PV204 Security technologies

Trusted Boot, TPM, SGX

Petr Švenda  ***svenda@fi.muni.cz***  ***@rngsec***
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on Cryptography and Security

Overview

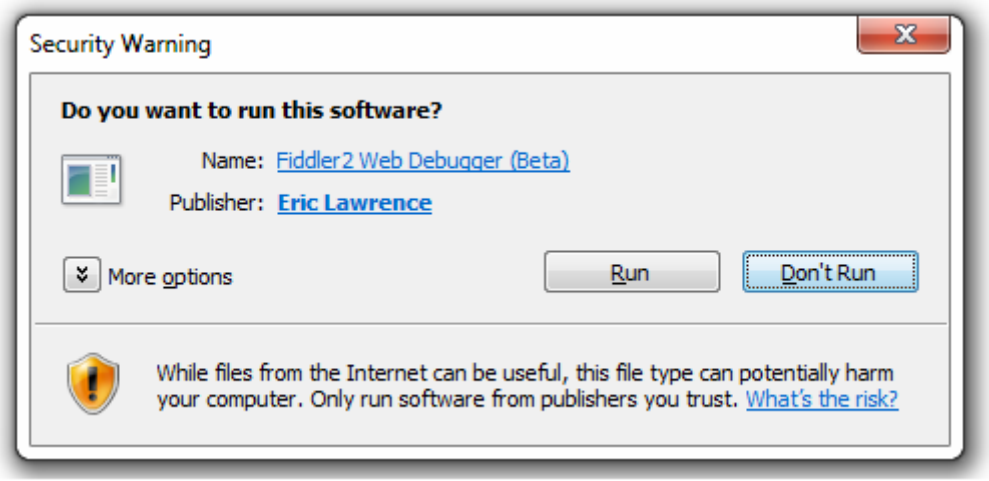
- Booting chain of programs
- BIOS as root of trust
- Verified and Measured boot
- Trusted boot in the wild
  - Trusted Platform Module
  - Chromium, Windows 8/10, UEFI…
- Dynamic root of trust
  - Intel's TXT, SGX

Motivation – untrusted host platform

- Traditional role of operating system
  - Isolate processed
  - Manage privileges, authorize operations
- But how to deal with
  - Debugger, disassembler
  - Intercepted multimedia output
  - Malware run along with banking app
  - Keyloggers, Evil maid
  - System administrators, Service providers
  - …

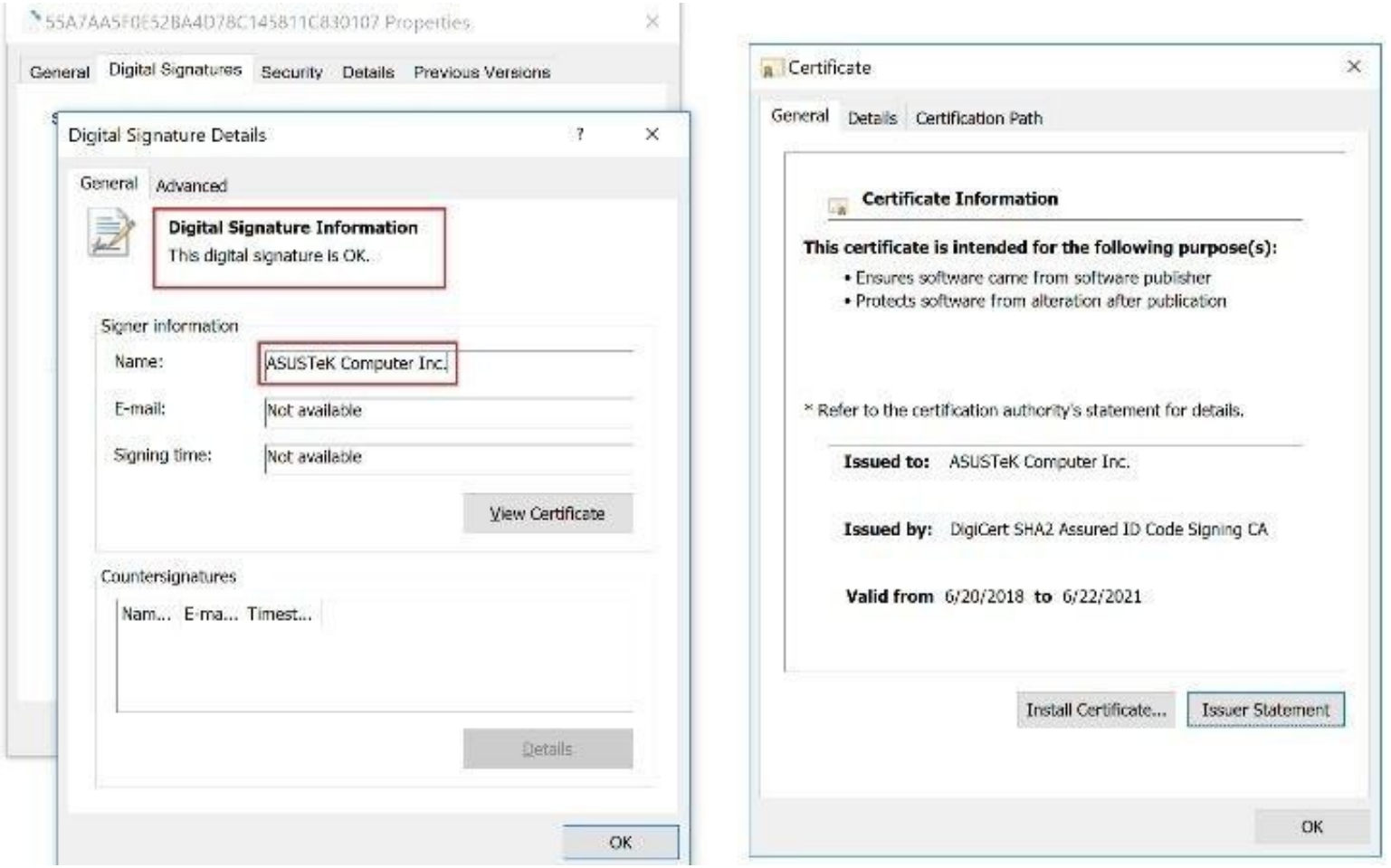
Solution?

- Code signing (e.g., Microsoft AuthentiCode)
  - Application binary is signed, PKI used to verify certificate
  - If not signed, user is notified
  - Mandatory signing for selected applications (drivers…)



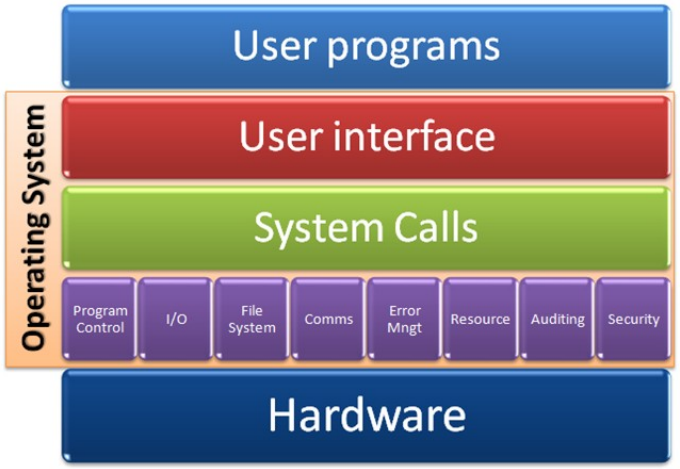
Signed == Secure?

Signed == Secure?



Trust in program's functionality

- Trust in a program's code?
  - Signed code can still contain bugs and vulnerabilities
- Trust only in a program's code?
  - Underlying OS layers
  - Underlying firmware
  - Underlying hardware
  - Memory used by the program
  - Other code with access to the program's memory/code
  - …
- The program is almost never executed "alone"



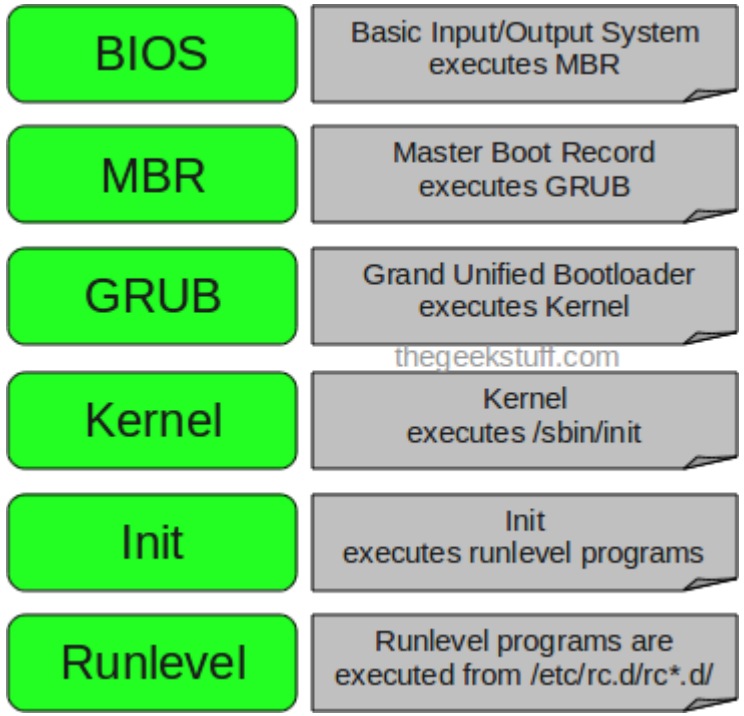
Problem statement

- How to make sure that valid programs run only within valid environment?

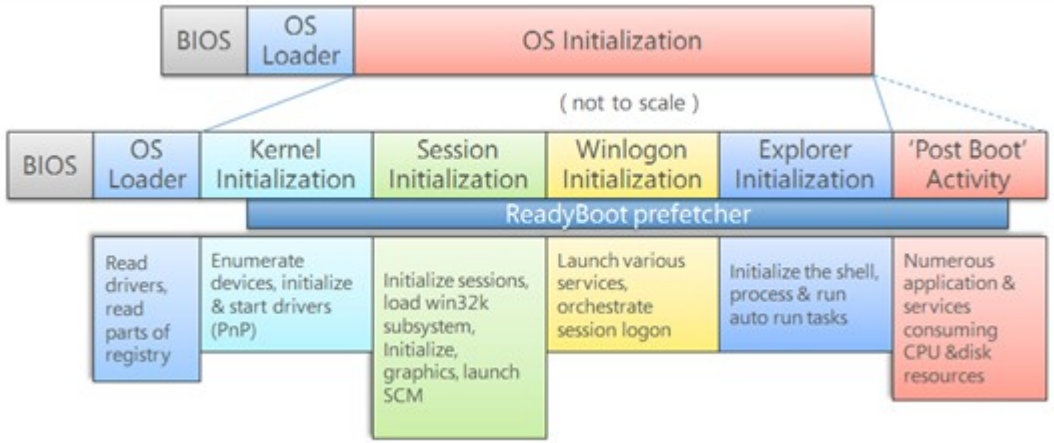
1. Is it possible to start valid environment on previously compromised machine?
2. Is it possible to prevent tampering of apps against attacker with physical access?
3. How to prove to remote party what apps are running on local machine?

Classical boot chain

Linux



Windows (7)



How to detect that BIOS or OS Loader was modified? (evil maid, bootkit…)

http://www.thegeekstuff.com/2011/02/linux-boot-process/
http://social.technet.microsoft.com/wiki/contents/articles/11341.the-windows-7-boot-process-sbsl.aspx

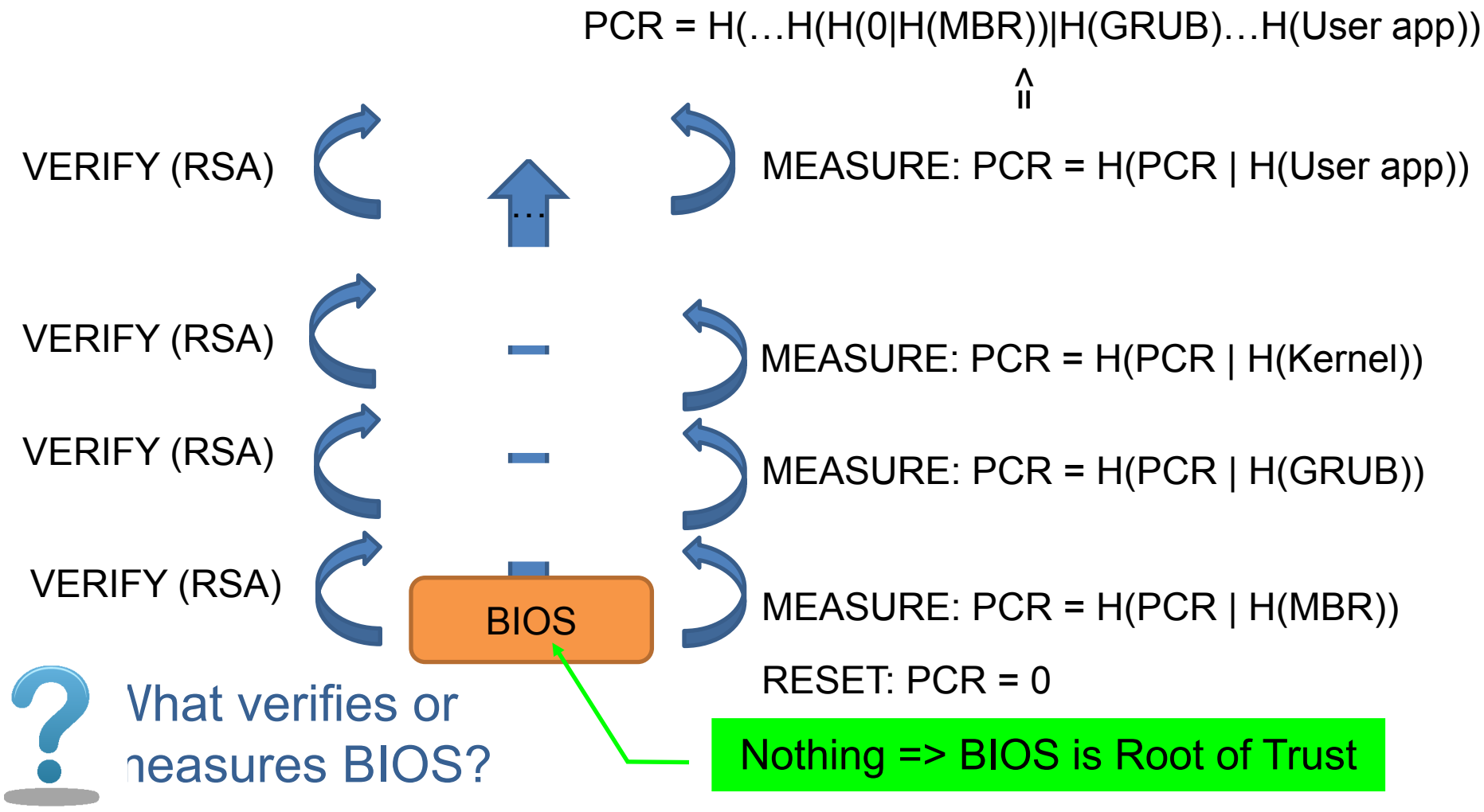
How to arrive at expected chain of apps?

1. Just trust the whole boot process
2. Make all applications in protected read-only memory
   - If read-only => cannot be updated. But is it really what is running?
3. Signature-based approach: “Verified boot”
   - Before next app is executed, its signature is verified
   - Requires valid (unforged) public key (integrity)
   - Requires trust to owner of private key (signs only valid applications)
4. Create un-spoofable log what executed: “Measured” boot
   - Before next app is executed, its hash (“measurement”) is computed added to un-spoofable log (TPM’s PCR)
   - Will NOT prevent run of unwanted app, but environment cannot lie about what was executed (after-the-fact examination)
   - Requires (protected) log storage (Trusted Platform Module)
   - May require authentication of log (Remote attestation)

Trusted boot

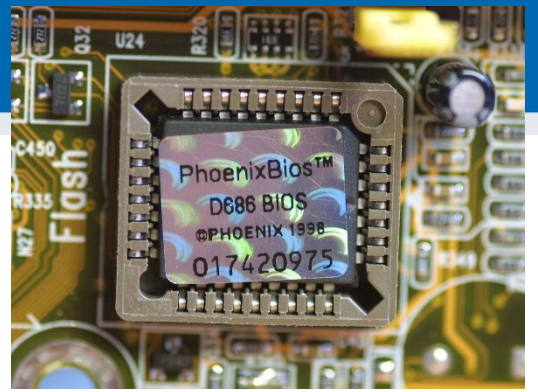
“Verified” boot

“Measured” boot



Root of trust (for verified/measured boot)

- Verified and Measured boot need some root of trust
  - Initial piece of code that nobody verifies/measures
- Static root of trust
  - Start building trusted chain after reset of whole device
- Dynamic root of trust
  - Start building trusted chain without reset of device (faster)
- What can be root of trust?
  - static root of trust: BIOS, UEFI firmware, Intel Boot Guard
  - dynamic root of trust: Intel TXT, Intel SGX
- Root of trust requires special protection
  - As nobody verifies than nobody will detect eventual modification



BIOS as root of trust

- First code executed on CPU of target machine
- Privileged access to hardware
  - E.g., can write into memory of OS code via DMA
- Provides code for System Management Mode (SMM)
  - Routines executed during the whole platform runtime
  - x86 feature since 386, all normal execution is suspended
  - Used for power management, memory errors, hardware-assisted debugger…
  - Very powerful mode (=> also target of “ring -2” rootkits)

BIOS – security considerations

- How BIOS verifies integrity of next module to run?
- Where public key(s) for verification are stored?
- How to handle updates of signing keys?
- How BIOS checks signatures on its own updates?
- How BIOS can be compromised?

How BIOS can be compromised?

1. Maliciously written by BIOS vendor (backdoor)
2. Replacement of genuine BIOS by malicious one
   - By physical flash (SPI programmer) of BIOS code
   - By lack of flashing protection mechanism by original BIOS
   - By code logic flaws in BIOS locking mechanisms
3. Modification of other code/data used by BIOS
   - Bug in parsing unsigned data…

- Currently used protections:
  - Chipset-enforced protection of flash memory with BIOS
  - BIOS signature verification before new version is written
  - Hardware-aided check of executed code (TPM, TXT, SGX)
  - Check of BIOS signature before execution by CPU (IBG)

Attacks against BIOS locks

1. Attacks typically via BIOS code vulnerability
   - BIOS usually does not takes (much) user input, but may parse BIOS update blob with some parts unsigned (logo)
   - Buffer overflow in logo parsing => Locks are not locked yet => write own BIOS
   - http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf
2. Write into flash memory by SPI programmer

Which one is more serious? Different attacker models
1. Is remote, but patchable
2. Is local attacker, but requires design changes to prevent

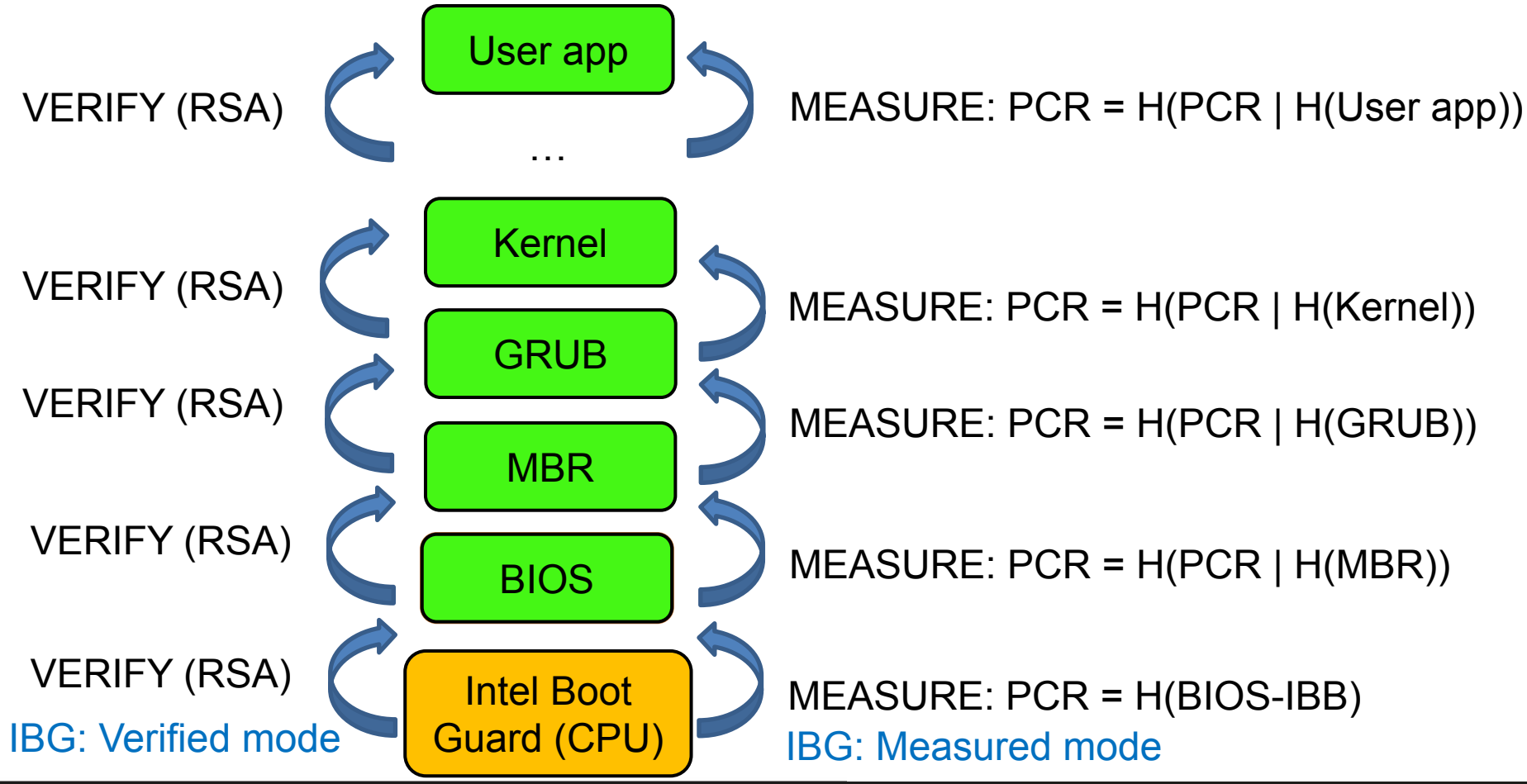
INTEL BOOT GUARD (IBG)

Intel Boot Guard (IBG)

- Recently (2014) introduced feature to protect BIOS
  - Piece of trusted processor-provided, ROM-based code
  - Runs first after reset, verifies *Initial Boot Block (IBB)*

1. "Measured" boot mode (TPM-based)
   - Passively extends TPM's PCRs by hash of IBB
2. "Verified" boot mode (digital signature)
   - OEM vendor hardcodes public key via fuses into CPU
   - Intel Boot Guard checks signature of IBB by OEM's key
   - Only vendor-approved IBB=>BIOS=>OS is executed
3. Combination of measured and verified mode

Intel Boot Guard – new root of trust



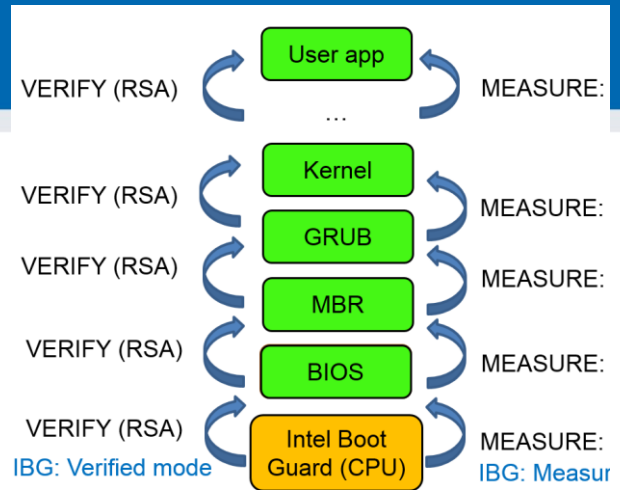
Intel Boot Guard – security improvements

- What attacks are mitigated by Intel Boot Guard?
- Direct BIOS flash by SPI programmer
  - Mitigated, signature/measurement mismatch
- Remote change of BIOS / BIOS data
  - Mitigated, signature/measurement mismatch
- Other bug(s) in BIOS code
  - Not mitigated, signed code still contains bug

- Any new attacks opened by IBG?

How hard is to incorporate backdoor?

- OEM vendor can sign backdoored BIOS
  - But multiple OEM vendors exist, open-source coreboot
- Intel Boot Guard is written by Intel only
  - But OEM fuses own verification public key, right?
  - But it is the IBG code that actually verifies
- Trivial backdoor (inside IBG code inside CPU)
  - if (IBB[SOME_OFFSET] == BACKDOOR_MAGIC) then always load provided BIOS (no signature check)
  - Or possibly verify by some other public key (secure even when BACKDOOR_MAGIC is leaked)

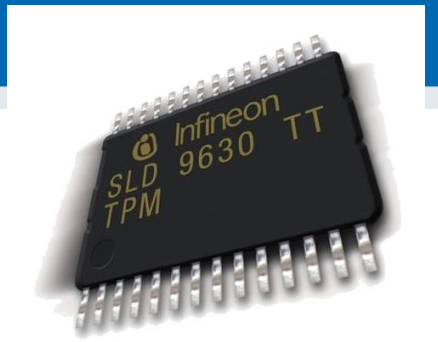


Short summary

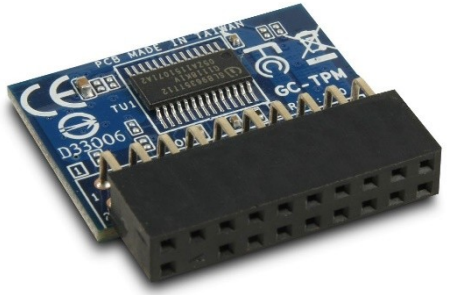
- Signature-based “verified” boot approach
  - Whitelisting approach – run only what is signed
  - Robust signature process needed (trust in private key owner)
  - Integrity of verification public key is critical
  - Key management is necessary (multiple keys, key updates)
- “Measured” boot approach
  - Un-spoofable log of hashes of executed code
  - Can be remotely verified (remote attestation, explained later)
- Root of trust needs to be protected
  - Historically was BIOS (+ update signatures + write locks)
  - Recently Intel Boot Guard inside CPU (signature of BIOS)

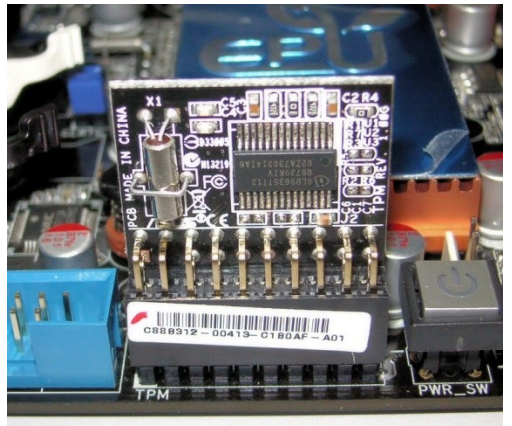
TRUSTED PLATFORM MODULE

TPM hardware

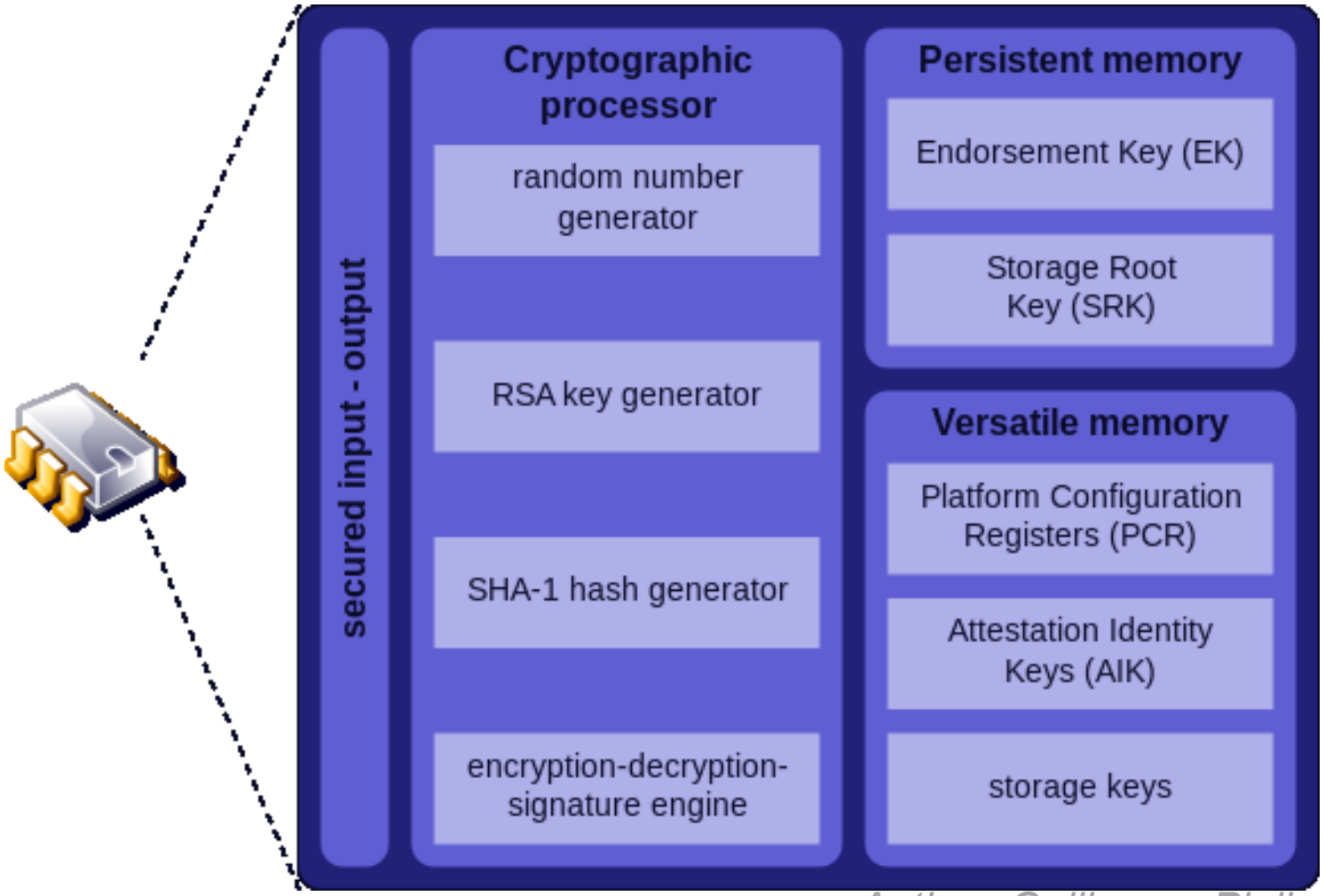


- Cryptographic smart card connected/inside to device
  - Secure storage, secure crypto environment…
  - (But not programmable JavaCard ☺)
- Physical placement
  1. Additional chip on motherboard
  2. Incorporated inside CPU
  3. Incorporated in peripheral (Ethernet card)
- Accessed during boot time
  - “Measured” boot (TPM’s PCR registers)
  - Bitlocker encrypted drive keys
- Accessed later (private key operation)





Trusted platform module



Author: Guillaume Piolle

Trusted Platform Module (TPM)

- ISO/IEC 11889 standard for secure crypto-processor
- Versions published by Trusted Computing Group
  - https://trustedcomputinggroup.org
  - TPM 1.2 (2011)
  - TPM 2.0 (2016, not compatible with 1.2, but downgrade switch)
- Tools to communicate with TPM
  - Windows: Microsoft PCPTool, TSS.MSR, Windows API
  - Linux: tpm_tools, tpm2_tools, GUI TPMManager

TPM 1.2 vs. TPM 2.0

- TPM 2.0 introduced algorithm flexibility (no fixed SHA-1)
  - If (some) algorithm is broken, no need to create “TPM 3.0”
- TPM 2.0 often supports legacy API 1.2 (switch in BIOS)
- TPM 2.0 seems to focus on IoT-like devices (support TLS)

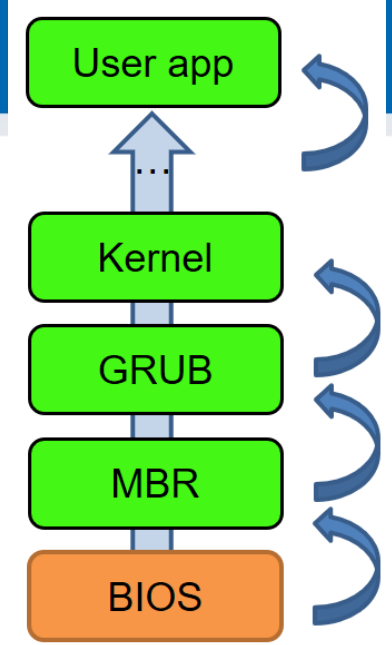
	TPM 1.2	TPM 2.0
Algorithms	SHA-1, RSA	Agile (such as SHA-1, SHA-256, RSA and Elliptic curve cryptography P256)
Crypto Primitives	RNG, SHA-1	RNG, RSA, SHA-1, SHA-256
Hierarchy	One (storage)	Three (platform, storage and endorsement)
Root Keys	One (SRK RSA-2048)	Multiple keys and algorithms per hierarchy
Authorization	HMAC, PCR, locality, physical presence	Password, HMAC, and policy (which covers HMAC, PRC, locality, and physical presence).
NV RAM	Unstructured data	Unstructured data, Counter, Bitmap, Extend

https://en.wikipedia.org/wiki/Trusted_Platform_Module

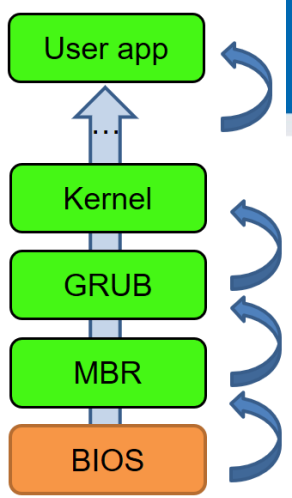
Provided security functions

I. “Measured” boot with remote attestation
   – Provide signed log of what executed on platform (PCR)
II. Storage of keys (disk encryption, private keys…)
   – Can be additionally password protected
III. Binding and Sealing of data
   – Encryption key wrapped by concrete TPM’s public key
IV. Platform integrity
   – Software will not start if current PCR value is not right

TPM PCR



- Platform Configuration Register (PCR)
- Measurement cumulatively stored in PCR
  - measurement = SHA1(next block to execute)
  - PCR[i] = SHA1(PCR[i] | new_measurement)
  - Current block measure & store next before passing control
- PCR cannot be erased until reboot
  - Every part that was executed is stored
  - Possible to perform after-the-fact verification what executed
- Idea: boot what you want, but PCR will hold trace
- Multiple PCRs to support finer grained reporting



Platform attestation – PCR registers

- **W:** PCPTool.exe GetPCRs
- **L:** cat `find /sys/class/ -name "tpm0"`/device/pcrs

Table 12-1. Example PCR Allocation

PCR Number	Allocation
0	BIOS
1	BIOS configuration
2	Option ROMs
3	Option ROM configuration
4	MBR (master boot record)
5	MBR configuration
6	State transitions and wake events
7	Platform manufacturer specific measurements
8–15	Static operating system
16	Debug
23	Application support

```
bug>PCPTool.exe GetPCRs
<PCRs>
  <PCR Index="00">8cb1a2e093cf41c1a726bab3e10bc1750180bbc5</PCR>
  <PCR Index="01">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
  <PCR Index="02">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
  <PCR Index="03">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
  <PCR Index="04">1e3c5e15b5f023765147535e092d22d7c17421e1</PCR>
  <PCR Index="05">75acbe8a48ba02a85d6301b33005d08678176c87</PCR>
  <PCR Index="06">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
  <PCR Index="07">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
  <PCR Index="08">0000000000000000000000000000000000000000</PCR>
  <PCR Index="09">0000000000000000000000000000000000000000</PCR>
  <PCR Index="10">0000000000000000000000000000000000000000</PCR>
  <PCR Index="11">ebb98df76613280f20dc38221143a9e727399486</PCR>
  <PCR Index="12">67afac5ca0fc6c9a3d881d681121f7d43d0c7128</PCR>
  <PCR Index="13">be1d9bd7318a9140b26f00a5283f37a6111bb1e5</PCR>
  <PCR Index="14">7f599cd09efefc7422085a0f490f8f1cba8761a8</PCR>
  <PCR Index="15">0000000000000000000000000000000000000000</PCR>
  <PCR Index="16">0000000000000000000000000000000000000000</PCR>
  <PCR Index="17">ffffffffffffffffffffffffffffffffffffffff</PCR>
  <PCR Index="18">ffffffffffffffffffffffffffffffffffffffff</PCR>
  <PCR Index="19">ffffffffffffffffffffffffffffffffffffffff</PCR>
  <PCR Index="20">ffffffffffffffffffffffffffffffffffffffff</PCR>
  <PCR Index="21">ffffffffffffffffffffffffffffffffffffffff</PCR>
  <PCR Index="22">ffffffffffffffffffffffffffffffffffffffff</PCR>
  <PCR Index="23">0000000000000000000000000000000000000000</PCR>
</PCRs>
```

Remote attestation of platform state

- So you measured your boot. How to prove your state to remote party?

- Idea:
  1. Take PCR values (inside TPM)
  2. Sign it (inside TPM) by TPM's private key (AIK)
  3. Remote party holds public key and can verify signature
     => trust in authenticity of PCR values

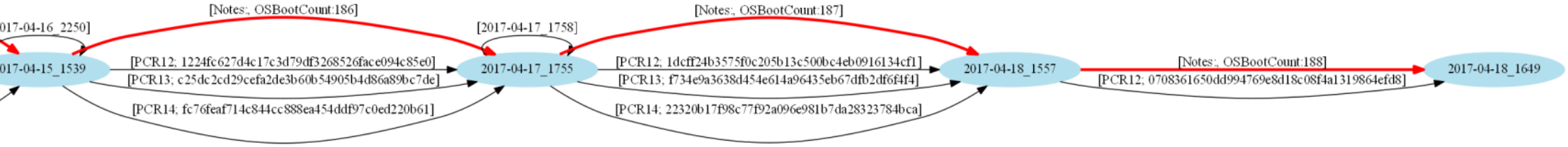
TPM platform info

- Provides information about your platform state
- Included in PCR12 (Operating System information)

```
<PlatformCounters>
  <OsBootCount>44</OsBootCount>
  <OsResumeCount>2</OsResumeCount>
  <CurrentBootCount>0</CurrentBootCount>
  <CurrentEventCount>66</CurrentEventCount>
  <CurrentCounterId>179136858</CurrentCounterId>
  <InitialBootCount>0</InitialBootCount>
  <InitialEventCount>64</InitialEventCount>
  <InitialCounterId>179136858</InitialCounterId>
</PlatformAttestation>
```

Reboot =>

```
<PlatformCounters>
  <OsBootCount>45</OsBootCount>
  <OsResumeCount>0</OsResumeCount>
  <CurrentBootCount>0</CurrentBootCount>
  <CurrentEventCount>67</CurrentEventCount
  <CurrentCounterId>179136858</CurrentCou
  <InitialBootCount>0</InitialBootCount>
  <InitialEventCount>67</InitialEventCount>
  <InitialCounterId>179136858</InitialCounter
</PlatformAttestation>
```



TRUSTED BOOT – REAL IMPLEMENTATIONS

Verified boot - Chromium OS

- Starts with read-only part of firmware/BIOS (root of trust)
  - Cannot be forged, but also cannot be not updated
  - Contains permanently stored root RSA public key
- “Verified” boot strategy is used
  - Verifies that all executed code is from Chromium OS source tree
  - Code signatures verified by (shorter) keys signed by root key
    - speed tradeoff + possibility to update compromised keys
- Does not completely prevent user to boot other OSes
  - Developer mode turned on => signature on kernel not checked
  - TPM is used to provide mode reporting (normal/devel/recovery)
- https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot
- https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-crypto

Chromium OS uses of TPM

- Limited remote attestation (PCR[0] used)
  - to store developer and recovery mode switches
- Prevent rollback attack
  - Prevented by strictly increasing version of key & firmware
  - Version is written in TPM's NV RAM location, only read-only firmware can update this location
  - Key version prevents update to older (compromised) key
  - Firmware version prevents update to vulnerable firmware
- Store selected user's private keys (secure storage)
- Wrap selected disk encryption keys by TPM's system key
- https://www.chromium.org/developers/design-documents/tpm-usage

Secured and Trusted Boot

UEFI SECURE BOOT

UEFI secure boot principles

- Platform key (RSA 2048b, PK) for authentication of platform owner
- Key exchange keys (KEKs) for authentication of other components (drivers, OS components…)

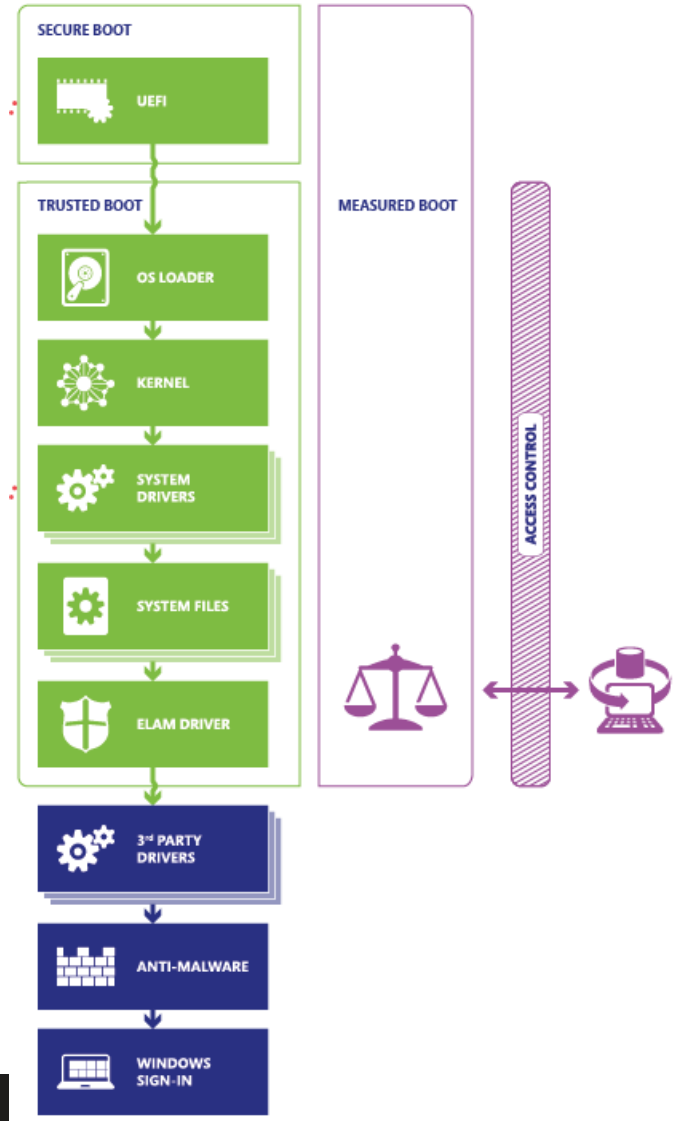
1. “Setup" mode – platform key (PK) is not loaded yet
   - Everybody can write its own platform key (become owner)
   - Once PK is written, switch to “user” mode
2. “User” mode
   - New keys (PKs, KEKs) can be written only if signed by PK
   - New software components loaded only if signed by KEKs

Secured and Trusted Boot

WINDOWS 8/10 TRUSTED BOOT

Windows 8/10 trusted boot

- Certified Windows 8/10 devices have trusted boot by default
  - "Verified" boot used (UEFI+OS sign)
  - "Measured" boot used (TPM)
- TPM PCRs used for measurements
- TPM used for keys protection
  - BitLocker disk encryption key
  - ROCA relevant CVE-2017-15361
    - If Infineon TPM used, patch!



http://technet.microsoft.com/en-US/windows/dn168167.aspx

Usage of TPM in BitLocker (disk encryption)

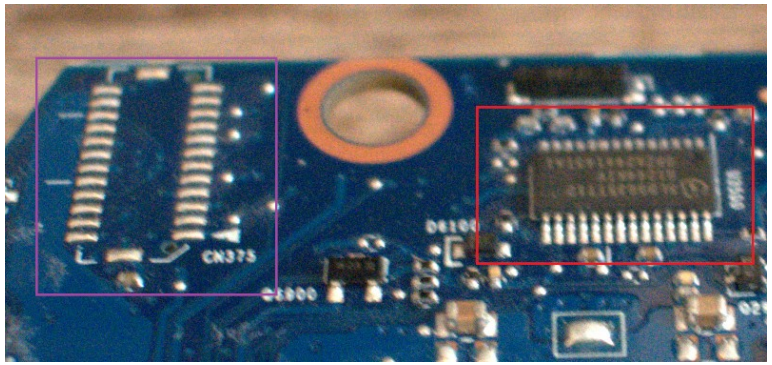
- Source of Volume Master Key (VMK)

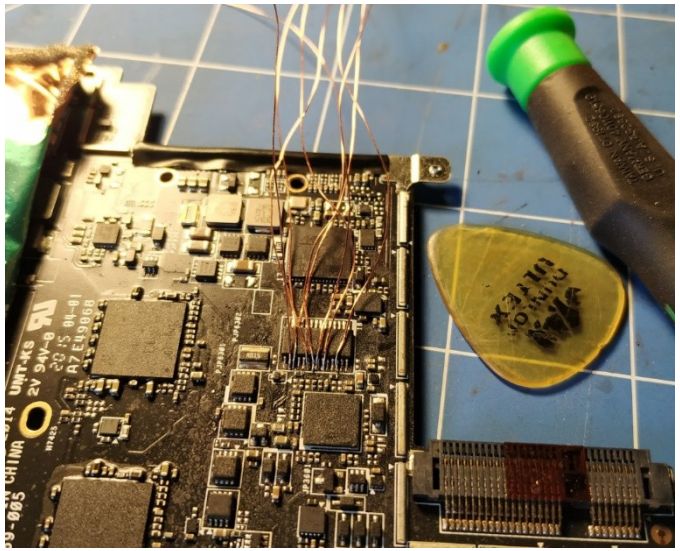
Source	Identifies	Security	User Impact
TPM only	What it is	Protects against software attacks, but vulnerable to hardware attacks.	None
TPM + PIN	What it is + What you know	Adds protection against most hardware attacks as well.	User must enter PIN each boot
TPM + USB key	What it is + What you have	Fully protects against hardware attacks, but vulnerable to stolen USB key.	User must insert USB key each boot
TPM + USB key + PIN	What it is + What you have + What you know	Maximum level of protection.	User must enter PIN and insert USB key each boot
USB key only	What you have	Minimum level of protection for systems without TPM, but vulnerable to stolen key.	User must insert USB key each boot

M. Russinovich et. al., Windows Internals Part 2, 6th Edition

Attack: Sniffing keys for BitLocker

- Nice writeup how to sniff BitLocker key when send from TPM to OS, then decrypt disk image
  - https://pulsesecurity.co.nz/articles/TPM-sniffing





- Add slide with reasoning about the attack

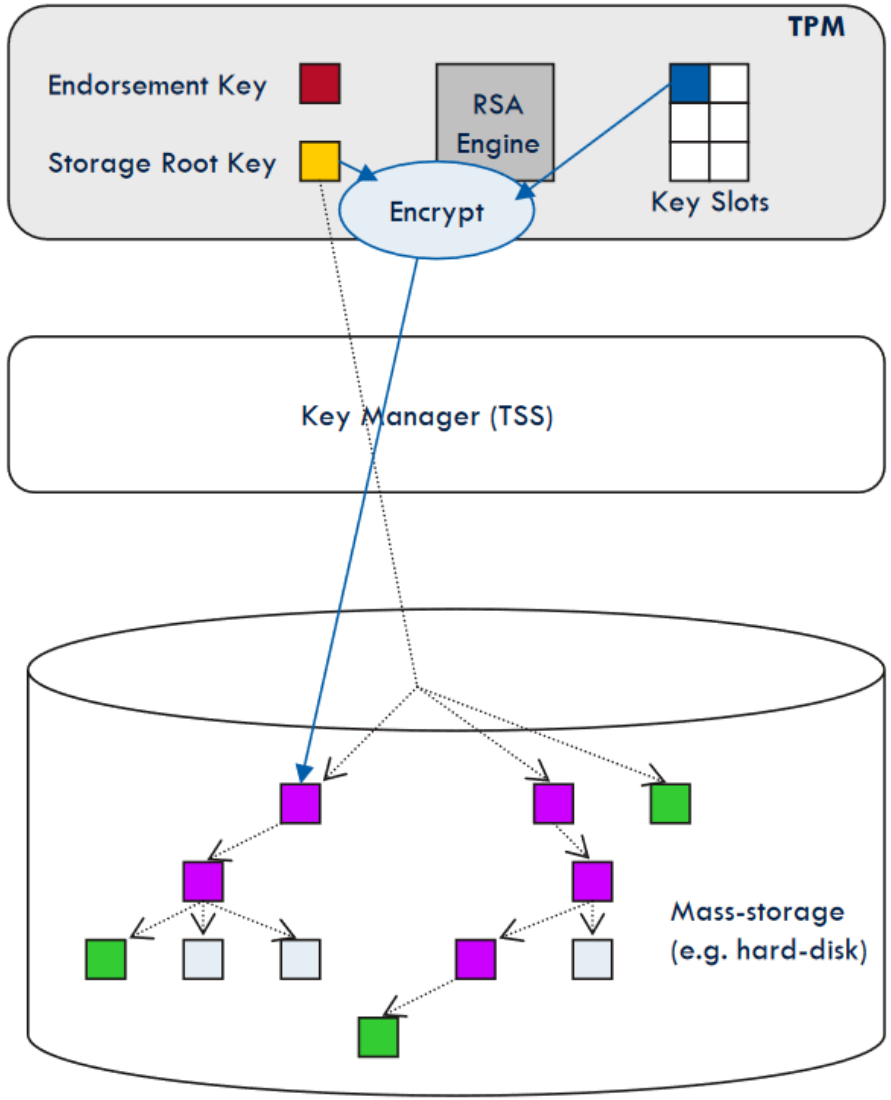
BASIC COMPONENTS

TPM keys

- Endorsement key (EK)
  - Generated during manufacturing, permanent
  - Remain in TPM device during whole chip lifetime
- TPM Storage Root Key (SRK)
  - Generated by use after taking ownership
  - New Storage root key can be generated after TPM clear
  - Used to protect TPM keys created by application
- Various delegate keys
  - Separate keys signed/wrapped by EK, SRK…
  - Application can generate and store own keys
  - Good practice: do not have single key for everything

TPM storage keys

- Application keys encrypted under SRK
- Exported as protected blob
- Stored on mass-storage
- If needed, decrypted back and placed into slot
- Key usable until removed



http://www.cs.unh.edu/~it666/reading_list/Hardware/tpm_fundamentals.pdf

TPM policy

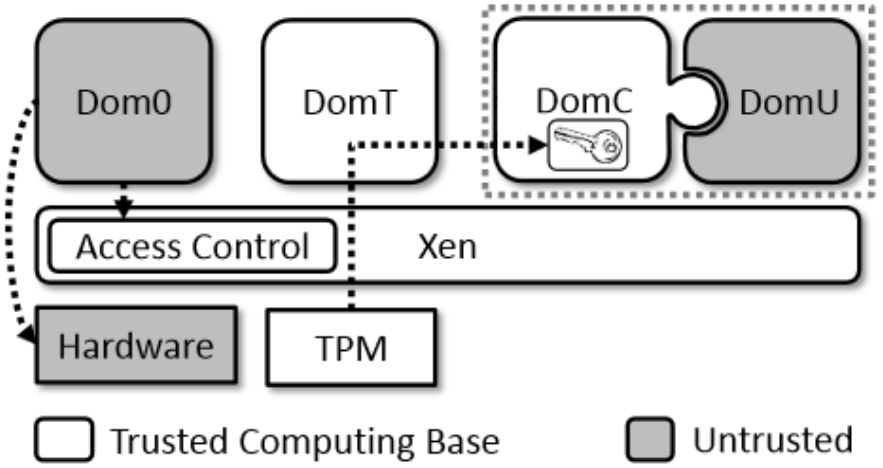
- TPM releases secret only when PCR contains particular value
- Enforcement even in measured-only mode
  - Key is not released if unexpected component was started (started => is included in measurements)
- Conditions can use ANDs and ORs
- How to handle policy updates?
  - Change policy of state only from already valid state

Programming with TPM

- TPM Platform Crypto-Provider Toolkit
  - https://research.microsoft.com/en-us/downloads/74c45746-24ad-4cb7-ba4b-0c6df2f92d5d/
  - Source code examples for Attestation and Trusted boot
- The TPM Software Stack from Microsoft Research (Java, C#)
  - https://github.com/Microsoft/TSS.MSR
- Measured Boot Tool
  - https://mbt.codeplex.com/, List of most important functions, example code
  - Demonstrates TPM secure boot and remote attestation on Windows 8/10
- TrouSerS: open-source stack for TCG
  - Linux version (2008)
    - http://trousers.sourceforge.net/
    - https://sourceforge.net/projects/trousers/files/tpm-tools/
  - Windows port (2010)
    - http://security.polito.it/trusted-computing/trousers-for-windows/

Usage of TPM in cloud-computing

- Combination of virtualization and trusted computing
- Modified Xen hypervisor used to make standard TPM available for secret-less virtual machine
- Results in significant decrease in the size of trusted computational base (TCB)



http://bleikertz.com/research/acns2013.pdf

DYNAMIC ROOT OF TRUST

Static Root of Trust Measurement (SRTM)

- Start trusted immutable piece of firmware
  - E.g., BIOS loader or Intel Boot Guard
- Initiates measurement process
  - Integrity of every next component is added to TPM's PCRs
  - Start → BIOS → PCI EEPROM → MBR → OS …
- But do we need to start (trusted boot) only after reboot?
  - Takes relatively long time
  - Can we execute the same process, but dynamically?
  - Can we exclude long chain (BIOS, PCI…)?
    - Long chain => large Trusted Computing Base (TCB)!

Dynamic Root Trust Measurement (DRTM)

- Launch of measured environment at any time
  - "Late lunch" option
  - No need to reset whole platform
  - Can be also terminated after some time
- Measurement process similar to static root of trust
  - Application trust chain executed from dynamic root
- Implementation of DRTM
  - Intel's TXT (not used much in practice, server CPUs typically)
  - Intel's SGX (all Skylake processors and newer, from 2015)

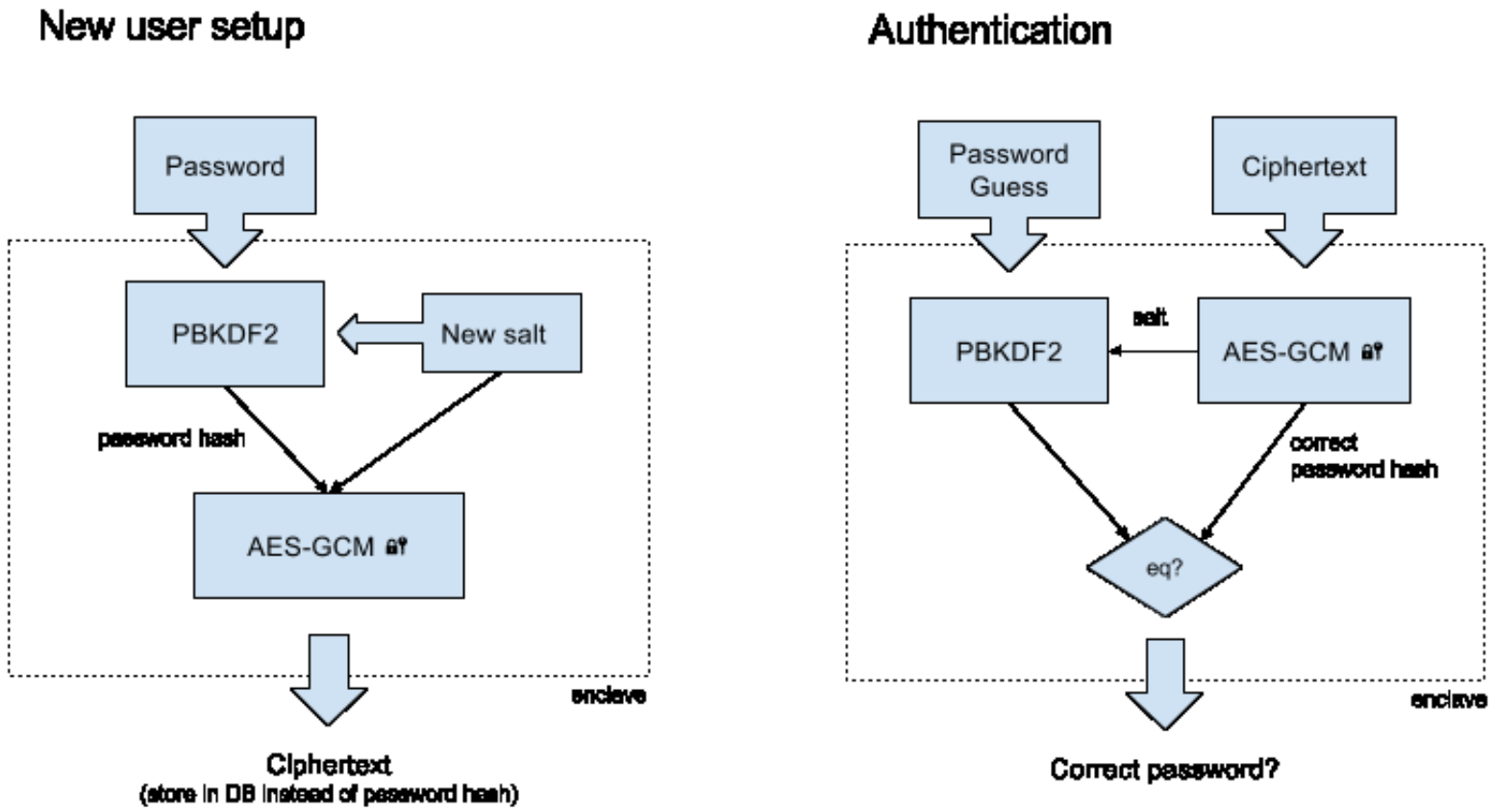
Intel's SGX : Security enclave

- Intel's Software Guard Extension (SGX)
  - New set of CPU instructions intended for future cloud server CPUs
- Protection against privileged attacker
  - Server admin with physical access, privileged malware
- Application requests private region of code and data
  - Security enclave (4KB for heap, stack, code)
  - Encrypted enclave is stored in main RAM memory, decrypted only inside CPU
  - Access from outside enclave is prevented on CPU level
  - Code for enclave is distributed as part of application
- Trusted Computing Base significantly limited! ☺
  - But proprietary Intel code inside CPU ☹

Intel's SGX – some details

- EGETKEY instruction generates new enclave key
  - SGX security version numbers
  - Device ID (unique number of CPU)
  - Owner epoch – additional entropy from user
- EREPORT instruction generates signed report
  - Local/remote attestation of target platform
- Debugging possible if application opt in
- Enclave cannot be emulated by VM

SGX hardened password verification



https://jpp.io/2016/01/17/using-sgx-to-hash-passwords/

Intel SGX is very active research area

- Many small enclaves to cover whole program
  - User-annotated code split into many enclaves (“microns”)
    - Secure interaction between microns (attest, auth. encryption)
  - Tor, H2O, FreeTDS and OpenSSL successfully transformed
    - 2685, 154, 473 and 307 LOC changes required respectively
    - TCB only 20KLOC, PANOPLY specific overhead 24%
- Memory randomization of code inside enclave
  - SGX program modified with custom LLVM compiler
  - Added in-enclave loader for ASLR & swDEP (2703 LOC)
  - Code&data split into 32/64B units randomized separately
- Full library OS based on SGX (Haven, Graphene-SGX)

Recent attacks against SGX

- SGX is not a silver bullet
- Vulnerable to side-channels
  - Attacker with physical access explicitly excluded from attacker model
  - Impacted by Spectre attack (2017)
    - https://github.com/lsds/spectre-attack-sgx
    - https://github.com/osusecLab/SgxPectre
  - Impacted by Foreshadow attack (CVE-2018-3615) https://foreshadowattack.eu/
    - Reading out attestation private key
- Bugs of enclave code are still problem (developer)
- Not everything is running inside enclave (other code, user input…)

Programming with Intel's SGX

- Intel SGX SDK
  - https://software.intel.com/en-us/sgx-sdk
  - 6th generation core processor (or later) based platform with SGX enabled BIOS support
- Example: Hardened password hashing
  - https://jbp.io/2016/01/17/using-sgx-to-hash-passwords/
  - https://github.com/ctz/sgx-pwenclave
- More SGX info
  - http://theinvisiblethings.blogspot.cz/2013/08/thoughts-on-intels-upcoming-software.html
  - http://theinvisiblethings.blogspot.cz/2013/09/thoughts-on-intels-upcoming-software.html

TRUSTED COMPUTING - CRITIQUE

Trusted computing - controversy

- For whom is your computed trusted?
  - Secure against you as an owner?
- Is TC preventing users to run code of their choice?
  - Custom OS distribution?
  - Open OEM system – locked on first installation
  - Physical switch to unlock later
- Why some people from *Trusted Computing* consortium think that Trustworthy Computing might be better title?

Trusted computing - controversy

- R. Anderson, `Trusted Computing' FAQ (2003)
  - http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html
- J. Edge, UEFI and "secure boot“
  - http://lwn.net/Articles/447381/
- R. Stallman, Can You Trust Your Computer?
  - https://www.gnu.org/philosophy/can-you-trust.html
- Selected problems addressed in current designs

Summary

- Two principal solutions for trusted boot
  - Verified boot (signatures) and Measured boot (PCR+RA)
- Start from clean (and trusted) point
  - Allow only intended software to run
  - Or prove what actually executed
- Additional hardware inside motherboard / CPU provides wide range of new possibilities (TPM)
- Size of Trusted Computing Base matters (TPM/SGX)
- Controversy about implication of trusted boot
  - Who owns and control target platform