CHAPTER 22

Platform Security Technologies That Use TPM 2.0

Okay, we've written a whole book on TPMs, and you've apparently read the whole thing. Perhaps our attempts to keep the book interesting were successful. . .or you're extraordinarily persistent. . .or maybe you just cheated and skipped to the conclusion.

Either way, we've reached the end of the matter. TPMs are great and awesome, the security equivalent of sliced bread, no doubt about it. And TPMs by themselves offer a good level of security. For instance, an application like Microsoft's BitLocker can use a TPM to securely store a hard disk encryption key and control access to the key.

But there are also platform-level technologies that combine TPMs with other platform- and vendor-specific security features to produce even stronger solutions. The goal of this chapter is to describe three of those technologies and how they integrate with TPMs.

The Three Technologies

Three major platform technologies use TPMs. This chapter describes these three technologies at a high level, how they make use of TPM 2.0 devices, and how they empower applications to use TPMs. This chapter aims to be non-partisan and, for that reason, steers clear of comparisons of these three technologies and avoids marketing-oriented statements.[1] This is a TPM 2.0 book, so the focus is on how TPMs are used in each of these environments. In the interests of maintaining neutrality and accuracy, the sections on the technologies were written by experienced current and former representatives of the companies mentioned.

[1]It should be noted that Intel sponsored the publishing of this book, including the publishing costs. Intel seeks to advance the adoption of TPM 2.0 devices for the betterment of the computing security ecosystem.

Some Terms

Before we go any further, we need to define some terms:

- *Trusted computing base (TCB)*: Everything in a computer system that provides a secure environment. Basically, it's the set of hardware and software components that must trusted in order to provide security to the system.
- *Measured boot*: A boot method where each component is measured by its predecessor before being executed. Typically these measurements are accumulated in PCRs via extend operations.
- *Chain of trust*: A chain of operations that comprise a measured boot.
- *Root of trust for measurement (RTM)*: The base component of a chain of trust that is implicitly trusted. As such, it must be small and immutable (in ROM or protected by hardware).
- *Static root of trust (SRTM)*: The base component of the chain of trust that starts at power-on and extends to sometime before the OS boots. In the server version of Intel TXT, the SRTM is the CPU microcode. In other architectures, the SRTM is a ROM image.
- *Dynamic root of trust (DRTM)*: The chain of trust that starts after the OS has booted in non-secure mode. This allows the dynamic establishment of a measured boot environment. In Intel TXT, the CPU microcode is also the DRTM. DRTM is sometimes called *delayed launch*.
- *Authenticated code module (ACM)*: ACMs are Intel TXT digitally signed code modules that are invoked by the special Intel `TXT GETSEC` instruction. ACMs are the next components to execute after the SRTM and DRTM components execute. Which ACM is invoked and which sub-functionality is invoked is determined by a register setting when the `GETSEC` instruction is executed.
- *Unified extensible firmware interface (UEFI)*: A standardized version of BIOS that is CPU independent and standardizes boot and runtime services.
- *SEC phase*: The security phase of the UEFI BIOS. This is the first code to execute after reset.
- *PEI phase*: The pre-EFI phase of UEFI BIOS. This is the next phase after the SEC phase. The SEC and PEI phases together comprise what used to be called the BIOS *boot block*.

Intel® Trusted Execution Technology (Intel® TXT)

Intel TXT has been shipping since 2002 in client machines and since 2010 in servers. Intel TXT provides a chain of trust that is rooted in the microprocessor's hardware and is extended in stages to the OS and even to applications, depending on how higher levels of software make use of it.

This section describes Intel TXT at a high level first, including its features that offer advantages over a TPM-only solution, and then delves into the details of how it uses TPM 2.0's capabilities. At a high level, the advantages of Intel TXT over a TPM-only solution are a hardware-based root of trust, a smaller TCB, and specific checks of the hardware and software configuration performed by the ACMs. This section highlights how these advantages are implemented.

Other Intel technologies use TPMs, including Intel Boot Guard. This chapter doesn't describe these technologies or how they use TPM 2.0 devices, because Intel TXT is currently the most prevalent technology and a representative example of how TPM 2.0 devices are used. Also note that there are two flavors of Intel TXT: one for client platforms and one for server platforms. Many of the principles of operation are shared, but we focus on the server version, because it uses a superset of TPM functionality.

High-Level Description

Intel TXT for servers can defend against BIOS attacks, reset attacks, rootkits, and software attacks and allows the system integrator and user many options for configuring the level of protection. Although it does prevent or mitigate some attacks, its primary purpose is to notify the user and system software of the presence of a possible attack and prevent a verified launch if an attack is detected. Intel TXT hardware and software and the TPM are tightly integrated in a way that protects both the TPM and the TXT registers from unauthorized access. Critical measurements stored in the TPM cannot be spoofed, and the TPM protects OEM and user policies from unauthorized alteration.

How does it do this? A short description is that a chain of trust is extended from the Intel processor and/or chipset hardware through the BIOS. Then, after the OS has booted, if the user desires to enter secure mode at the OS level, a measured launch sequence is initiated by the OS or a software program running on top of the OS (DRTM). This measured launch ensures that there are no security holes in the system before launching the OS and entering secure mode. Basically, a chain of trust may be extended from the hardware all the way up to the highest levels of software, enabling a system administrator or user to create and use security policies. This chain of trust always measures components before actually executing them.

Intel TXT Platform Components

There are many components to Intel TXT:

- *CPU and chipset hardware*: The chipset contains special Intel TXT registers, many of which are readable and/or writeable only by Authenticated Code Modules and CPU microcode.
- *CPU microcode*: This is hardwired firmware inside the microprocessor for executing groups of micro-operations that are combined to perform assembly language instructions as well as other internal CPU functions.
- *Intel Authenticated Code Modules (ACMs)*: These ACMs can only be created by Intel and are digitally signed with a private key that is only known to Intel. The public key is hardwired into hardware registers in the chipset, and only a module signed with the matching private key is allowed to execute. ACMs are invoked by Intel microcode, and they function as extensions of microcode. For server Intel TXT, there are two ACMs, the BIOS ACM and the SINIT (measured launch initialization) ACM:
    - The BIOS ACM contains several sub-functions (calls), two of which are:
        - The *Startup ACM*[2] call is called by CPU microcode at power-on to start the SRTM. It typically measures the BIOS boot block, or, as it's called in UEFI, the SEC and PEI phases of BIOS.
        - The *Lock Config* call is made by the BIOS just before it exits the part of the BIOS measured by the Startup ACM. This performs some bookkeeping and locks some registers to prevent hostile software or firmware from changing critical hardware settings.
    - The SINIT ACM contains only one call and is called by the OS or applications running under the OS in order to perform a measured launch (DRTM).

    Both ACMs always run in a special internal CPU memory that prevents DMA accesses to the memory and any snooping of the ACM code and data.

---

[2]The Startup ACM isn't a separate ACM, but a function contained in the BIOS ACM. The misleading name has historical roots.

- GETSEC: This is a special Intel TXT assembly language instruction that invokes a function determined by a register setting. These functions invoke microcode flows used to enter, launch, and exit ACMs and exit the measured launch environment (MLE).[3] Which sub-functionality *(leaf[4])* is invoked by the GETSEC instruction is determined by a register setting. This is how the BIOS ACM *Lock Config* and SINIT ACM calls are invoked.
- *BIOS enabling for Intel TXT*: There is a table inside the BIOS, the firmware interface table (FIT), that tells the microcode and ACM whether Intel TXT is enabled, where the BIOS ACM is located, and which sections of BIOS to measure.
- *TPM*:
  - PCRs in the TPM are used to store measurements of components involved in the boot process. Some of these PCRs can only be extended by microcode, and some are only extended by ACMs.
  - NV indices are used to track some state information required by the verified launch process.

  The specifics of PC-compatible TPMs are described in detail in the TCG PC Client Platform TPM Profile (PTP) Specification. That specification describes the accessibility and number of the PCRs, special interfaces for measuring BIOS boot code, and other special TPM features used to support PC platforms.
- *OS/middleware enabling for Intel TXT:* The OS or middleware has to start the measured launch. In some cases, this might be an application or module running under the OS; in others, it might be a commercial virtual machine manager (VMM) software package.
- *High level applications that use Intel TXT to make security decisions:* Intel's Mount Wilson software is an example of this. For more examples and a much more detailed explanation of such high-level descriptions, read the book *Building the Infrastructure for Cloud Security: A Solutions View* (Apress, 2014).

All of these components work together to enable Intel TXT.

[3]For a full description of this instruction and its leaves, see the "Safer Mode Extensions" chapter in the *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 2B. This manual can be downloaded from www.intel.com.
[4]*Leaf* is TXT jargon for a sub-function within an ACM.

Intel TXT Boot Sequence

Let's look at one possible boot sequence at a medium level of detail. If you desire more details, see the book *Intel Trusted Execution Technology for Server Platforms* (Apress, 2013).

One quick note about error handling so that we don't have to describe it repeatedly in the following sequence: if a failure occurs at any point in the sequence, a chipset register is written with an error indication. This chipset register, `TXT.ERRORCODE`, is only writable by ACMs and microcode to prevent less privileged and possibly hostile code from clearing it. An error value in this register prevents a measured launch later in the boot cycle, as described shortly.

Figure 22-1 and Figure 22-2 illustrate the Intel measured launch process and how various components interact with the TPM. Figure 22-1 is a complete timeline from power-on through launching a trusted OS. This includes the SRTM before OS boot and the DRTM initiated by the OS. Figure 22-2 provides more detail about the secure launch sequence, specifically the steps taken to verify that both the platform and the system software are trusted.

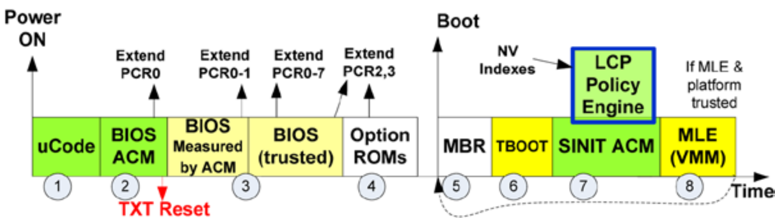

Figure 22-1. *Intel TXT boot timeline*

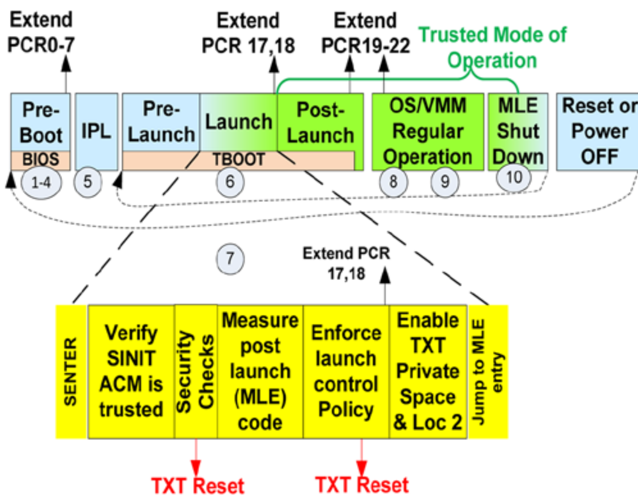

Figure 22-2. *Breakout of measured launch details*

The boot sequence is illustrated by Figure 22-1 and consists of two stages: the SRTM stage and the DRTM stage. The SRTM stage starts with the CPU microcode and extends up to OS boot. The DRTM stage starts when the SINIT ACM is invoked and extends through OS boot.

The first part of the sequence (SRTM) starts at power-on and protects against BIOS and reset attacks:

1. *Microcode*: When reset is de-asserted, the microcode checks the BIOS FIT to determine where the BIOS ACM is located in the BIOS image. The microcode verifies the signature of the BIOS ACM and does some other sanity checks on the ACM. If all is well—the ACM is uncorrupted, and it's the correct ACM—then the microcode starts the ACM running in protected CPU internal memory.

2. *Startup ACM*: The BIOS ACM contains a few different entry points that can be invoked by microcode or the BIOS. The Startup ACM call is invoked by microcode when the platform is powered-on or reset. This call's main function is to measure certain portions of the BIOS that must be trusted to operate correctly in order to guarantee system integrity, as well as to extend those BIOS measurements into PCR0. The regions of BIOS to be measured are specified by entries in the FIT table which are configured by the BIOS OEM. Some critical regions of the FIT table itself as well as the reset vector and some other regions of BIOS are required to be measured, and the BIOS ACM ensures that this is the case. Other regions of BIOS can be optionally measured, and it's up to the BIOS developer to properly configure the table to measure the correct regions of BIOS. The whole BIOS image doesn't need to be measured, and any regions of flash memory that can change under normal boot situations aren't measured, because this will cause false failures. At a minimum, to guarantee system integrity, the boot block of the BIOS must be measured—this block includes the basic system and memory initialization code. If the Startup ACM detects an error (probably indicating some sort of security issue), it sets an error code in `TXT.ERRORCODE` register and resets the CPU, and then the microcode directs the CPU to the reset vector. In this case, only a non-verified launch is possible. If the Startup ACM code completes successfully, the BIOS is executed.

3. *BIOS continues the static chain of trust*: The BIOS continues the chain of trust by measuring any additional BIOS code to PCR0 and measures other platform components to other PCRs. BIOS also creates a log of everything measured to the PCRs. All code in the BIOS trust boundary must be measured before that module executes. And before the BIOS executes any unmeasured code (code outside the BIOS trust boundary), it calls the BIOS ACM to lock the platform configuration to prevent untrusted code from altering the platform configuration. These calls to the BIOS ACM also test and perform security checks to ensure system integrity.

4. *Option ROMs*: Unless provided by the OEM, option ROMs are outside the trust boundary and option ROM code is measured into PCR2 while any option ROM configuration is measured into PCR3.

5. *OS boot*: When the BIOS completes, it boots to the OS loaded on the system. The OS is running in normal, non-verified boot mode, but it's locked and loaded to perform the DRTM phase of booting.

The second part of the sequence (DRTM) starts at the `GETSEC(SENTER)` leaf, which is invoked by the OS or a software component running in the OS. This provides a dynamic root of trust for measurement that measures the SINIT ACM and the MLE, which is sometimes the VMM.

6. *Measured launch*: When the OS wants to boot into trusted mode, it executes the `GETSEC(SENTER)` instruction. This causes a microcode flow that verifies the SINIT ACM in a manner similar to the BIOS ACM (see steps 1 and 2), loads it, and starts executing it.

7. *SINIT ACM*: The SINIT ACM verifies that no other security issues have occurred by checking the `TXT.ERRORCODE` register. It does some hardware configuration checks for certain security issues. It then measures the trusted OS code. The ACM also includes a Launch Control Policy (LCP) engine that performs policy checks, which includes checking the measured OS code and PCRs against lists of known good values. If any checks fail in the SINIT ACM, a platform reset is performed. If all is well, the ACM performs the measured launch and the OS enters secure mode. This is referred to as the *Measured Launch Environment (MLE)*. Measurements of the SINIT ACM, policies, and measured OS code are extended into PCR17 and 18.

8. *Trusted mode*: At this point, the trusted environment has been enabled, and the OS has access to Locality 2 and thus the dynamic PCRs. The trusted OS continues the dynamic chain of trust by measuring additional OS components and configuration into PCRs 18–22.

9. *Applications*: Local applications can use the values in PCRs to seal secrets that can only be unsealed when the platform is in that same trusted environment. For example, the OS can seal an encryption key it uses to encrypt private and privileged information. Only when the platform successfully performs the measured launch can the OS recover the key and decrypt the data. This is sometimes referred to as *local attestation. Remote attestation* is where external agents use the PCR values to make a trust decision—perhaps quarantining an untrusted platform while connecting trusted platforms to the production network.

10. *Termination*: The final stage is when the OS terminates the trusted environment. This can either shut down the platform (power-off or restart) or just exit the trusted mode, in which case the OS can re-enter it by performing another measured launch without the need to reset the platform. After the MLE shutdown, the OS no longer has Locality 2 access to the TPM.

This seems like a lot of detail, but we've actually skipped the low-level details of the Intel TXT policy, the security checks performed by the ACMs, the details of how TPM NV indices are used for communicating TXT status, and the BIOS enabling and provisioning of TXT.

How TPM 2.0 Devices Are Used

So, how do TPMs fit in this picture? Intel TXT uses PCRs and NV indices, primarily. Other TPM 2.0 features figure into how PCRs and NV indices are accessed and used: special hardware-triggered TPM commands, policy commands, and localities. These are described at a high level here.[5]

[5]TPM 1.2 also had PCRs, NV indices, hardware-triggered TPM commands, and localities. Policies and algorithm agility are the new TPM 2.0 features used by TXT.

NV Indices

NV Indices play an important role in Intel TXT. They are used to do the following:

- Securely pass information and states between ACMs
- Securely maintain state between platform resets and power cycles
- Allow OEM and platform owner to provide hashes of two policy lists, platform supplier and platform owner, of known good platform configurations
- Protect OEM and user policies from malicious alteration

Access to these indices is controlled by index attributes and a combination of password and index policy authorizations as described in Chapters 13 and 14 of this book. The ACM verifies that the attributes are correct before trusting their content.

PCRs

PCRs are used by both ACMs. Because TPM 2.0 supports algorithm agility, Intel TXT supports this agility at all levels from ACMs through Intel TXT launch-measured policies and BIOS trust policies. The details of this agility support are described in detail in the *Measured Launched Environment Developer's Guide*, which you can download from Intel's web site, and the *Intel TXT BIOS Writer's Guide*, which is available to OEMs.

The BIOS ACM extends the BIOS measurements and other early initialization values into PCR0. BIOS extends measurements of other platform configuration components into PCR0-7.

When doing a measured launch, the `GETSEC(SENTER)` instruction microcode performs the special hardware-triggered `_TPM_Hash_Start`, `_TPM_Hash_Data`, and `_TPM_Hash_End` commands. These commands are triggered by writing to special TPM interface registers that can only be written from Locality 4. Chipset hardware restricts access to these Locality 4 registers to hardware or, in this case, microcode. The special hash commands extend PCR17 with measured launch measurements during the microcode's execution of the `GETSEC(SENTER)` instruction.

After entering the SINIT ACM, this ACM extends other dynamic launch measurements into PCR17 and PCR18. If the Intel TXT measured launch policies are satisfied, then the OS is trusted and has access to PCRs 17-22; the OS uses these to measure additional OS code and OS configuration. Later, when higher-level software makes decisions about levels of trust, these measurements are used.

Conclusion: Intel TXT

This completes a high-level view of how Intel TXT uses TPM 2.0 devices. If you're interested, you can dive into the details by accessing the Intel documents referenced earlier.

ARM® TrustZone®

ARM TrustZone has been a feature of the ARM processor architecture since 2002 and first appeared in real processors—specifically the 1176JZF™—shortly afterward in 2003. Since then, not much has changed with TrustZone itself, but many additional features, technologies and use cases have grown up around it.

It's not uncommon for TrustZone and Intel TXT to be compared and/or lumped together as each architecture's 'security extension', but below a rather superficial level the two aren't particularly similar and such comparison doesn't aid understanding. This section explores a little of what TrustZone is, how it works, and how it relates to TPM technology. At a high level, TrustZone provides a safe place for a software TPM implementation to execute.

High-Level Description

At the simplest level, TrustZone provides a facility to create a virtual second processor inside a single system on chip (SoC). Through the implementation of a special operating mode, the SoC is able to create two separate parallel software stacks (or *'worlds'*): the Normal World (NWd), which runs the main OS and user interface, and the Secure World[6] (SWd), which runs a trusted software stack implementing security features. The two worlds are kept separate by the SoC hardware so that the main OS can't interfere with programs or data in the SWd. This enables users to retain trust in the integrity and confidentiality of SWd data even when they can't trust the state of the device as a whole.

Typically, a system designer doesn't want to impact the user experience of the device and so keeps the SWd hidden away, often using it to create a virtual security processor that the main OS calls when needed. For the most part, this idea of a virtual security processor is useful, but one very important detail must be made clear: while the SWd is completely protected from direct access by untrusted NWd code, the reverse isn't true. SWd code can, in principle, access *any* memory or device in the system. This asymmetric setup has many positive implications—high-speed data transfer and the ability to integrity-check NWd memory among them—but it also gives the SWd control over the entire device, not just the security module it implements.

TrustZone Is an Architectural Feature

The first thing to understand about TrustZone is that it's an architectural feature of ARM. And to understand that, you need to remember how the ARM partner ecosystem works.

[6]Note a slight problem of terminology here. The original naming of these architectural features follows a *secure* vs. *non-secure* theme, but as we all know, there is no such thing as absolute security: every protection system has its limits. In recent years, this terminology has given way to the more subjective *trusted* vs. *normal* concept, but remnants of the *secure* naming remain in the names of various components. This chapter uses the *(non-)secure* and *(un-)trusted* terms interchangeably.

ARM (the company) doesn't make chips itself: it designs processors and subsystems and controls an architecture specification that other companies take as the blueprint for their own chips. An architectural feature is something that is baked into the architecture specification and is implemented through standard mechanisms and signals (not as software or an auxiliary module/IP block) and is promised to be compatible on any ARM-based SoC regardless of any differentiating features they may implement. ARM-based SoCs are required to conform to the architecture specification (and pass a conformance test), so by specifying it in the architecture, it's assured that all such SoCs[7] have TrustZone.[8]

Another principal driver for implementing TrustZone as an architectural feature is that the security separation is then enforced by the chip hardware and doesn't rely on software or logical access control systems (which always fall to bugs in the end). This benefit is realized in ideal conditions and makes TrustZone extremely elegant and robust, although there are practical limitations on how much device makers can rely on this in the real world.

Protection Target

TrustZone is designed primarily to defeat software-borne attacks[9] such as those coming from rogue websites, errant downloads, root kits, and the like. It isn't designed to protect against concerted, targeted hardware penetration or lab attacks (like a smartcard might be). This makes sense when we consider the evolution of computing devices over the past decade or so: they have become increasingly networked, connected, and dynamic. Bulk data transfer is the norm, and data and applications flow seamlessly from one device to another with limited checks and balances. In such an environment, the growth in potential for scalable indiscriminate software attacks far outstrips those for targeted physical intrusion.

To be clear, in the TrustZone threat model, all software in the NWd is considered potentially hostile (either by rootkit infection or by deliberate replacement of kernel or similar). So while the SWd and NWd kernel often work together to provide overall device and application security, the SWd should never rely on information it receives from the NWd when making security decisions. This is important when considering TPM-like use cases.

---

[7]Specifically Cortex™-A class (or *application*) processors. ARM also designs R (*realtime*) and M (*microcontroller*) class processors, which don't have the kind of TrustZone feature described here.
[8]As you'll see later, simply having TrustZone isn't necessarily useful. It does have to be implemented correctly, something that requires skill and care.
[9]The term *shack attack* is sometimes used in association with TrustZone to describe a low-value, low-skill type of physical attack somewhere between the all-software hack attack and the high-end, skilled, and expensive lab attack. An example of a shack attack might be nondestructive bus probing on exposed wires. The degree to which an SoC can protect against shack attacks depends on the chip hardware design and isn't inherent to the TrustZone system.

System-Wide Security

ARM often describes TrustZone as *system security*,[10] but what does that mean? In this case, the *system* refers to everything in the SoC connected to the central processor by the AMBA®[11] AXI™ bus.[12] So in addition to providing simple memory and process separation for the two-worlds model, it also extends protection to data and interrupts handled by peripherals.[13]

Bus masters can be marked *secure*, meaning they're controlled by software running in the SWd, or *insecure*, meaning they can be accessed by either world.[14] When a secure peripheral interacts with the system, nothing in the untrusted world can see it or directly interfere with it: not even kernel code. Typical use cases for such a thing would be a Secure Element chip (cryptographic key storage device not accessible to normal code) or a touchscreen UI (trusted user interaction).

Implementation of TrustZone

The successful implementation of TrustZone in an SoC and system depends on many aspects of design but there are three major pieces to consider: the *NS bit*, the *Monitor,* and *secure interrupt handling*.

The NS bit

The NS (or 'Non-Secure') bit is the central manifestation of TrustZone in the ARM processor architecture. It's a control signal that accompanies all read and write transactions to system bus masters, including memory devices. As the name suggests, the NS bit must be set low in order to access SWd resources.

To understand how something so simple can reliably achieve world separation, it's sometimes useful to think of NS as an extra address bit[15] that effectively partitions the memory space into two parallel logical regions: 32-bit space plus NS. This analogy makes TrustZone isolation and error behavior intuitive: attempts from NWd to access SWd memory will fail, even if it knows the exact 32-bit address it wishes to attack, because the 33rd bit is different and so doesn't map to the desired memory location.

---

[10] www.arm.com/products/processors/technologies/trustzone/index.php.

[11] Advanced Microcontroller Bus Architecture. See http://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture for further definitions and acronyms.

[12] Technical note: Only AXI masters are able to correctly preserve TrustZone signals. Work is required to securely integrate AHB™ or APB™ devices.

[13] Again, *peripheral* here refers to masters connected directly to the AMBA AXI bus inside the SoC. It doesn't mean external devices like VDUs or printers.

[14] Remember the asymmetrical nature of TrustZone: SWd can access everything.

[15] The "33rd (or 41st or 65th) address bit" analogy can fail when you look at certain deep details, but it's close enough to be useful.

Clearly the security of the system would break down if NWd code were somehow able to set the NS bit in resource requests directly, so it's set, maintained, and checked by processor registers and bus components such as the memory controller and address space controller. Returning to the 33rd bit analogy for a moment, code makes a normal 32-bit request; and the processor hardware, knowing that the code is executing in insecure mode, adds NS=1 to the transaction.

The Monitor

Of course, nothing is ever quite as simple as a single bit in the architecture. A small amount of firmware is required to coordinate the two worlds, facilitate switching, and so on. This firmware[16] component is called the *Monitor*.

Alongside the two explicit operating modes (Secure and Non-Secure), there is a third processor mode called *Monitor mode* that runs a third separate software stack. In order to transition from NWd to SWd (or vice versa), requests must transition through Monitor mode and the Monitor firmware ensures that the transition is allowed, orderly, and secure.

The Monitor is able to access all the crucial security data in the system, so its quality and integrity are paramount. The code should be as small as possible and tested and reviewed[17] regularly in order to be, if such a thing is possible, bug-free.

World Switching

When NWd software wishes to contact SWd, it must issue a Secure Monitor Call (SMC) instruction. This invokes the Monitor, which must set the state of the NS bit in the Secure Configuration Register in the System Control Processor (CP15) (so that bus and memory devices know which world is executing and therefore calling them) and bank-sensitive registers to keep the system secure and consistent.

SMC calls are very simple: they take a single 4-byte immediate value that indicates to the software in the SWd what service is being requested, and the SWd runs that service. It's the responsibility of the system designer(s) to agree on conventional numbering and meanings for each value.[18]

---

[16]Note a coming confusion: from ARM V8, there is an official definition of *firmware* for Exception Level 3 (high privilege level) that is more than just the Monitor components. This text only refers to the code responsible for coordinating world switching, not any other firmware duties such as power management.

[17]The Monitor may even be a legitimate target for formally proven code.

[18]To help with this, ARM publishes various recommended calling conventions, but the system isn't required to follow them.

Interrupts

Earlier we introduced the idea that interrupts from secure peripherals can be routed directly to the SWd without ever passing through any untrusted code at any privilege level. At this point, it's important to introduce another configuration for peripherals: not secure or insecure, but *switchable*. Some peripherals (a touchscreen, for example) only need to be secured part of the time: when executing sensitive transactions. At all other times, it's acceptable, even required, for the NWd to have control.

To police this and ensure that the correct software stack has control at the correct time, all such interrupts are actually caught by the Monitor, and the Monitor decides (based on a configuration table) which driver (SWd or NWd) should receive the interrupt. When entering a secure transaction, the SWd can reserve the peripheral, meaning it receives all the interrupts. When it has finished, it can release the peripheral, informing the Monitor that it should send interrupts on to the NWd driver instead.

To deal with the various practical issues of performance, potential conflicts, and so on, a typical ARM system reserves the two interrupt signals for separate purposes: IRQ[19] for normal interrupts and FIQ[20] for secure.[21] This allows certain efficiencies such as static routing tables for certain events.

Relationship to TPMs

Historically, ARM SoCs have been most prevalent in mobile devices: smartphones, tablets, and the like. As such, TrustZone systems haven't typically *used* a *separate* hardware TPM, but rather have used TrustZone *as the TPM*.

Starting around a decade ago with the Mobile Trusted Module specification, and continuing today with the TPM 2.0 Mobile and PC Client specifications, the trusted computing community has developed the concept of a *firmware TPM*. With fTPM, rather than relying on separate hardware chips, the TPM functionality is implemented in a protected firmware execution space such as TrustZone and then called by the NWd OS for measurements, sealing, and so on in the normal way.

While no hard-and-fast requirements or architecture are specified for the precise implementation of the fTPM (beyond conformance to the TPM 2.0 library specification of course), the operating environment is required to provide some fundamental protection for the TPM roots of trust and PCRs. In keeping with the TrustZone protection target, no software outside of the TPM implementation should be able to modify or access roots of trust directly, or manipulate PCRs except though the authorized interfaces.

A well-implemented TrustZone system is able to provide these guarantees (and, indeed, several implementations are commercially available).

---

[19]An interrupt request (IRQ) is a signal sent from a hardware peripheral to alert the processor to an event.

[20]A fast interrupt request (FIQ) is an additional signal like IRQ but is (supposedly) handled faster.

[21]Although not actually required, there are two reasons for this recommendation: compatibility (existing NWd software makes much more use of IRQ than FIQ) and security (the ARM architecture allows for masking control of FIQ in CP-15 but not IRQ).

AMD Secure Technology™

The AMD Secure Processor™ (formerly known as the Platform Security Processor [PSP]) is a dedicated hardware security subsystem that runs independently from the platform's main core processors and is integrated into the SoC. It provides an isolated environment in which security-sensitive components can run without being affected by the software running as the main system workload. The PSP can execute system workloads as well as workloads provided by trusted third parties. Although system workloads are preinstalled and provide SoC-specific security services, the system administrator has complete control over whether and which third-party workloads are installed on the PSP. The PSP is made up of the following components:

- Dedicated 32-bit microcontroller (ARM with TrustZone technology)
- Isolated on-chip ROM and SRAM
- DRAM carved out via hardware barrier and encrypted
- Access to system memory and resources
- Secure off-chip NV storage access for firmware and data
- Platform-unique key material
- Hardware logic for secure control of CPU core boot
- Cryptographic coprocessor (CCP)

The PSP uses the ARM TrustZone architecture, as described in the section on ARM TrustZone, but there are some differences: rather than being a virtual core, the PSP is a physically disparate core integrated into the SoC that has dedicated SRAM and dedicated access to the CCP. The PSP provides the immutable hardware root of trust that can be used as the basis for optionally providing the chain of trust from the hardware up to the OS.

The CCP is made up of a random number generator (RNG), several engines to process standard cryptographic algorithms (AES, RSA, and others depending on processor model), and a key storage block. The key storage block contains two key storage areas: one dedicated to storing system keys that can be used by privileged software but that are never readable; and the other into which keys can be loaded, used, and evicted during normal operation by software running either on the PSP or on the main OS. During boot, SoC-unique e-fused keys are distributed to the CCP system key storage block.

Hardware Validated Boot

Hardware Validated Boot (HVB) is an AMD-specific form of secure boot that roots the trust to hardware in an immutable PSP on-chip ROM and verifies the integrity of the system ROM firmware (BIOS). The PSP ROM contains the initial immutable PSP code. The PSP ROM validates a secure boot key and then uses the key to validate the PSP firmware, which it reads from system flash. The PSP firmware loads and starts the system application execution. The system manufacturer can choose whether the PSP validates the BIOS platform-initialization code. The PSP then initiates BIOS execution. The PSP completes its own initialization and enters steady state while the BIOS and OS finish booting on the x86. The platform manufacturer decides whether to implement UEFI secure boot. The platform manufacturer also decides what interfaces are provided for the user to select whether UEFI secure boot is enforced. In this way, the platform manufacturer decides when to terminate the chain of trust that was rooted in the immutable hardware.

Figure 22-3 shows the scope of HVB as it relates to the UEFI secure boot.

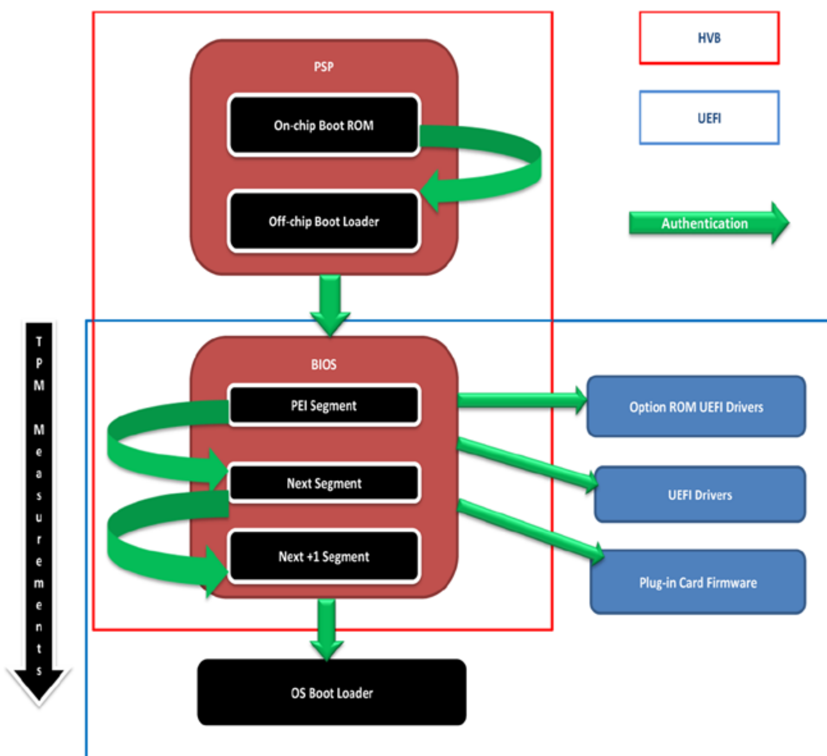

Figure 22-3. *Hardware Validated Boot Overview*

TPM on an AMD Platform

As a founding member of the Trusted Computing Group, AMD strives to support a wide range of options for the OEM and platform owner. To this end, platform manufacturers have several choices when integrating TPMs into AMD-based platforms. Platform manufacturers can continue to choose among the discrete TPM hardware options that are widely available; or the platform manufacturer can choose to integrate an AMD-provided TPM application as one of the system applications running on the PSP SWd. This firmware TPM utilizes the CCP for cryptographic processing.

SKINIT

`SKINIT` is the instruction that initiates the late launch CPU reinitialization to start the DRTM. `SKINIT` takes one parameter: the address of the Security Loader (SL) code. The SL must fit within 64KB of memory known as the Security Loader Block (SLB), which is protected from tampering and snooping. CPU microcode ensures that the CPU is reinitialized to a known state so that the developer can launch whatever SL code they need to run in the secured state. The SL is expected to validate and initialize a Security Kernel (SK) and then to transition control to the SK. The `SKINIT` instruction writes the contents of the SLB to an address that is redirected into the TPM via the `_Hash_Init`, `_Hash_Start`, and `_Hash_End` signals. These signals measure the contents of the SLB into PCR 17. Further details about the CPU characteristics that are validated and how the `SKINIT` instruction works are available in the *AMD64 Architecture Programmer's Manual Volume 2: System Programming.*[22]

This concludes a whirlwind overview of AMD Secure Technology™ that covers the high points of the introduction of an on-chip hardware root of trust into AMD SoCs. More information can be found on AMD's web site: www.amd.com/en-us/innovations/software-technologies/security.

Summary

This chapter has discussed three platform technologies that use TPM 2.0: Intel TXT, ARM TrustZone, and AMD Secure Technology. There are other technologies on PCs and other platforms that also use TPM 2.0, and, we hope, many more will be developed in the future. And this is where you, the reader, come in. Go out and do wonderful things with TPMs!

[22]http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/.