CHAPTER 2

Fundamental Principles of Intel® TXT

The first step to more secure computing is improved hardware. So before we discuss how to use the technology, let's define what constitutes an Intel® TXT–capable platform and the underlying principles behind the technology. We will take a look at the unique components, how they work together, and what they do to produce a more secure environment.

What You Need: Definition of an Intel® TXT–Capable System

Intel TXT relies on a set of enhanced hardware, software, and firmware components designed to protect sensitive information from software-based attacks. These components are illustrated in Figure 2-1 and Figure 2-2.

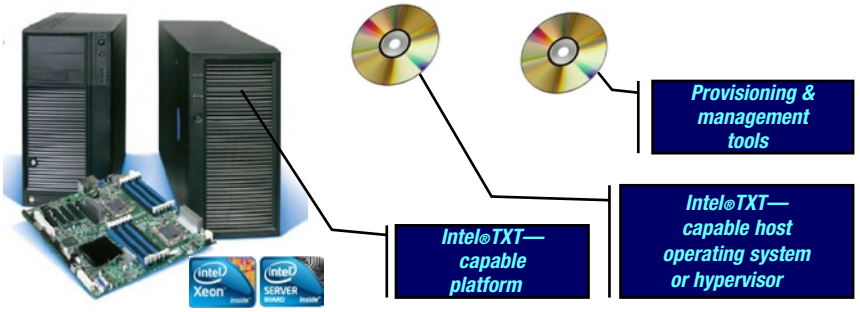

Figure 2-1. *Intel® TXT–capable system*

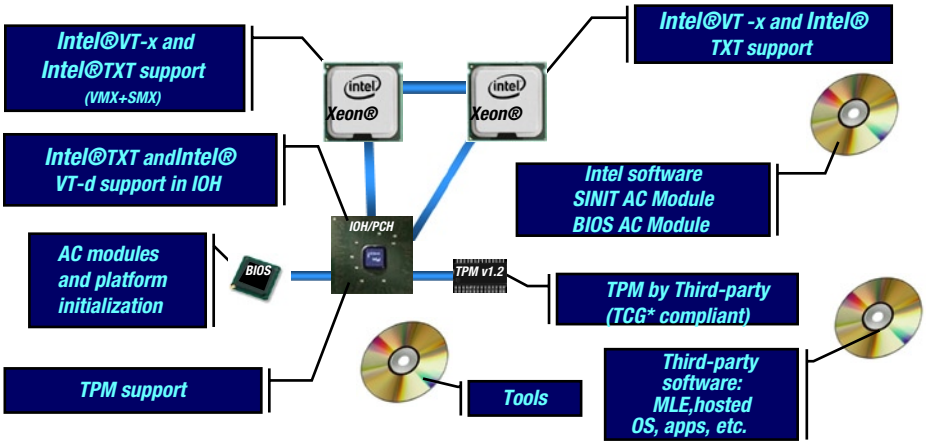

Figure 2-2. *Intel TXT platform components*

As illustrated in Figure 2-1, it takes more than just an Intel TXT–capable platform. An Intel TXT–capable platform provides the foundation; however, to get the full value from this technology one needs to install an Intel TXT–capable operating system. (Intel TXT–capable platform systems will be discussed shortly.) In addition, provisioning tools and other management software may be needed to tune Intel TXT operation to meet datacenter policy, as well as enable service clients to benefit from it. For now, let's focus on the platform and answer the question: What makes Intel TXT different and why?

Intel® TXT–Capable Platform

The platform components are illustrated in Figure 2-2, but don't worry, it is the platform manufacturer who is responsible for integrating and testing these components. This means that by the time a platform reaches the datacenter, it is Intel TXT–capable (or it is not). Intel TXT–ready server platforms did not start shipping until early 2010, so older platforms are not Intel TXT–capable. As of 2012, most major manufacturers offer Intel TXT–capable server platforms. It should be pointed out that some manufacturers have designed server platforms that can be upgraded to being Intel TXT–capable via a BIOS upgrade and/or a plug-in module. So check with your platform vendor to find out if existing platforms are Intel TXT–capable.

As we review the various components of Intel TXT, these topics may seem disjointed or incomplete, but as the chapter progresses, it will become apparent just how they all work together.

Intel TXT Platform Components

The recipe for a platform that is able to support Intel TXT includes the following ingredients specifically designed for Intel TXT.

Processor

All Intel IA32 server processors, beginning with the Intel® Xeon® 5600 series, (codenamed Westmere[1]) support Intel TXT. This support includes a new security instruction (GETSEC, part of the Safer Mode Extensions known as SMX). This new instruction performs multiple security functions (referred to as *leaves*). Two of the leaf functions

[1]Released in the first quarter of 2010.

(GETSEC[ENTERACCS] and GETSEC[SENTER]) provide the ability to securely validate *authenticated code modules* (ACMs) and (if valid) invoke that authenticated code using a very secure area within the processor that is protected from external influence (see the "Authenticated Code Modules" section). All processors that support Intel TXT also support Intel® Virtualization Technology (i.e., VMX instructions and an enhanced architecture for increased isolation and protection). There is no limit on the number of processors, thus Intel TXT is found on UP, DP, and MP platforms. Actually, the number of processor cores is limited to a 32-bit value (4,294,967,295 maximum value), but for all practical purposes, this is not a limiting factor.

Chipset

Intel chipsets designed to work with Intel Xeon 5600 series and later processors include:

- special TXT registers
- an enhanced architecture
- controlled access to the Trusted Platform Module (TPM)

It is the chipset that enforces TPM locality (defined later) and enforces different access rights to the TXT registers, depending on the security level of the entity attempting to access the TXT registers. Other chipset manufactures can also build chipsets that implement Intel TXT (they would provide their own ACMs). Unless otherwise noted, we will be discussing Intel chipsets and ACMs.

Trusted Platform Module (TPM)

Intel TXT makes extensive use of the *Trusted Platform Module* (TPM), a component defined by the Trusted Computing Group[2] (TCG). The TPM provides a number of security features (more on this in the section titled "The Role of the Trusted Platform Module (TPM)." In particular, the TPM provides a secure repository for measurements and the mechanisms to make use of those measurements. A system makes use of the measurements for both reporting and evaluating the current platform configuration and for providing long-term protection of sensitive information.

BIOS

The platform BIOS is specially designed to configure the platform for secure mode operation. BIOS plays a significant role, whose duties include:

- Supplying the ACMs. ACMs are created by the chipset vendor and provided to the BIOS developer to be included as part of the BIOS image.
- Configuring the platform for Intel TXT operation.
- Performing a security check and registering with the ACM.
- Locking the platform's configuration using the processor's GETSEC security instruction.

Once locked, BIOS and other firmware can no longer modify the configuration. BIOS performs this locking before executing any third-party code such as option ROMS on add-in cards.

[2]The Trusted Computing Group (TCG) is a not-for-profit organization dedicated to advancing computer security whose goals are to develop, define, and promote open industry standards for trusted computing building blocks and software interfaces (www.TrustedComputingGroup.org).

Authenticated Code Module (ACM)

An *authenticated code module* is a piece of code provided by the chipset manufacturer. This module is signed by the manufacturer and executes at one of the highest security levels within a special secure memory internal to the processor. Most Intel-based platforms use Intel chipsets, thus the ACMs for those platforms are created by and signed by Intel.

Each ACM is designed to work with a specific chipset, and thus is matched to the chipset. ACMs are invoked using the processor GETSEC instruction. There are two types of ACMs: BIOS ACM and SINIT ACM. The BIOS ACM measures the BIOS and performs a number of BIOS-based security functions. The SINIT ACM is used to perform a secure launch of the system software (operating system). SINIT is an acronym for Secure Initialization because it initializes the platform so the OS can enter the secure mode of operation.

The BIOS image contains both the BIOS ACM and the SINIT ACM; however, the OS may choose to supply a more recent version of the SINIT ACM when it does a secure launch.

As we will see later, the ACM is the core root of trusted measurements, and as such, it operates at the highest security level and must be protected against all forms of attack. Because of this, the following extra precautions are taken to verify and protect the ACM.

- The processor's microcode performs a number of security checks on the ACM before allowing the ACM to execute:
  - ACM must match the chipset
  - ACM version must not have been revoked
  - ACM must be signed
  - Signature must be valid
- Before the processor makes these checks, it copies the ACM into protected memory inside the processor, so that the ACM code cannot be modified after it is validated.
- A checksum of the public signing key is hardcoded in the chipset and used to verify that the ACM was signed with the proper signing key. Each chipset generation uses a different signing key. Thus, only someone who has the private key can create an ACM. The private keys are very well guarded and only a few very trusted Intel people have access to the private keys.
- The signature uses cryptography to validate that the ACM has not been altered.

Thus the GETSEC instruction will only execute the ACM if the ACM was signed with the correct key, matches the chipset, and has not been modified.

The Role of the Trusted Platform Module (TPM)

It is hard to talk about Intel TXT without referring to the TPM. The TPM is an important component specified by the Trusted Computing Group (www.trustedcomputinggroup.org) expressly for improving security. We really applaud the TCG for defining such an ingenious device. The TPM specification is in three volumes (four, if you count the compliance volume) and there are books just on the TPM. Luckily for us, we don't have to go into great detail. However, an understanding of the key features will provide a better appreciation of how Intel TXT provides secure measurements.

Currently, platforms incorporate TPMs adhering to version 1.2 of the TCG TPM specification. Version 2.0, which is expected to be released late 2013. It provides the same functionality plus includes enhanced features, capabilities, and capacities.

The primary security functions provided by the TPM are illustrated in Figure 2-3.

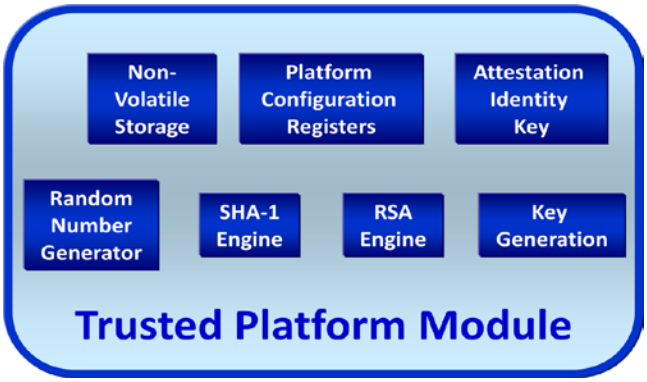

Figure 2-3. *Trusted Platform Module*

TPM Interface

The TPM interface is simple, making it easier to defend. The TPM connects to the chipset by the Low Pin Count (LPC) bus[3] and is only accessed via memory mapped I/O (MMIO) registers protected by the chipset. This means that I/O devices cannot access the TPM at all and the OS can easily control which processes have access to the TPM. In addition, the TPM enforces additional protections and access rights.

Localities

The TPM has different privileged levels called *localities*. Each locality is accessed by a different 4K page allowing both hardware and software to control access to each locality. The chipset controls which localities are active (that is, whether a read or write to that page will be ignored) and system software can control which processes have access to those pages via normal memory management and paging functions. There are five localities (0 through 4) and Intel TXT uses the localities as shown in Table 2-1.

Table 2-1. *TPM Locality*

Locality	Access
0	Always open for general use.
1	Operating system (not used as part of TXT).
2	System software (OS) in secure mode. Only after the OS has successfully performed a secure launch. The OS may close this access at any time.
3	ACMs. Only an ACM that has been invoked by the GETSEC instruction has access to locality 3.
4	Hardware. Only the processor executing its microcode has Locality 4 access.

Note that these localities are not hierarchical. Thus, locality 4 is not a super-user and does not automatically have access to keys, data, and resources available to other localities. Later, we will look at the use of localities for protection of keys and data.

[3]Future TPMs may use the Serial Peripheral Interface (SPI) bus, but the principles are the same.

Control Protocol

The interface to the TPM is message based. That is, an entity creates a request message and sends it to the TPM (via a locality page) and then reads the response message (via the locality page). The protocol allows the entity (using messages) to set up a secure session. Sessions allow the user to encrypt the message content, but more importantly, the means to authenticate authorization.

Messages that require authorization contain one or more authorization fields, called *HMAC keys*. HMAC stands for Hash Method of Authentication. The HMAC key is calculated as the hash of the message, a nonce (number used once), and the authorization value. The nonce and authorization values are not part of the message, and thus must be known by both parties.

The HMAC in a request demonstrates that the caller knows the authorization value and is therefore authorized to issue the TPM command. In the response, the HMAC value authenticates that the response came from the TPM and has not been modified during transport.

The protocol is as follows. Each message in a session uses a different set of random numbers (nonces) to generate the HMAC key(s). The requester provides the odd nonce that the TPM uses to calculate the HMAC value in its reply and the reply contains the even nonce that the requester uses to calculate the HMAC value in its next request. This prevents *replay attacks* and *man-in-the-middle attacks* because the HMAC value changes for any change in the message. Even the exact same command will have a different HMAC value each time it is issued (because the nonce is different).

Let's look at an example.

1. The caller sets up a session by sending a start session command to the TPM and the TPM responds with a session handle and the TPM's nonce.
2. The caller creates a command that requires authorization (that is, requires that the caller prove that the caller knows the password associated with the object of the command). To do this, the caller creates a cryptographic hash of the command, plus the nonce provided by the TPM, plus the password.
3. The caller sends the command, the caller's nonce, and the hash digest to the TPM. Note that the password is not sent.
4. The TPM then performs the same hash except using the real password of the object. If that hash digest matches the digest in the command message, then the TPM concludes that the caller knows the password, and that the command has not been altered in transport.
5. In the response, the TPM provides a new TPM-nonce that the caller uses in the next command. The response also contains the hash digest of the response parameters, plus the odd nonce from the request message, plus the password.

Note that the caller could have also set up a session so that the command parameters and response parameters were encrypted.

The bottom line is that the TPM is a very hardened device that protects its resources and requires specific access authorization that is protected from tampering.

Random Number Generator (RNG)

Random numbers are used in safeguarding communication between a requesting entity and the TPM, especially when that entity is remote to the platform. Unlike a pseudo RNG that software applications use, the TPM's random number generator is a true RNG and can provide entropy for other purposes. Thus, in addition to using random numbers to protect TPM transport, an entity can request a random number from the TPM for other purposes, such as encryption and securing external channels.

SHA-1 Engine

The TPM hashing algorithm is SHA-1 (pronounced *Shah-one*), which is a cryptographic hash function designed by the United States National Security Agency as a US Federal Information Processing Standard. SHA stands for *secure hash algorithm*. This hashing produces a 20-byte (160-bit) digest. The hash is used for various authorization and authentication processes. The TPM's SHA-1 engine is used internally to do the following:

- Authenticate request messages
- Authorize response messages
- Calculate the hash digest of code and data
- Extend PCR values
- Authenticate/authorize keys

For more information on SHA-1, see the "Cryptographic Hash Functions" section.

RSA Engine and Key Generation

The TPM uses the RSA asymmetric algorithm for digital signatures and for encryption. RSA stands for Ron Rivest, Adi Shamir, and Leonard Adleman, who published the algorithm for public-key cryptography in 1978. The key generator uses that algorithm to generate asymmetric keys (a public key and a private key). The private key is kept internal[4] to the TPM (thus only known to the TPM). The public key) is used by an entity to encrypt data that can only be decrypted by someone with the private key (the TPM). The RSA engine uses the keys to encrypt and decode messages and data.

The "Cryptography" section provides details on how keys are generated and used.

Platform Configuration Registers (PCRs)

Pay attention. There will be a quiz later. Just joking, but PCRs are one of the two TPM features that are accessed as objects and the basis of attestation, as well as used in launch control policy. Thus, they play a key role.

The TPM provides special protected registers to store measurements. These registers are known as Platform Configuration Registers (PCRs). Each PCR has an inherent property of a cryptographic hash. That is, each PCR holds a 20-byte hash digest. Entities never write directly to a PCR, rather they "extend" a PCR's content. For the extend operation, the TPM takes the current value of the PCR, appends the value to be extended, performs SHA-1 hash on the combined value, and the resulting hash digest becomes the new PCR value. Therefore, each PCR holds a composite hash of all of the elements that have been measured to it. This means that PCRs cannot be spoofed—the only way for a PCR to have a specific value is if the exact same measurements are extended to it in the exact same order. Any variation results in a PCR value that is very different. This is a key concept that provides secure generation and protection of measurements.

Intel TXT uses TPMs that have 24 PCRs. Some PCRs are static, which means that their values are only reset when the system first powers on by the power-up-reset signal. Thus, measurements stored in those PCRs persist until the platform powers down and are used to hold the static chain of trust measurements. Other PCRs are dynamic, which means their value can be reset to their default values, but only via certain localities (as specified by the TCG). These are used for the dynamic chain of trust measurements. The PCRs that hold the dynamic chain of trust measurements can only be reset by the ACM.

[4]Because the TPM has limited internal storage, the key might be encrypted and stored external to the TPM as a blob. But the value of the key is not known externally.

Nonvolatile Storage

TPMs provide nonvolatile storage, that is, memory that is persistent across power cycles and has more security features than system memory. This nonvolatile random access memory (NVRAM) is the other TPM feature that is exposed externally as objects and plays an important role in creating and protecting the launch control policy set by the datacenter manager.

Compared to system memory, TPM NVRAM is very small, on the order of 2000 bytes, so it is a precious resource. NVRAM is allocated in small chunks and the entity that allocates a chunk of NVRAM specifies access rights for that data. To give you an idea of how flexible and powerful TPM NVRAM is, let's look at what needs to be specified to allocate TPM NVRAM:

- An index (an ID that identifies that chunk).
- The size of that chunk.
- Read access rights.
  - Specifies which localities are allowed to read the data.
  - Specifies if any PCRs must contain specific values in order to read the data. Data cannot be read if any of the selected PCRs do not contain their expected value.
- Write access rights.
  - Specifies which localities are allowed to write the data.
  - Specifies if any PCRs must contain specific values in order to write the data. Data cannot be written if any of the selected PCRs do not contain their expected value. The set of selected PCRs can be different than those selected for read access. Also, the expected PCR values can be different.
- Attributes/permissions. Can specify any of the following:

  Read

  - Temporal read lock. The value can be read until locked by a read of zero length; unlocked at the next power cycle.
  - What authorization is required to read; may specify:
    - None.
    - A specific authorization value (like a password) is required.
    - If physical presence assertion is required. Physical assertion typically means BIOS is performing the access.

  Write

  - Persistent Lock. The value can be locked for write access (by a write of zero length) and once locked can never be altered.
  - Temporal Write Lock. The value can be locked for write access (by a write of zero length), and once locked, cannot be altered until the next power cycle.
  - Write Once. The value can only be written once per power cycle/ unlocked at the next power cycle.
  - Whether partial writes are allowed.

  - What authorization is required to write; may specify:
    - None (when PCRs or locality restrict write access).
    - A specific authorization value (like a password) is required.
    - If physical presence assertion is required.

As you can see, NVRAM capabilities are very flexible. To give an example, ACMs makes use of these capabilities to allocate a chunk of NVRAM that they use to save state and pass data between ACMs. The protection comes from specifying that only localities 3 and 4 can write to that index (since only the ACM and microcode can access the TPM using locality 3 or 4). We will see later that this allows the ACM to store the BIOS measurement and detect if BIOS code has changed from one boot to the next. Other indexes are used for setting launch control policy and they specify different attributes and access rights. The bottom line is that even though hardware and system software control who can access the TPM, it is the entity that allocates the chunk of NVRAM that determines under what conditions the data can be read and written, as well establishes if a password is required, and then specifies the password.

Attestation Identity Key (AIK)

The *Attestation Identity Key* (AIK) is a concept that allows a TPM to certify that keys and data originated from the TPM without revealing any platform-specific identity information. An entity (like a certification authority) can create an AIK. This causes the TPM to create an asymmetric key pair that it uses to sign keys created by the TPM or sign data reported by the TPM (such as PCR values). The root of the certificate can be traced back to the TPM vendor and verified that it is, in fact, a compliant TPM.

TPM Ownership and Access Enforcement

One last point to cover is the TPM enforcement. A TPM arrives at the platform manufacturer in an unlocked state to make it easy for the manufacturer to create NVRAM indexes. At this stage, the TPM does not enforce any access controls. The manufacturer provisions the TPM creating the indexes required by Intel TXT and then performs an OEM lock. The OEM lock protects those indexes so that their access policy cannot be changed. It also prevents anyone from creating an NVRAM index that falls in range of indexes reserved for the platform manufacturer.

When the platform arrives at the customer's datacenter, the TPM is disabled and inactivated to prevent rogue software from taking ownership and mounting a "Denial of Service" attack. Taking ownership is a process where an entity reads the TPMs Endorsement Key (EK) and uses it to encrypt a TPM Owner Authorization value (think of this authorization as the TPM owner's password) and then sends the encrypted password to the TPM in a "Take Ownership" command.

The TPM Owner Authorization value is typically a SHA-1 hash of a password or pass phrase and it is used to prove to the TPM that certain commands (such as allocating NVRAM) are authorized by the platform owner.

Prior to establishing TPM ownership, the TPM does not accept commands that allocate TPM resources, like allocating TPM NVRAM and creating keys. After establishing TPM ownership, those commands require owner authorization (that is, knowledge of the TPM owner's password).

Cryptography

Since not everyone is familiar with encryption standards, let's review some fundamental concepts.

- Encryption is the process of using an algorithm (called a cipher) to transform plaintext information to make it unreadable to anyone except those possessing a special key.
- Decryption is the reverse process, to make encrypted information readable again (that is, make it unencrypted by changing ciphertext back into plaintext).

- Encryption is used to protect data and for authentication (such as for a digital signature).
- There are two parts to encryption. The first is key creation and the second is the encryption/decryption process.
- There are two classes of encryption algorithms (symmetric and asymmetric).

Symmetric Encryption

Symmetric encryption is where the same key[5] used to encrypt the data is also used to decrypt it. Symmetric encryption is generally faster than asymmetric encryption. Its primary use is where the same entity is performing both the encryption and decryption of the data, such as for protecting data saved to disk. This is sometimes referred to as *one-party encryption* because the entity that encrypts the data can also decrypt the data.

Asymmetric Encryption

Asymmetric encryption is where a different key is used to decrypt the data than the key used to encrypt it. These two keys are referred to as the public key and the private key. It is not possible to derive the private key from the public key. The following are some use models:

- *Signing*. The private key is the encryption key. The signer uses it to encrypt a hash of the module being signed (known as the module's signature). The pubic decryption key is appended to the module being signed along with the encrypted signature. Other parties can verify that the module comes from the signer by verifying that the public key was issued by the signer—usually through a *certification authority* (CA). The module's integrity is then verified by calculating the hash of the module and comparing it to the modules signature decrypted with the public key.
- *Endorsement*. The private key is the decryption key. The public key is provided to another party that uses it to encrypt sensitive data (such as a password) that can only be decrypted by the first entity.
- *Secure transmission*. Uses two pairs of keys where the private keys are the decryption keys. Each end of a secure channel creates their own key pair and provides the other end of the channel with the public key, which is used to encrypt transmissions.

Private keys must be very guarded and are never passed in the clear. Public keys do not require protection because the knowledge of the public key does not reveal any secrets nor does it provide access to any privileged information.

Intel TXT supports PKCS[6] #1.5, a public standard for encryption that allows key sizes of 1024, 2048, or 3072 bits and recommends a minimum key size of 2048.

Cryptographic Hash Functions

A cryptographic hash is the result of performing a specific hash algorithm on a block of data. The purpose of the hash function (often called *hashing*) is to map a large/variable size block of data to a smaller fixed-length data set, referred to as a hash, hash digest, or digest. The SHA-1 algorithm maps any amount of data to a 20-byte hash digest while the SHA-256 algorithm maps any amount of data to a 32-byte hash digest.

[5]Sometimes one key is derived from the other, but the concept is still the same.
[6]PKCS stands for Public Key Cryptography Standards, which was published by RSA Data Security Inc., now a subsidiary of EMC Corporation.

Cryptographic hash functions are used for many purposes. Intel TXT uses it to detect changes in code, data, messages, configuration, or anything else that may be stored in memory—because any accidental or intentional change to what is being measured or verified will drastically change the hash value. The following are properties of a cryptographic hash:

- The digest is easy to compute.
- It is not reasonably possible to create a block of data that produces a given hash. We say "not reasonably possible" because given enough time, anything is possible. A 20-byte value has 2^{160} values. So do the math to figure out how long it would take to just generate 2^{160} values. Even using a million computers, it would take billions of billions of years.
- It is not reasonably possible to modify a block of data without changing its hash digest.
- It is not reasonably probable to find two different blocks of data with the same hash digest.

Therefore, it is reasonable to claim that a hash of an entity is a unique identifier of that entity and only that entity will have that hash value. There are many applications for cryptographic hash functions:

- *Digital signature*. The signer encrypts the hash of the module being signed. Other parties can verify that the module comes from the signer by verifying that the public key was issued by the signer, and then verify the module's integrity by calculating the hash of the module and comparing it to the modules signature decrypted with the public key.
- *Digital fingerprint*. A hash of a block of data differentiates that data from any other data. One use is when that block of data is code. Thus the hash of the code becomes a unique identifier. The same applies to a data file, configuration settings, and so on. We have already discussed how Intel TXT measures code, configuration, and other elements to TPM PCR registers.
- *Message authentication.* When both the sender and receiver have a secret authentication value (typically the hash of a password or passphrase). The sender of a message creates a hash of the message concatenated with the secret authentication value to produce an authentication key, and appends that key to the message. The recipient verifies that the sender knows the correct secret authentication value by performing the same hash calculation (using his copy of the secret authentication value) and comparing the result to the authentication key passed with the message. The secret authentication value is not sent in the message. If the sender used the wrong value or the message was modified in transit, the keys will not match, and thus the recipient rejects the message. Therefore, this is both authentication and authorization because the sender authorizes the message by using the secret authentication value and the recipient authenticates that the message came from an authorized source and was not modified in transit. This is the Hash Method of Authentication (HMAC) that we discussed as part of the TPM control protocol.
- *Checksums*. A hash digest of a block of data can easily be used to validate that the data has not been altered. An example is how Intel TXT verifies the launch control policy. Since the size of TPM NVRAM is limited, some of the policy is stored in flash or on disk. The hash measurement of the portion of the policy stored on disk is securely stored in the TPM NVRAM. Any change to the external policy data results in a mismatch of its hash measurement to the hash value stored in the TPM.

Why It Works and What It Does

Now that we have all of the pieces, let's put the puzzle together and discuss what Intel TXT does that is different from previous platform designs.

Key Concepts

Here are some of the key concepts of Intel TXT.

- Intel TXT provides for the following:
  - Secure measurement
  - Dynamic launch mechanisms via special instructions
  - Configuration locking
  - Sealing secrets
- Intel TXT helps detect and/or prevent software attacks such as:
  - Attempts to insert nontrusted VMM (rootkit hypervisor)
  - Reset attacks designed to compromise secrets in memory
  - BIOS and firmware update attacks

Measurements

A measurement is a cryptographic hash of code, data, or anything that can be loaded into memory (such as policies, configuration, keys, and passwords). The default hashing algorithm is SHA-1, which produces a 20-byte (160-bit) digest with the following properties:

- A single bit change in the entity being measured causes a majority of the bits in the hash digest to change. Thus any change to the entity being measured can be easily detected.
- The hash algorithm is not reciprocal—that is, the hash process cannot be reversed. Thus, it is impossible to reconstruct the measured entity from its hash digest. This is important when using the hash to authenticate passwords (that is, a password hash used to verify a password cannot be used to construct a valid password).
- The likelihood of any two entities (that are not identical) having the same measurement are astronomical (2^{160} is an enormously large number).

Intel TXT also supports SHA-256 for some measurements, but SHA-1 is the default for all measurements. And it is the only hashing algorithm supported by the TPM 1.2 family devices. Earlier, it was noted that there was a new specification for the TPM 2.0 family of devices on the horizon. These new devices will be able to support additional hashing and encryption algorithms.

Secure Measurements

Measuring code and configuration is not meaningful unless the measurements can be guaranteed to be authentic, accurate, and protected.

So what constitutes a "secure" measurement? The answer is twofold:

- The measurement is hardware rooted (hardware root of trust).
- The measurements are protected from tampering.

To understand *root of trust*, we need to define *chain of trust*. The chain of trust starts with a trusted component that measures the next component. That component then measures the next component, and so on until all components that need to be measured have been. The root of trust is the component that started the measurement chain.

For TXT, the root of trust is the processor. When the server boots, and before any BIOS code is permitted to execute, a special microcode built into the processor performs a validity check on the BIOS ACM and then measures the BIOS ACM before it is executed. The BIOS ACM then measures a portion of the BIOS referred to as *startup BIOS*. The startup BIOS then measures other portions of the BIOS code. This is important because the entity that starts the measurement chain and makes the first measurement establishes the accuracy of that measurement chain.

At this point, it should be obvious that the final value is not necessarily a single operation, but results from multiple measurements that need to be processed together to form a single result. This process is referred to as *extending*. It is the TPM that provides the protection from tampering. The TPM contains a set of Platform Configuration Registers (PCRs). These PCRs cannot be written but rather are extended. The phrase *extending a PCR* means hashing the existing PCR content with the new measurement to create the new PCR content. Thus, extending is the means to concatenate multiple measurements into a single result. It is the TPM itself that performs the extending operation, thus software only provides the measurement to be extended and identifies which PCR is to be extended. But not all software has access to the TPM and certain PCRs can only be extended from specific localities.

Since PCRs are extended (not written), even if malicious software was able to extend a PCR, the only impact is that the PCR would carry an invalid measurement—remember that one of the SHA-1 hash properties is that it is not possible to create a block of data that produces a given hash. Thus, it is not possible to extend a PCR to get a given result, except by measuring the exact same objects in the exact same order. Since one of the Intel TXT principles is not to execute code until it has been measured, we can conclude that the measurement already includes the malicious code—and thus, by definition, making the PCR value different from the expected "known good" value. Thus, malicious software is not able to extend the register back to the known good value.

In addition to extending measurement to a PCR, the platform maintains a log of the objects extended to that PCR. This allows a verifying entity to use the log to validate the PCR content. Another way to look at this is that the log that contains the information and the PCR value validates the log. We will see later that for some applications, it is the final value of the PCR that is of interest and for other applications, it is the log.

Static and Dynamic Measurements

Until now, we have looked at measurements generically, but there are multiple PCRs and thus multiple measurement results. Static measurements are those that are made once each time the platforms boots. There are 16 PCRs reserved for static measurements (PCR0–15). These PCRs are only cleared when the platform powers up or there is a hard reset: that is, a signal to all components (TPM, processors, and chipset) to return them to their power-on state. Static PCRs are used in measuring the platform configuration, as follows:

- PCR0: CRTM, BIOS, and host platform extensions
- PCR1: Host platform configuration
- PCR2: Option ROM code
- PCR3: Option ROM configuration and data
- PCR4: IPL code (usually the MBR)

- PCR5: IPL code configuration and data (for use by the IPL code)
- PCR6: State transition and wake events
- PCR7: Host platform manufacturer control
- PCR8–PCR15: Reserved for the OS; not used by TXT

PCR0 is the primary CRTM PCR and speaks to the integrity of PCR1–PCR7. CRTM stands for Core Root of Trust Measurement (see the "The Intel TXT Boot Sequence" section), which means it is measured first and starts the chain of measurements for the static PCRs. There is also a Dynamic Root of Trust Measurement (DRTM) for the dynamic PCRs.

Dynamic measurements are measurements made to PCRs that can be reset without resetting the platform. There are eight dynamic PCRs, used as follows:

- PCR 16: Reserved for debug; not used by Intel TXT.
- PCR17–PCR20: Reset upon a successful secure launch.
  - PCR17: DRTM and launch control policy
  - PCR18: Trusted OS startup code
  - PCR19: Trusted OS (for example OS configuration)
  - PCR20: Trusted OS (for example OS kernel and other code)
- PCR20–PCR22: Can be reset by the trusted OS while in secure mode. Note that PCR20 is reset at secure launch and by the trusted OS.
  - PCR21: Defined by the trusted OS
  - PCR22: Defined by the trusted OS
- PCR23: Reserved for applications; can be reset and extended by any locality. Outside the scope of Intel TXT.

To summarize, Intel TXT provides the means to securely measure and report various platform components, and thus detect changes in those components. Measured components include:

- Platform Configuration (Static Root of Trust)
  - PCR0: BIOS code
  - PCR1: BIOS settings
  - PCR2: Option ROM code
  - PCR3: Option ROM settings
  - PCR4: Boot Sector - Master Boot Record (MBR)
  - PCR5: Boot configuration
  - PCR6: Platform state changes
- Operating System (Dynamic Root of Trust)
  - Operating system code (PCR18+ OS-specific PCRs)
  - Operating system settings (PCRs are OS-specific)
  - Other components specified by the operating system

These components are divided into two groups because the root-of-trust for those groups is different.

The Intel TXT Boot Sequence

By now, it should be easy to piece together what happens when the platform boots. The boot sequence that generates the static chain of trust measurements goes like this:

1. Processor microcode
    a. Loads BIOS ACM into secure memory internal to the processor to protect it from any outside influence.
    b. Validates that the ACM is authentic using cryptographic signature.
    c. Starts the core root of trust measurement by extending the BIOS ACM measurement to PCR0.
    d. Only if all checks pass, the microcode will
        - Enable ACM mode by enabling access to TPM locality 3 and opening access to TXT private registers.
        - Execute the BIOS ACM startup code.
        - The use of internal secure memory and then validation assures the ACM is free of malicious code before and during execution.
2. BIOS ACM startup code
    a. Measures critical BIOS components.
    b. Extends PCR0 with those measurements.
    c. If the ACM determines that secrets might have been left in memory (i.e., a potential reset attack), it validates that BIOS startup code has not been modified or corrupted. (More about reset attacks to follow.)
    d. Exits ACM mode disabling locality 3 and access to TXT private registers.
    e. Jumps to the BIOS startup code.
3. BIOS
    a. Measures the remainder of the BIOS code and extends PCR0 with those measurements. Does not execute any code until that code has been measured and extended into a PCR.
    b. Configures the platform.
    c. Measures the BIOS configuration and extends that into PCR1.
    d. Calls BIOS ACM using the GETSEC instruction to perform security checks.
        - Processor microcode loads the ACM into secure internal memory and performs the same validation as before. It then enables ACM mode and executes the requested ACM function.
        - ACM performs its security checks, exits ACM mode, and returns to the BIOS.
    e. Finishes platform configuration.

f. Calls the BIOS ACM using the GETSEC instruction to lock the BIOS configuration.

- Processor microcode loads ACM into secure internal memory and performs the same validation as before. It then enables ACM mode and executes the requested ACM function.
- ACM performs configuration locking, exits ACM mode, and returns to the BIOS.

After locking the configuration, the BIOS may execute other firmware.

- Measures any option ROM code (for I/O devices) and extends those measurements into PCR2.
- Executes the option ROM code for each device.

4. Each option ROM
   a. Measures any hidden option ROM code and extends that measurement into PCR2.
   b. Configures and enables the device.
   c. Measures device configuration information and extends that measurement into PCR3.
   d. Returns to BIOS.
5. BIOS
   a. Selects a boot device.
   b. Measures the IPL (Initial Program Load) code, typically the Master Boot Record (MBR), and extends that measurement into PCR4.
   c. Executes the IPL code.
6. IPL code
   a. Measures IPL configuration and extends that measurement into PCR5.
   b. Loads the OS boot loader and executes it.

This sequence is illustrated in Figure 2-4.

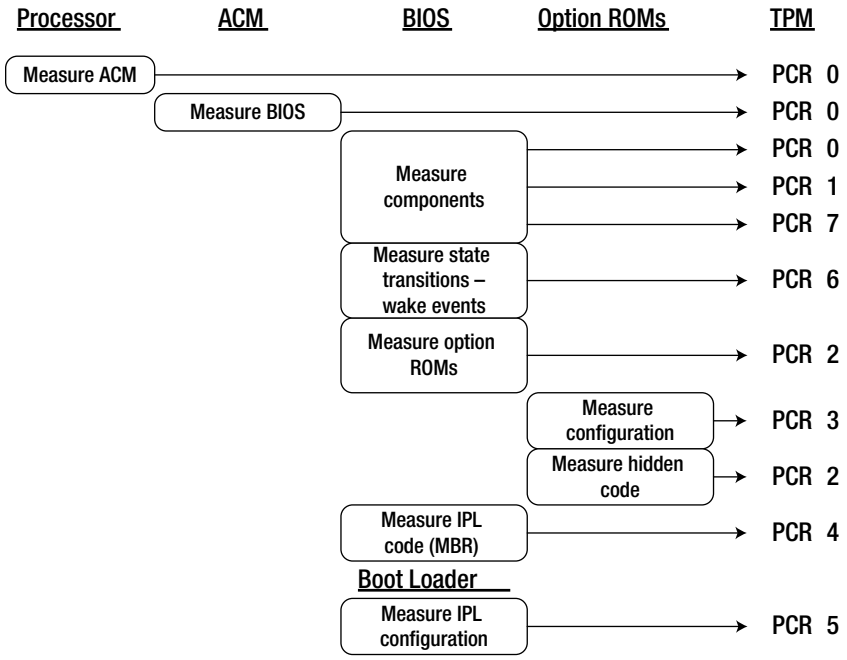

Figure 2-4. *Typical static measurement sequence*

At this point, the BIOS has brought the platform to a state where it is ready to boot the operating system (IPL). Normal bootstrap process is followed where the OS loader gets executed. Typically, what happens for an Intel TXT–enabled OS, instead of loading the first kernel module, the first module loaded is a special Trusted Boot (TBOOT) module. The purpose of the TBOOT module is to initialize the platform for secure mode operation and initiate the Measured Launch Process.

Measured Launch Process (Secure Launch)

This is the part of the technology that controls how and if a host OS can enter the secure mode of execution, and thus becomes a trusted OS. The following are the characteristics of secure mode:

- The trusted OS code has been measured and that measurement is a known good value (as established by the platform owner).
- The trusted OS starts in a deterministic state.
  - All processors except the bootstrap processor are in a special idle state until the OS explicitly starts them up directly in protected mode. This is different from the normal bootstrap process where those processors are started in real mode and have to be transitioned into protected mode. Thus, this removes that window of vulnerability.
  - System Code is protected against access by I/O devices (referred to as Direct Memory Access or DMA).

- The OS has TPM locality 2 access and the dynamic PCRs have been reset to their default values. DRTM PCRs have been extended.
  - PCR17 has been extended with the measurement of the SINIT ACM (that is, the Dynamic Root of Trust Measurement, or DRTM) and the policy that allowed the secure launch.
  - PCR18 has been extended with the measurement of the trusted OS.

What the OS has to do to enter secure mode and start the DRTM is drastically different from what BIOS does for the static root of trust measurements. However, the principles are the same.

Before the host OS enters secure mode, it must put the platform in a known state. The host OS then invokes the GETSEC instruction specifying the SENTER operation. This causes the processor microcode to do the following:

- Load SINIT ACM into secure memory, internal to the processor, in order to protect it from any outside influence.
- Verify that the ACM matches the chipset and that the hash of its public key matches the hash hardcoded into the chipset.
- Validate that the ACM is authentic by verifying its measurement matches its cryptographic signature.

  Again, the use of internal secure memory and then validation assures the ACM is and remains free of malicious code before and during execution.
- Start the dynamic root of trust measurement by clearing PCR17–PCR20 and then extending the SINIT ACM measurement to PCR17.
- Only if all checks pass, the microcode will
  - Enable ACM mode, which enables access to TPM locality 3 and opens access to TXT private registers.
  - Execute the SINIT ACM code.

The SINIT ACM now takes control and does the following:

1. Checks to see if it has been revoked—that is, if a newer version of the SINIT ACM has executed and set the revocation level (stored in the TPM) to indicate this SINIT ACM is an older version that is no longer to be used.
2. Validates that the platform was properly configured by BIOS and that the OS has properly conditioned the platform for a secure launch.
3. Measures the trusted OS code.
4. Executes the Launch Control Policy engine that interprets the launch control policy set by the datacenter to determine if the OS is to be trusted to do a secure launch.
5. If any of these checks fail, it is considered an attack and the ACM issues what is called a *TXT reset*, which prevents a secure launch until the platform has been power-cycled. This assures that no malicious code remains in memory.
6. If these checks pass, then the ACM
   a. Extends PCR17 with the policy it used to allow the launch
   b. Extends PCR18 with the trusted OS measurement
   c. Enables locality 2 access
   d. Exits ACM Mode and jumps to the trusted OS entry point

The trusted OS then takes control and is now able to extend PCR17–PCR22. The trusted OS is in control of what gets extended, and when. For example, the OS could measure additional code into PCR19, trusted OS configuration into PCR20, and extend geographic location into PCR22.

Protection Against Reset Attacks

We briefly discussed how the BIOS had to be trusted to protect against reset attacks. Here is how that protection unfolds.

When the OS successfully performs a secure launch, it sets a special nonvolatile *Secrets* flag to indicate that it has (or will have) secrets and other privileged information in memory. When the trusted OS gracefully shuts down and exits the secure environment, it resets the *Secrets* flag. The flag can only be reset via the trusted OS invoking the GETSEC instruction. This is usually done as part of a graceful shutdown.

If a *reset attack* occurs (that is, a reset before the trusted OS clears the *Secrets* flag), the BIOS ACM (which is the first to execute after the reset) assures that the BIOS scrubs the memory. For desktop and mobile platforms with simpler memory architectures, the BIOS ACM actually clears the memory. However, high-end workstations and servers have more complex memory architectures, so the BIOS must be trusted to scrub the memory. But the ACM does not blindly trust the BIOS, because it could have been corrupted and thus contain malicious code. The ACM uses two methods to verify that the BIOS had not been corrupted: autopromotion and signed BIOS policy. Each OEM selects the method that works best for their platform.

Autopromotion means that the ACM remembers the BIOS measurement and if the platform passed the launch control policy, then that BIOS will be trusted to scrub memory in the event of a reset attack.

Signed BIOS policy allows the OEM to provide a list of known good BIOS measurements signed by the OEM. The signature is validated by a hash measurement of the public key stored in the TPM. Thus, if the BIOS measurement does not match a value in the signed list, or the list signature is not valid, or the list is signed with the wrong key, then BIOS is not considered trusted.

If the ACM is not able to verify that the BIOS is trusted using one of these methods, then it locks the memory controllers. Since this is done before BIOS executes its first instruction, the platforms boots without any memory. We call this a "bricked" system, because a platform without memory is about as useful as a brick. BIOS may be able to recover from a bricked system, if it can restore the BIOS or revert to a BIOS version that will be trusted. Otherwise, the only way to recover is to turn off power and remove the coin battery. This signals the ACM that a person, not malicious software, is in control of the platform.

One point that needs to be mentioned is the potential of a BIOS update to look like a reset attack. First, this is only an issue when the BIOS update is done while the trusted OS is in secure mode (OS present BIOS update) and only if the platform resets before the OS exits the secure environment (that is, doesn't reset the *Secrets* flag). For this case, autopromotion will fail. There are several ways for the OEM to mitigate this problem. One is to use signed BIOS policy. Others are to defer the BIOS image update until the next boot or provide a failsafe BIOS image (*failsafe* means revert to the original BIOS image whose measurement is known to the ACM). There are more, so it is not fair to say signed BIOS policy is better, but platforms incorporating signed BIOS policy can recover from corrupted BIOS without the need to remove the coin battery.

Launch Control Policy

Intel TXT includes a policy engine that permits the datacenter to specify what is a known good platform configuration and which system software is permitted to do a secure launch. The term for this feature is *launch control policy* (LCP). In Chapter 4, we will go into detail on how to create a launch control policy and provide guidance on selecting a policy. For now, let's focus on the mechanics.

There are actually two policy engines: one in the BIOS ACM that controls the policy used to validate the BIOS integrity when there is a reset attack and one in the SINIT ACM that enforces the launch control policy. We have already discussed the BIOS policy engine in the "Protection Against Reset Attacks" section.

There are two authorities for launch control policy: the *platform supplier* (the OEM) and the *platform owner* (the datacenter). The platform supplier policy is sometimes referred to as the *default policy* because it is created by the OEM before the platform ships. Typically, the platform supplier's default policy will allow any host OS to perform a secure launch on any platform configuration. That is not very exciting, so let's talk about the platform owner policy, what it does, and how it overrides the default policy.

There are two elements to a launch control policy:

- The specification of valid platform configurations (called the PCONF element).
- The specification of OS versions that are allowed to perform a secure launch (called the Measured Launch Environment, or MLE element).

The LCP engine analyzes these two elements independently of each other, but both must be satisfied. That is, any OS in the MLE element may perform a secure launch on any platform by the PCONF element. In addition, the result of omitting the PCONF element is that all platform configurations are allowed. Likewise, the result of omitting the MLE element means that any system software is allowed to perform a secure launch on a platform with a valid platform configuration.

Platform Configuration (PCONF)

So how does one specify the platform configuration? Since BIOS went through all the trouble to measure everything into PCRs, the PCONF policy element uses those measurements. Only the first seven PCRs convey platform configuration information.

- PCR0: BIOS code
- PCR1: BIOS settings
- PCR2: Option ROM code
- PCR3: Option ROM settings
- PCR4: Boot sector - Master Boot Record (MBR)
- PCR5: Boot configuration
- PCR6: Platform state changes

The policy authority (for example, the datacenter manager) selects which PCRs are relevant and creates a composite hash from those PCR values. The Intel TXT architecture actually allows selection of any of the PCRs. The data structure that specifies which PCRs were selected and the composite hash is called a *PCRInfo*. The PCONF element can hold any number of PCRInfo structures (the actual limit is greater than 100 million). Typically, only a few PCRInfos are needed per platform.

For each PCRInfo, the policy engine will create a composite hash from the actual values of the specified PCRs and compare that to the composite hash in the policy. If they match, it means all of the selected PCRs have the correct value, and thus the platform configuration is trusted. If none of the PCRInfos produce a match, then the platform is not trusted and there can be no secure launch.

Trusted OS Measurements (MLE Element)

By now you are probably thinking that the MLE does the same thing but with the dynamic PCRs (PCR17–PCR22); however, that's not correct. PCR0–PCR7 values are stable after IPL, but dynamic PCRs are at their default values. Thus, the MLE element simply contains a list of known good TBOOT measurements that are compared to the TBOOT hash value measured by the ACM. That hash value will only be extended into PCR18 if LCP passes. In other words, the policy engine will calculate the hash of the trusted OS and compare that measurement to the list of measurements in the MLE element. If it matches any one of them, then the OS is trusted to perform a secure launch.

Protecting Policies

So how are the policies protected against tampering? The answer is: using the TPM NVRAM. But a policy can consume a large number of bytes—more than the TPM NVRAM provides. Thus, the policy is split in part into TPM policy data stored in the TPM NVRAM (referred to as *NV policy data*) and in an optional *policy data structure* (stored in a file) that holds the PCONF and MLE elements (that is, it contains the lists of allowed platform and MLE measurements). The policy data structure is optional, because a policy that allows any platform configuration and any OS does not need PCONF and MLE elements.

The platform owner creates a specific TPM NVRAM index (40000001) called the *PO policy* index with a property that only allows the platform owner to write that data (which is the small NV policy data mentioned earlier). One of the fields of the NV policy data is a hash value, which is the hash measurement of the optional policy data structure.

The policy engine calculates the hash of the policy data structure and compares it to the hash value stored in the TPM. If the policy data structure had been compromised, the hash values won't compare, and thus the policy will fail.

Sealing

Earlier, it was stated that measurements are used for many purposes. One of those purposes is the sealing of secrets. Data can be sealed to one or more PCR values. Sealing is the act of having the TPM encrypt a block of data so that it cannot be decrypted unless the specified PCRs have the correct values. The caller specifies the PCRs and their composite hash when it passes the TPM the block of data. The TPM returns an encrypted "blob."

Later, the software can ask the TPM to decrypt the blob. The TPM will only decrypt it if the current PCR values match what was specified at the time the data was sealed.

Let's look an example to see how sealing could work to protect sensitive data. Upon installing an OS, that OS creates a symmetric encryption key, which it uses to encrypt data that it stores on disk. The OS seals that key, specifying its own PCR18 value, and then saves the blob to disk. Each time the OS boots up, it has the TPM unseal the blob, which provides the OS with the key it uses to encrypt and decrypt data going to and from disk. Because the data was sealed to PCR18, only that OS is able to unseal the blob—and thus only that OS is able to decrypt the data.

Any other system software will have a different PCR18 measurement, and thus could not unseal the blob. Even if an attacker is able to reverse assemble the OS code, it is not able to gain knowledge of the disk encryption key. Since each platform has a unique disk encryption key, knowledge of a key used on one platform is of no value on other platforms.

Attestation

We have seen how the PCR measurements can be used to set launch control policy, determine who can access TPM NVRAM, and used to seal data. Now let's discuss the value of those measurements to applications both internal and external to the platform.

Because of the uniqueness property of cryptographic hashes, PCR values and their logs can be used to identify exactly what version of software is executing, as well as its environment.

Several key concepts come into play:

- The TPM protects the measurements.
- The combination of TXT hardware and ACMs guarantee the accuracy of the static root of trust measurement and the dynamic root of trust measurement.
- The combination of authentic root of trust measurement and chain of trust principle (requiring any code that is able to extend a PCR to be measured prior to its execution) enforces the ability to detect tampering.
- The TPM provides a "TPM Quote" where the TPM signs the PCR values to assure that values are not maliciously or inadvertently modified in transit. This guarantees authenticity of the measurements.

This accuracy, security, and dependability mean applications can use these measurements to make their own trust decisions. Unfortunately, other properties of the cryptographic hash make it difficult to use the values directly. This is where an attestation authority comes into play. An *attestation authority* (similar to a certification authority) is a trusted third-party entity that is able to authenticate the PCR values and translate those values by comparing them with a database of known good values.

There are several models for attestation. One is where the application retrieves the PCR values and uses an attestation service to authenticate that the target platform is, in fact, what it claims to be. Another model is where the attestation authority retrieves the PCRs from the target platform and provides the translated results to its client applications.

Either way, the client application is able to determine or confirm (among other things) the type of platform, the OS vendor, and exact version of system software that is executing on the target platform.

Another method (known as *whitelisting*) is where the attestation authority maintains a list of known good results and simply validates if the target platform meets one of the trusted profiles.

Later, we will discuss how this information is used to enforce policies set by cloud service clients.

Summary

Intel Trusted Execution Technology is compliant with the TCG PC Client and Servers Specifications. Cryptographic secure hashing algorithms generate unique measurements of platform components, platform configuration, system software, and software configuration. Those measurements are stored in PCRs protected by the TPM. Each PCR may contain multiple measurements and the PCR log identifies each element that was measured to that PCR. The value in the PCR can be used to verify the log.

PCR0 contains the static root of trust measurement, as well as BIOS code measurements. These measurements cannot be spoofed because they are rooted in hardware. The validity of PCR1–PCR7 rests on the validity of PCR0. That is, PCR1–PCR7 can only be trusted if the PCR0 measurement is known to be good.

PCR17 contains the dynamic root of trust measurement, as well as policy measurements. These measurements cannot be spoofed because they are rooted in hardware. PCR18 contains the measurement of the trusted OS. The validity of PCR18 rests on the validity of PCR17. PCR19–PCR22 usage is OS-specific—their purpose is defined by the trusted OS. Therefore, their values can only be trusted and interpreted if PCR18 is a known good value.

The platform owner controls what is considered a valid platform configuration, and thus which OS is allowed to perform a secure launch. An OS in secure mode has demonstrated compliance with the datacenter's launch control policy, and thus has TPM locality 2 access. The secure launch (more accurately referred to as a *measured launch*) means that the OS's measurements have been extended into the dynamic PCRs, and therefore the OS is able to seal data to its own unique measurement.

PCRs and their logs can attest to the type, health, and state of a platform to both local and remote applications. Those applications are able to use that information to make their own policy decisions.