Trusted computing

Jesse Walker, Ph.D.
Oregon State University
Agenda

Problem statement

The measurement problem: measuring what program loaded
The attestation problem: reading our measurement
The policy problem: how only authorized programs execute
The ownership problem: who gets to authorize
Problem: Electronic wallet

Why should we believe our software securely implements our electronic wallet?

Why do we believe the software is not infected by malware designed to steal our credit cards, debit cards, banking password, Bitcoins, etc?
Problem: modern vehicles have over 200 microprocessors

Why do we believe the software running on them is the software installed by the manufacturer?

Why do we believe this software is not infected with malware, e.g., demanding a ransom to avoid an intentional crash?
Problem: Cellular radio

It is illegal to deploy a LTE radio that operates outside parameters prescribed by the FCC

Why should the FCC believe the software running on a radio conforms to its specifications?

What about hobbyist software? What about after malware infects the software controlling the radio?
Problem: Electronic Voting Machines

Numerous studies and the 2016 elections focused scrutiny on the integrity of electronic voting machine software.

Why should we believe the voting machine software is the software intended to run on the machine?

What kind of easily scalable test could discover whether the voting machine loaded the intended software instead of rogue software installed by malicious agents?
Problem statement

There is a broad need to somehow verify that a system’s software is the software it’s supposed to be running.

How could we possibly accomplish this?

- Trust that this is so? The existence of malware and other hacks widespread in the Internet shows this is ineffective.
- Ask the software whether it is the “right” software? Compromised software might lie, and how could it know what is “right?”
- Install anti-virus software? Anti-virus software can’t identify new malware, and can be compromised itself.

Is there some way to do better?
Problem analysis

The problem seems to require us to solve a number of sub-problems

• We need some efficient, secure, and automatic way to identify what software is loaded and record the result – this is the measurement subproblem

• We need an efficient, secure way, and scriptable way to read the results of our measurements – this is the attestation subproblem

• We need some efficient and secure way to verify whether or not the read measurements identify the intended software and configuration parameters, and what degree of verification we require to run – this is the policy subproblem

• We need a secure procedure to authorize both what software will get loaded and whether the computing device actually loads that software instead of some other software – this is the ownership subproblem
Measurement problem idea!!

Maybe we can somehow use cryptographic hash function to create unique a “fingerprint” for every program loaded into memory.

A hash function is secure if finding collisions, 2nd-preimages, and preimages is computationally infeasible.
Why might this work?

We assume cryptographers know how to construct cryptographic hash functions.

By the 2nd preimage resistance property, for any program \textit{loaded-program} for which 
\[ \text{hash}(\text{loaded-program}) = \text{hash}(\text{intended-program}), \]
we must have \text{loaded-program} = \text{intended-program} with very high probability.

Or else \textit{hash} is not a cryptographic hash function, or an insecure process computed the digest value \text{hash}(\text{loaded-program}).

So the only measurement problem is how to compute the hash digest value over the \textit{loaded-program} securely.
Elements of a solution: Trusted Computing Base (TCB)

A solution can be constructed using special hardware called the Trusted Computing Base or TCB

The elements of the TCB used to solve the measurement problem are

• A Program Configuration Register (PCR)
• A hardware-based implementation of a cryptographic hash function
• The ability to lock and unlock regions of memory

Architecture:

• The cryptographic hash function can hash any region of memory
• The cryptographic hash function writes its digest into the PCR
• Only the cryptographic hash function can write into the PCR, but
• Any program running on the computing device can read from the PCR
• Lock memory after loading a program but before hashing, i.e., measuring
• Unlock program memory before releasing it to run
The architecture illustrated

Processor
Boot Loader

Memory

Program

a520fb3319f
What could go wrong?

An attacker could:
- Replace the boot loader
- Exploit a vulnerability in the boot loader
- Over-write the PCR value
- Find 2\textsuperscript{nd} program with same digest
- Replace the program

Architectural mitigations:
- Boot loader in hardware or ROM
- Make the boot loader so simple and small that vulnerabilities become unlikely
- Make the register read-only by software
- Use a wide-enough cryptographic hash
- Lock memory before hashing; the computed digest will uniquely identify whatever program got itself loaded
Discussion

Load time (before execution) is the right time to measure the software image, because
• Data structures in program memory have not yet changed
• The loader patches all unresolved run-time references before allowing the program to execute, so the program’s memory no longer the same after patching
• Self-modifying code is necessary to provide some exotic functions

Most programs also have launch parameters, aka, inputs to the program when it starts, and these should be measured too
• Compute the PCR value as hash(program || launch-parameters) instead of hash(program)

Locking program memory before measurement crucial
• Otherwise other software running on another core or controller could change program blocks in parallel with the hash computation

Unlocking program memory after measurement crucial
• Otherwise program cannot update its own variables
A deeper dive

Real systems are hierarchical

- The hardware boot loader loads a bootstrap loader, which offers more functionality than that provided by the hardware boot loader
- The bootstrap loader loads a loader for some particular operating system
- The operating system loader loads the operating system

How do we create measurements for real systems?

Two solutions have been proposed:

- Certificate chains
- Hash chains
A **digital certificate** is a shared public data structure asserting something about the world

- The most common type of certificate, called an **identity certificate**, asserts that a public encryption or signature verification key is possessed by some named entity

Every certificate has

- A **subject** field, which is the entity to which the assertion applies
- A predicate asserting some property or attribute about the subject
  - The subject possesses the private key corresponding to a public key named in the certificate
- An **issuer**, who makes an assertion about the subject by creating the certificate
  - When the subject is the issuer, then the certificate is called **self-signed**
- The issuer’s public signature verification key or its hash
- The issuer’s signature over the other fields in the certificate
Review: signature schemes

Recall a **signature scheme** consists of a set of three algorithms:

- A **key generation** algorithm: \((sk, vk) \leftarrow \text{KeyGen}(\text{security-level})\)
- A **signing** algorithm: \(\sigma \leftarrow \text{Sign}(sk, \text{message } m)\)
- A **verify** algorithm: \(\text{Verify}(vk, m, \sigma) = \text{true if } \text{Sign}(sk, m) \text{ produced } \sigma \text{ else false otherwise}\)

A signature scheme is **secure** if it is computationally infeasible to forge a signature without the secret signing key.

We assume efficient and secure signature schemes exist
- e.g., RSA signatures, or Schnorr signatures over a properly chosen elliptic curve
Measurement based on certificate chains

The chain of programs loaded will be $P_1, P_2, ...$, with $P_1 = \text{boot-loader}$

The boot loader stores the digest of itself in the hardware PCR, but each of the other programs store its hash in memory

The boot loader manufacturer publishes a public signature verification key $pvk_{\text{boot-loader}}$ with each boot loader, and installs the corresponding secret signing key $ssk_{\text{boot-loader}}$ in the boot loader

Before letting the program $P_{i+1}$ execute, program $P_i$

- Generates a public/secret verification/signing key pair $(pvk_i, ssk_i)$,
- Generates a certificate $Cert_i$ with $\text{subject} = \text{hash}(P_{i+1} \parallel \text{Launch-parameters}_{i+1})$ and signed by $ssk_i$ (i.e., verified with public signature verification key $pvk_i$),
- Securely delete $ssk_i$, and
- When $i = 1$, $P_1$ (i.e., the boot loader) also generates a certificate $Cert_0$ with $\text{subject} = \text{hash}(P_1 \parallel \text{Launch-parameters}_1) \parallel pvk_1$ and signed by $ssk_{\text{boot-loader}}$ (i.e., verified by the public verification key $pvk_{\text{boot-loader}}$) (recursion base case)

The measurements of all the loaded program are contained in the chain $Cert_0, Cert_1, Cert_2, ...$
Homework Problem

What happens to the security of measurement based on certificate chains if process $P_i$ fails to delete its secret signing key $ssk_i$ before letting process $P_{i+1}$ start to execute?

Hint: What happens if $P_i$ becomes infected by malware?
Measurement based on hash chains

When the hardware boots, it begins by setting the PCR to a well-defined value, usually 0.

For each process $P_i$ in the chain booted it extends the PCR by computing

$$new\text{-}PCR\text{-}value \leftarrow \text{hash}(old\text{-}PCR\text{-}value \ || \ P_i \ || \ launch\text{-}parameters_i)$$

As long as the hash function is 2\textsuperscript{nd}-preimage resistant, an attacker cannot change the value recorded of the launched software $P_0 = \text{boot}\text{-}loader, P_1, P_2, \ldots$ without detection.
## Comparison

<table>
<thead>
<tr>
<th>Certificate chain measurement</th>
<th>Hash chain measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can build arbitrary process trees, i.e. program $P_i$ can launch multiple programs $P_{i+1}$</td>
<td>Can only build process chain</td>
</tr>
<tr>
<td>Certificate chain explicitly documents all launched programs in the order launched</td>
<td>PCR value implicitly documents the launched programs and order launched</td>
</tr>
<tr>
<td>Certificate chain consumes memory for each certificate</td>
<td>The PCR contains the entire measurement</td>
</tr>
<tr>
<td>The TCB must include a hash function, signature scheme, cryptographic random number generator, and certificate scheme</td>
<td>The TCB must include a hash function</td>
</tr>
</tbody>
</table>
Digression: $2^{\text{nd}}$-preimage resistance is not enough

The above discussion only uses $2^{\text{nd}}$-preimage resistance, not collision resistance.

In point of fact the measurement hash function must be collision resistant, not just $2^{\text{nd}}$ pre-image resistant:

- Let the hash function digest be $n$-bits wide
- The developer creates a real program template $R$ and an attack program template $A$, both with containing an empty data area of at least $n/2$ bits
- For $i = 0$ to $2^{n/2} - 1$ create programs $R_i$ and $A_i$ from $R$ and $A$ by choosing an $n$-bit random number $r_i$ and encoding this into the empty data area
- If hash functions act like random mappings, by the birthday problem for two lists there exists integers $0 \leq j, k < 2^{n/2}$ such that $\text{hash}(R_j) = \text{hash}(A_k)$ with probability at least half

Thus the hash function needs to be collision resistance, to defend against evil developer attacks.
Measurement problem summary

Our measurement architecture appears to deliver a high assurance solution *provided*

- The TCB hardware is designed and built to provide this functionality (no intentional backdoors),
- The TCB hardware is exploit-free (no unintentional backdoors), and
- No one tampers with the TCB hardware

The measurement only tells what was loaded, not what software is running now

Research problem

- Since no one knows how to build tamper-proof hardware, we implicitly assume a human being guards her laptop or smart phone hardware from tampering
  - A bad assumption, as the Evil Maid attack demonstrates
- How do we make the “no tampering” assumption true in many of our motivating examples, where continuous supervision by a human is impractical?
  - For example, why do we believe an auto mechanic will never replace a vehicle’s TCB with rogue hardware?
The attestation problem

Now that we have a way to securely measure what programs get loaded, we need a way to securely report the measurement

• Even if the loaded software was subsequently compromised by malware after it began running!!

What we need is a way to read the PCR and know this is what we have read

Idea: why not have the TCB **hardware sign** the response to requests to read the PCR?

• And we need a way for the TCB to tell us what programs it has measured into the PCR

Then the resulting **signature** becomes the TCB hardware’s way of **attesting** to the software it loaded

• **Attest**: a verb meaning to provide or serve as clear evidence of; to be a witness; to formally certify
A **manifest** is a special type of certificate (chain) in which

- The subject is the digest of the program to which the manifest is loaded
- The certificate is signed by the program manufacturer

The manifest signature is the manufacturer’s assertion that the manifest subject identifies the software it intended to release

- But note a manifest does not attest that the manufacturer thinks the program is bug free, has no back doors, etc.

Manifests are widely used by all modern operating systems

Manifests are often embedded in the program executable itself using an operating system specific file format

- There are well-defined ways of inserting manifest into both ELF (Linux, Android, IOS) and COFF (Windows) executable file formats
Attestation: Retrieving the PCR value

Random challenge

a520fb3319f, manifest, challenge, signature

Processor

Prover

a520fb3319f

Memory

Program

Manifest

Program
What could go wrong?

An attacker could:

- Exploit a vulnerability in the prover
- Steal the signing key
- Forge the prover’s signature
- Publish an alternative prover signature verification key for a signing key held by malware
- Shuffle the manifest(s) returned
- Modify one or more returned manifest

Architectural mitigations:

- Boot loader in hardware or ROM
- Signing key never leaves hardware
- Manufacturer issues a processor-unique digital certificate for each processor
- Denial of service: the verifier will compute the PCR value ordering the programs differently, resulting in a different PCR value
- Denial of service: either a certificate signature will fail or the PCR value the verifier computes from the manifest(s) is different than the PCR value
Signature scheme in TCB hardware

It is essential that the Prover call a TCB primitive, called a **quoting service**, to produce the attestation response, or **quote**

- The prover software passes the challenge and (the hash of the) relevant manifests to the TCB quoting service
- The prover passes a buffer in which to build the quote
- The quoting service builds a response message by copying the parameters and the PCR value into the buffer and adding some other formatting (e.g., a message header)
- The quoting service locks the buffer, signs its contents, unlocks the buffer, and appends its signature and a manufacturer-provided certificate certifying the key as belonging to this hardware
  - The certificate proves that the signature was produced by the hardware, not by malware-corrupted software
- The quoting service releases the buffer back to the prover
Attestation requires that the verifier use a cryptographic random number generator

- The challenge must be unpredictable to avoid having a compromised prover call the TCB signer prior to receiving the challenge

The TCB’s signing key is in hardware, so it is always possible for an attacker with access physical to the hardware to steal the key

- If the key is stored in external non-volatile memory, the key can be read with a hardware probe monitoring the bus between the TCB processing unit and the external memory
- If the key is stored in fuses, the key value can be read with a sufficiently powerful scanning electron microscope
- If the signature algorithm implementation is not carefully implemented, the key value can be read by monitoring the EMI or power fluctuations from the TCB processing unit
- Individual key bits can be read by glitching the power to the TCB processing unit during the signing operation
Details, details

Since the PCR value depends on the launch parameters, the launch parameters must be returned to the verifier

- It is customary to hash the manifests and launch parameters, and deliver this digest value with the verifier’s challenge to the quoting service
- Then the quoting service signs $\text{PCR-value} \parallel \text{manifests-and-launch-parameters-hash} \parallel \text{verifier-challenge}$ to produce the quote
- The prover constructs the response message by adding the manifests and launch parameters to the quote

When the prover is a component of the top-level program, it is common to include a public signature verification key in the hash of manifests and launch parameters, to secure a communications channel between the program and the verifier

- If the TCB hardware has not been tampered, then this ties the public signature verification key to the TCB hardware constructing the quote
Surprise!!

This attestation scheme only works remote
• An attacker who has compromised the local operating system can often predict future outputs from the system random number generator and so get the hardware to precompute the appropriate attestation
• An attacker can relay attestation requests to a different system and return the responses, including the certificate certifying that the attestation was produced by the hardware

Using the trick of including a public signature verification key in the quote that is subsequently used to create a secure channel assures us that the prover’s software environment was loaded by some TCB, but not necessarily the intended machine
• This is often good enough, provided we have reason to believe that the TCB hardware has not been compromised
Attestation problem summary

We can deliver high assurance measurements of what software was loaded on a computing device provided

• The TCB hardware is designed and built to provide this functionality (no intentional backdoors),
• The TCB hardware is exploit-free (no unintentional backdoors),
• No one tampers with the TCB hardware
• The manufacturer issues a certificate attesting that the TCB hardware’s public signature verification key indeed belongs to the hardware
• The software is signed by someone who can attest that this is indeed the intended software

The attestation only tells what was loaded, not

• What software is running now, or
• That the software has no intentional backdoors (it provides only the documented functionality), or
• That it has no unintentional backdoors (exploitable vulnerabilities, i.e., unintentional undocumented functionality)
The policy problem

So far we know how to

• Measure what software and configuration parameters were loaded
• Securely retrieve these measurements remotely

What we don’t know how to do yet is assure that the intended software and configuration parameters are loaded onto the local machine

We need some sort of authorization mechanism, which can distinguish between authorized and unauthorized software and/or configuration parameters and enforce that only authorized can be used

The authorized set of software and configuration parameters is called the computing device’s policy
Three boot models

There are three basic models for loading programs

• **Unmeasured boot** – a program is loaded and then executed without measurement. If the program has a manifest, it is ignored. This is how almost all software is booted until very recently.

• **Measured boot** – a program is loaded, measured into a PCR, and then allowed to execute. A program may or may not have a manifest. If it has a manifest, a remote verifier can use attestation to learn what was loaded.

• **Secure boot** – a program is loaded and measured into a PCR, but not allowed to execute unless it is on a **white list** of authorized programs. Secure boot always requires a manifest, to specify the digest of the program loaded

The choice of unmeasured, measured, or secure boot is a **policy** choice that has to be set somehow, to reflect the preferences of the device **owner**, and stored in some kind of protected storage:

• DIP switches and straps

• Sealed storage
DIP switches and straps

A manual physical switch to force some setting of the hardware on the motherboard
  • Imune to software attacks
  • Setting can be changed by anyone with physical access to the computing device

This is practical only for setting the boot loader settings
  • At most only a few bits of information, such as the choice among unmeasured, measured, or secure boot

Also not sufficient: how does one program know what which programs it launches are authorized?
Sealed storage

The TCB hardware usually can **seal** a program’s parameters

- Both launch parameters and parameters generated by the program itself, e.g., public/private key pairs

The TCB derives a symmetric key for based on the value of the PCR when the program begins execution

\[
\text{symmetric-key} \leftarrow \text{kdf(tcb-secret, PCR-value)}
\]

The TCB uses the key with a symmetric key authenticated encryption scheme to encrypt the program’s launch parameters
Review: Symmetric key authenticated encryption

A **symmetric key authentic authenticated encryption scheme** is a set of three associated algorithms

- **Encrypt** – Encapsulates a plaintext into a ciphertext under some key (or trapdoor) $K$
- **Decrypt** – Decapsulates a ciphertext into a plaintext under some (the same) $K$ such that
  \[
  \text{Decrypt}(\text{ciphertext}, K) = \text{plaintext}
  \]
  if and only if $\text{ciphertext} = \text{Encrypt}(\text{plaintext}, K)$, i.e., ciphertext is not a forgery
- **KeyGen** – describes how the key $K$ must be chosen

A symmetric key authenticated encryption scheme is called secure if

- It is computationally infeasible for an attacker to learn anything about plaintext, except its length from ciphertext, and
- It is computationally infeasible for an attacker to produce a valid ciphertext without the key $K$

We assume secure symmetric key authenticated encryptions schemes exist, provided that secure block ciphers exist
Sealing and unsealing

\[ \text{WrappingKey} \leftarrow kdf(tcb-secret, a520fb3319f) \]
What could go wrong?

The attacker could:

Steal the sealer’s key *tcb-secret*

Steal the app’s sealed parameters

Modify or delete the app’s sealed parameters

Move the app’s sealed parameters to another processor

Change the app’s program

Potential mitigation:

*tcb-secret* never leaves the hardware and device specific

So what? The sealed parameters encrypted

The decryption will fail – better make a backup of the sealed parameters

If every processor has its own sealer key, then the sealed parameters are *bound* to this processor

Since the app’s hash is used to compute the sealed parameters, no other program can unseal its parameters – the parameters *bound* to this program
Let’s fix a chicken-and-egg problem

Since the sealed data can include launch parameters, and the launch parameters are needed to put the PCR into the state needed to derive the symmetric key to encrypt and decrypt

The problem is the measurement algorithm design did not anticipate this need

• Originally we had $PCR\text{-value} = hash(program)$
• This did not take the program’s launch parameters into account, so we modified this to $PCR\text{-value} = hash(program \| program\text{-launch-parameters})$

A better design extends the $PCR\text{-value}$ with the launch parameters

• First measure the program: $PCR\text{-value} \leftarrow hash(PCR\text{-value} \| program)$
• Next compute the sealing key: $sealing\text{-key} \leftarrow kdf(tcb\text{-secret, } PCR\text{-value})$
• Decrypt the program’s sealed data: $plaintext \leftarrow Decrypt(sealed\text{-data, } sealing\text{-key})$ or on fail erase program from memory
• Finally measure the launch parameters: $PCR\text{-value} \leftarrow hash(PCR\text{-value} \| plaintext\text{.launch-parameters})$
Term project

Suppose a program is securely launched and we want to upgrade it to a new version
   • For example, to fix bugs, or to introduce new functionality

Suppose the program also uses sealed storage to save parameters
   • The old version of the program builds its sealing key as $K \leftarrow kdf(sealer-key, old-program-digest)$
   • The new version of the program builds its sealing key as $K' \leftarrow kdf(sealer-key, new-program-digest)$
   • Thus $K \neq K'$, so the new program can’t unseal the parameters encrypted under the old program’s key $K$

Design a mechanism to move the wrapped process from the old version of the program to the new version, or an old version of the sealed parameters with a new
   • Make sure your design handles replay, i.e., can detect when the attacker replaces correctly encrypted new parameters by correctly encrypted old parameters (hint: you design needs to somehow save state that detect replays)
The secure boot problem summary

We can deliver high assurance that the intended software was loaded on a computing device provided

• The TCB hardware is designed and built to provide this functionality (no intentional backdoors),
• The TCB hardware is exploit-free (no unintentional backdoors),
• No one tampers with the TCB hardware
• We use a secure authenticated encryption scheme to store the parameters of each program we launch
The ownership problem

So we can specify what programs and parameters are authorized by sealing this information, which is consumed by secure boot.

Who is authorized to set these parameters?

This problem is solved by setting a root authorization key, which is called the owner key:
• All changes to the list of programs or program parameters must be authorized by the owner key
• That is, the owner must authenticate using the owner key before the TCB will allow any changes to the system configuration

There are two models of ownership:
• Taking ownership
• Ownership transfer
Taking ownership metaphor

This is modeled on the notion of **imprinting** from the biological of ethology.
Taking ownership

When the computing device is powered on, the normal boot sequence can be interrupted to do administration of the low level hardware.

The device operator can specify an owner password, which becomes the root authorization key.

Once set, the device operator must present the owner password to make any changes to the boot sequence programs and their launch parameters.

Whoever sets the owner password first becomes the device’s owner.

It is critical that setting the owner password require physical access:

• Many users never set the owner password for their computing devices.
• If remote access were possible, eventually hackers would figure out how to set the owner password of a remote device over the Internet.
Ownership transfer

In this model the device is always owned, and the root authorization key is changed when control of the device passes from one party to another

- This model uses a public/secret signature verification/signing key pair as the root authorization key: all updates to boot programs and parameters must be verified by the root authorization key
- The manufacturer installs the original public verification key when building the device and thereby becomes the first owner
- A protocol executes when ownership is transferred from one owner to another by requiring the old owner to endorse (certify) the public signature verification key of the new owner

This model requires a global consensus history to record ownership transfers

- For instance, something like the Bitcoin block chain
- This is needed because a device that is reset to factory defaults will forget its ownership history
Comparison and summary

Only the Take Ownership model has been implemented, but as currently implemented requires a keyboard and display

- To enter owner password and give feedback – unsuitable for embedded devices

We also don’t know how to defeat the duplicate rogue device problem in the Take Ownership model

Ownership transfer mitigates these problems, but requires much more machinery

- Digital signatures, a global history like block chains
Example 1: TPM

TPM = Trusted Platform Module

A TPM is a security co-processor that provides
- Secure storage
- Secure or authenticated boot
- Attestation

Delivered either as a stand-alone co-process soldered onto motherboard or integrated into the Platform Controller Hub (PCH, aka the “chipset”)

Ownership based on the take-ownership model
- When unowned a user can enter a password as the owner key
- All future sensitive operations

Installed on hundreds of millions of PCs
TPM block diagram
Example 2: Intel SGX

SGX = Software Guard Extensions

SGX is an Intel processor feature beginning in SkyLake

• Secure program launch into a region of encrypted, authenticated, replay protected memory called an enclave
  • All code and data reside in the enclave’s encrypted memory
  • Receives commands/outputs results to the rest of the system through a well-defined API
• Keys and other secrets can be sealed to a particular program running in an enclave
• SGX can attest to which program was loaded into an enclave, along with all of its launch parameters

Ownership can be asserted with a public verification key as one of the program launch parameters

• All commands to the enclaved program must be signed by the corresponding private key
SGX block diagram
Example 3: ARM TrustZone

TrustZone a feature of ARM processors
• Processor splits the address space into trusted and untrusted zones
• Trusted applications and their data run in the trusted zone
• Normal applications run in the untrusted zone
TrustZone block diagram
Summary

There are broad needs to verify whether the intended software was actually loaded.

The idea behind trusted computing is:
- Sign software executables
- Use hardware to verify the signatures on the software at load time
- Use hardware to measure and then create quotes attesting to which software it loaded
- Boot images and parameters are access controlled using a root authorization key

This model can in principle defeat persistent malware and many other software-only attacks:
- But it offers very limited defense against hardware attacks

Trusted computing implementations have become ubiquitous in our computing devices:
- TPM
- Intel SGX
- ARM TrustZone (used by Android)
Feedback?