [Linux kernel TPM docs](#)
 - Linux kernel TPM docs