CHAPTER 11

Looking Ahead: Tomorrow's Innovations Built on Today's Foundation

Creativity is not the finding of a thing, but the making of something out of it after it is found.

—James Russell Lowell

Up to this point, this book has revealed the technical details of Intel's security and management engine, with the focus on the architecture and design of its firmware infrastructure. For the past several years, the engine has been serving as the trusted computing base of many state-of-the-art security technologies delivered by Intel platforms. Looking ahead, more innovative creations are to be done on the engine to make the most out of it. What are the next big things to come?

This chapter wraps up the book by first reviewing the critical building blocks of the engine and then briefly brainstorming next-generation technologies that can be built on the engine to further improve the security computing experience for people.

Isolated Computing Environment

The embedded engine was initially introduced by Intel in the south bridge as a *management* engine to resolve the hard problem of enterprise network administration. Managing, maintaining, and supporting network computers in organizations used to be stressful and expensive. For example, when an end-point computer has crashed, the information technology technician often has to make an onsite visit and debug the issue. Furthermore, monitoring statuses of all computers on a network is a difficult task.

Various software and hardware management solutions come with their advantages and disadvantages. To summarize, the cost of software tools is relatively low; however, software suffers constraints that cannot be easily overcome, such as security and dependency on the operating system. On the other hand, hardware methods are stable, more robust against attacks, and independent of the system under debug, but unfortunately, their higher price tags have prevented them from widespread deployment.

Intel's AMT[1] (advanced management technology)—built on the management engine and a key feature of Intel vPro—is both hardware and software. The AMT is hardware because it is natively embedded as part of the computer's chipset; it operates independently of the host operating system; and more importantly, its security is rooted in the hardware. The AMT is also software because the majority of its functionalities are realized by specific software programs that are compiled into the platform's flash device. Thanks to its dual identity, the AMT enjoys both the stability, security, and independency of hardware solutions, and the flexibility and affordability of software solutions at the same time.

The security and management engine features a dedicated processor, backbone hardware, fuse blocks, memory, and nonvolatile storage. It is designed to run normally, regardless of the state of the host. It can communicate with the host operating system and access the host's physical memory (with certain exceptions). The engine's isolation nature makes it significantly less vulnerable to threats and attacks from the host. Therefore, it is an ideal location for not only platform-level management and security solutions, but also those security applications that require the root of trust to be protected in hardware.

Nothing is perfect, and the engine has its disadvantages and limitations. For instance, to save power and prolong battery life, its clock frequency is set to hundreds of megahertz, much lower than that of processor cores on the main host system. The slower speed disallows the engine to meet performance targets of certain operations (for example, video gaming) that require extremely high throughput. Also, the engine has been designed to execute Intel-signed programs only. In the current architecture, it cannot yet be utilized as a generic trusted execution environment.

Security-Hardening Measures

The engine's capability of safeguarding itself and the sensitive data it handles is critical because of its assigned tasks and deep privileges, especially the right to read and write the host memory and its responsibility in processing high-value assets for many applications.

In order to safeguard it from being compromised, comprehensive hardening measures are applied during boot-time and runtime. The following describes a few examples at a high level:

- *Hardware root of trust*: Binary code and the data of firmware components are stored in the flash memory in the clear. Encryption is not used because the security architecture does not rely on security through obscurity. The concept of hardware root of trust contains two folds: first, the root of trust for integrity is a hardware ROM (read-only memory). Unlike the firmware in the flash memory, the binary of ROM by design is not available externally. Although, even if the code of ROM is leaked, the security of the engine should not be impacted; second, the EPID (*enhanced privacy identification*; see Chapter 5 for details) private key and other chipset keys are burned into the engine's security fuse block in Intel's factory. These keys comprise the root of trust for confidentiality and privacy for the engine.

- *Signed firmware*: Intel digitally signs the firmware image that is loaded by the ROM from the flash. The ROM verifies Intel's signature during the boot process. The hash of the public key for signature verification is hardcoded in the ROM. Applets loaded by the dynamic application loader (DAL; see Chapter 9 for details) are also signed by Intel with the same key and verified when being loaded to the engine.
- *Intact internal memory*: The engine's internal memory is intact from probing from the external world.
- *Protected external memory*: Due to the limited capacity of the internal memory, some versions of the engine require a reserved region of the host DRAM (dynamic random-access memory) to function. Because the DRAM is not in the engine's trust boundary, before being swapped to the DRAM, data pages are encrypted; both data and code pages are integrity protected. There is no point in encrypting code pages because they are available in the clear from the flash memory at rest.
- *Protected nonvolatile storage*: The engine's firmware may store secrets in the flash memory with protection for confidentiality, integrity, and/or anti-replay. The cryptographic keys utilized in these protections are derived from unique security fuses that differ from part to part.
- *Restrictive DMA (direct memory access) control*: The engine can access the host operating system's memory via its DMA devices. This powerful ability may be leveraged by malicious firmware applications to bypass memory protection mechanisms of the host. To reduce the possibility of abuse, DMA operations with the host memory are stringently controlled by a small privileged component in the engine's kernel.
- *Task isolation*: The number of the engine's firmware modules has been growing over the years. To preclude one compromised module from attacking other innocent modules, an embedded task isolation mechanism is applied. Essentially, the isolation architecture places a module in its own container and restricts its penetration into peer containers. Assets that are protected against being accessed by other containers include hardware devices, runtime memory, nonvolatile data, synchronization objects, and so forth.
- *Page attributes*: All pages of the engine's logical memory are tagged with attributes that are configured by the kernel during the boot process. The attribute entries are whether a page is code (executable) or data (nonexecutable), read-only, read-and-write, or no-access, the task it belongs to, and so on.

- *Return address scrambling*: For a function call, the return address that is stored in the stack is "scrambled" (exclusive-OR'ed) with a secret value that is randomly generated during the boot process and stored in a protected register. The prologue function calculates the scrambled return address and places it in the stack. Accordingly, the epilogue function first performs unscrambling and then jumps to the unscrambled address, if and only if the unscrambled address looks valid. With the return address scrambling in place, malware cannot easily take advantage of stack overflow bugs and instruct the processor to execute attacker's code that is located at a specific address.
- *Stack DMZ (demilitarized zone)*: Stack overflowing is commonly used by attackers. When creating the stack for a thread, the engine's kernel reserves extra pages (analogous to a DMZ) that are marked as "no-access" in the page attribute table. The attack of overflowing a stack will trigger an access violation exception if it lands on the DMZ.

The engine's reaction upon detection of a security violation varies depending on the presumed nature of the event. Minor violations may be logged and then ignored quietly. Some violations that can be a result of firmware bugs would trigger a self-reset of the engine. If a certain number of resets happen within a certain number of seconds, then the engine will enter a recovery mode and stop functioning. The engine responds to security violations that are very likely due to active attacks with ungraceful global reset in order to terminate the attack immediately. Figure 11-1 summarizes the aforementioned countermeasures into categories.

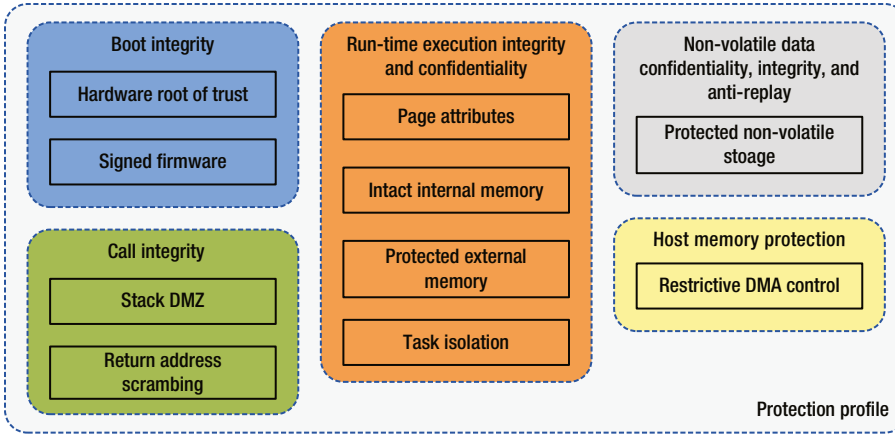

Figure 11-1. *The engine's security-hardening features*

As you can see from the list and Figure 11-1, the philosophy of *defense in depth* is exercised when designing the protection profile of the security and management engine. This means that the security of the engine tries not to rely on any single hardening measure. Consequently, a successful invasion must manage to turn down multiple fortifications, which considerably raises the difficulty of attack attempts.

For example, to install a rootkit that intends to access the host's system memory from the engine, an attack has to circumvent integrity protection, inject malicious code to one of the firmware modules, avoid being caught by runtime integrity checks, and then bypass the kernel's DMA permission filter. It is definitely a tremendous task to go through all of these defenses without triggering the alarm of the engine's security infrastructure.

■ **Note** Try to avoid relying on single hardening measures; always exercise defense in depth when architecting security solutions.

Another example of exercising the philosophy of defense in depth is reflected in the well-known FIPS (Federal Information Processing Standard) 140-2 standard[2] published by the NIST (National Institute of Standards and Technology). For a software or firmware module, the standard requires, among other things, a series of self-tests during the boot process:

- *An integrity test* using, for example, a digital signature to make sure that the module's binary image has not been altered.
- *Known-answer tests* for all cryptography algorithms supported by the module, minus the algorithm that was just checked in the integrity test. A known-answer test calls the underlying cryptography method with hardcoded input vectors and verifies that the output from the method matches the hardcoded expected result.

One can argue that the known-answer tests are redundant because, in theory, once the integrity test passes, the sequential known-answer tests that follow are impossible to fail. However, from a different angle, this double-insurance requirement can also be interpreted as a defense-in-depth strategy. For example, buffer overflow vulnerability or the like may exist in the integrity self-test implementation. An attacker that has intentionally modified the module to his benefit can possibly exploit such a bug and bypass the integrity self-test. The known-answer self-tests offer secondary defense to defeat the attack.

Basic Utilities

The following lists the majority of the engine's fundamental and generic functions that are widely needed by many applications. These have been discussed in previous chapters:

- Most standard cryptography algorithms
- Big-number arithmetic
- Secure timer
- Monotonic counter (increments by one when instructed, never decrements)
- Secure nonvolatile storage
- DMA with the host (limited to select modules only) and within the firmware memory
- HECI (host-embedded communication interface)
- Network interface, limited to select modules only
- Field programmable fuses (FPF)
- Secure firmware update

In addition, the infrastructure supports runtime debug for applications on both preproduction and production configurations. On production parts, variables that hold secret data or keys are replaced with zeroes or test values by the kernel as soon as the debug port is enabled.

Besides these basic methods, the security and management engine is equipped with several useful utilities in its extended infrastructure that are exclusively available on the engine for supporting platform-specific functions of upper-layer applications.

Anonymous Authentication and Secure Session Establishment

The EPID is an anonymous attestation and authentication scheme. It allows a verifier, which may be a local software program or a remote server, to use a group public key to verify a platform's membership of the group by examining the signature generated by the platform using its unique EPID private key. The authentication does not disclose the identity of the platform. The membership of an individual platform may be revoked under predefined circumstances, such as a detected compromise.

The SIGMA (SIGn and Message Authentication) is a protocol for mutual authentication and session key establishment. In the authentication phase, one direction (from the platform to the verifier) uses EPID, which is anonymous; whereas the other direction uses the traditional public key infrastructure (PKI) where a chain of certificates signed by certification authorities and rooted to the EPID authority prove the verifier's

identity. For the session key agreement stage, the ECDH (elliptic curve Diffie-Hellman) protocol is employed. To further raise the security bar, the SIGMA protocol can be configured to involve OCSP (online certificate status protocol) for the platform to be confident that the verifier's PKI certificate has not been revoked.

All recent releases of the security and management engine ship with an EPID private key in security fuses. The EPID and SIGMA are building blocks of many attractive features, for example, the Intel Identity Protection Technology[3] (IPT). For authentication, verifying the engine's authenticity is important to applications that take advantage of the engine's built-in functionalities. For session key agreement, the SIGMA protocol provides a convenient and secure approach to protect application-specific communications between a trusted entity and the platform, while maintaining the anonymity and confidentiality of the latter.

One of the potential problems of EPID is the heavy mathematical operations that must be conducted by the verifier and the platform. They slow down the SIGMA protocol execution and arguably worsen the user's experience. One feasible solution without introducing more computing resources is to have both the verifier and the platform save the encryption and integrity keys derived from a successful SIGMA session in their secure nonvolatile storage, respectively. This process is called *pairing*. The session keys resulted from pairing are used in future sessions, even across power cycles. The session keys may be renewed by either side requesting a new SIGMA session once a month, for example, to mitigate attacks against persistent keys and, at the same time, minimize the negative impact of SIGMA to the user's experience.

Protected Input and Output

Input (keyboard, mouse, fingerprint sensor, microphone, and so on) and output (for example, monitor and speaker) devices (I/O devices) constitute the interfaces that connect the human being and the machine. With a user-oriented mindset, safeguarding I/O devices is vital for solutions to any security problems. To realize secure input and output, the I/O devices may be connected with the security and management engine without involvement or interference of the host software. The host cannot access the clear I/O data because it is encrypted, and the decryption key is known to only the processing device and the engine.

Intel's PAVP (protected audio and video path) initially invented for supporting Blu-ray playback is a prototype for protected audio and video output. To display a secret frame, the creator encrypts the frame and transmits the encrypted frame to the graphics processor, which then decrypts and displays it on the screen. The key or the clear frame is not visible to the host. The link between the video output port and the monitor is protected by the wired or wireless HDCP[4] (high-bandwidth digital content protection) protocol. Bypassing the entire software stack is the ultimate mitigation against all types of I/O snooping attacks and it renders all malware on the host that aims at stealing the user's sensitive data through I/O ports inoperable.

Dynamic Application Loader

The security and management engine comes with a number of features stored on the flash chip. But they are not nearly enough to make the most out of the engine's rich set of capabilities. The DAL offers desirable flexibility and extends the boundary of the engine by loading Java applets to the engine from the host at runtime. As software, it is easier to create an applet, change its functionalities, and patch bugs. No firmware update is necessary for building new consumer features to the engine.

But some usages are not suitable for loading by the DAL. Generally, if a feature falls into one or more of the following categories, then it should be natively implemented in the firmware:

- Related to platform security, for example, Boot Guard and firmware-based TPM[5] (Trusted Platform Module). The defined objectives of the platform security features include measuring the integrity of the host, thus they must be running before the operating system is loaded.
- Related to system manageability, for example, AMT.
- Must be available even if the host is not running, for example, AMT and Remote Wake.[6]
- Require high data throughput.
- Code size is large.

Despite these limitations, consumer security features that launch on the operating system can still make good use of the engine through the DAL. Intel IPT sets a great example. Running new applets through the DAL, or other similar and better interfaces to be explored, will be one of the main domains for functional expansion of the engine in the future.

Summary of Firmware Ingredients

Figure 11-2 shows a summary of the security and management engine's firmware components. Notice that it is not an exhaustive list. Also notice that the engine may feature different sets of components for different products. As an example, the Bay Trail series tablets do not support AMT.

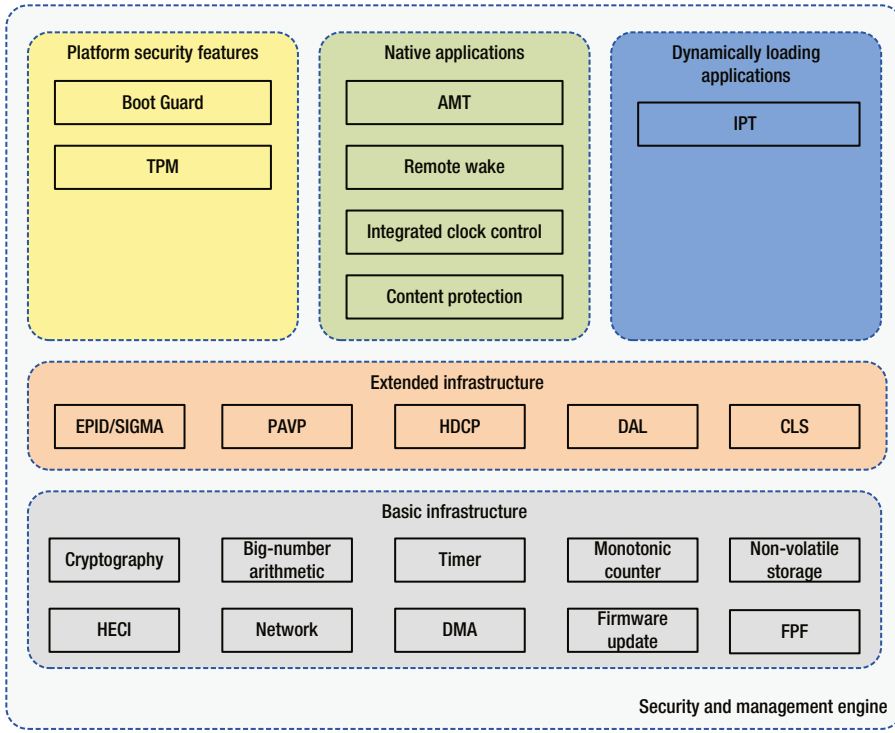

Figure 11-2. *The engine's firmware components*

Most of the firmware ingredients shown in the figure have been discussed in previous chapters. The following have not been mentioned or described in detail.

- *Big-number arithmetic*: Implements signed and unsigned addition, subtraction, multiplication, division, modulo, Montgomery reduction, greatest common divisor, least common multiple, and so forth. These arithmetic operations are extensively invoked by asymmetric-key cryptography, for example, the EPID.
- *Capability licensing service (CLS)*: Allows a remote trusted server to provision platform-specific permits and credentials to the engine. A sample usage of CLS is the Intel Upgrade Service (end of life in 2011) that unlocks advanced CPU (central processing unit) capabilities such as hyperthreading.
- *Integrated clock control*: Supports enablement and configuration of CPU overclocking.
- *Remote wake*: Supports waking up a computer from the sleep or off state from a remote location, so the user can access files on the computer. Network administrators can also use this technology to perform off-hour maintenance.

To realize its functionality, a firmware module may consume peers of the same box and modules in boxes below it. For example, IPT relies on DAL, and DAL depends on EPID/SIGMA and PAVP in the extended infrastructure, as well as cryptography, HECI, and other drivers in the basic infrastructure. However, a module does not consume a module in the boxes that are above it. For example, the drivers in the basic infrastructure box do not rely on upper-layer modules to function.

At this point, we have covered the basics of today's security and management engine. The framework is mature. The building blocks are well-established and ready to work for newer and better things. Next, let us explore future opportunities to make something out of the engine in more applications.

Software Guard Extensions

At the 2013 Workshop on Hardware and Architectural Support for Security and Privacy, researchers from Intel presented three papers describing an upcoming technology, Intel Software Guard Extensions (SGX), for securing software secrets and executions.

- *"Innovative instructions and software model for isolated execution."*[7] This article introduces the SGX's central concept of "enclave" and gives an overview of the SGX architecture and protection model. It also describes new CPU instructions for SGX, new hardware for handling the enclave page cache, and the processes for enclave creation and operation, including how an application transitions in and out of its enclave.
- *"Innovative technology for CPU-based attestation and sealing."*[8] This presentation explains the technical details of provisioning secrets to an enclave, including how to generate hardware-based attestation for software inside an enclave and how software in an enclave seals and unseals secret data.
- *"Using innovative instructions to create trustworthy software solutions."*[9] This paper focuses on the software programming model of SGX. Interestingly, for proof of concept, the authors had built on prototype hardware of SGX three trustworthy applications, namely, one-time password, enterprise rights management, and secure video conferencing. These three are perfect examples for demonstration, because they are highly-demanded real-world applications that exercise many of the SGX's infrastructural capabilities.

In a nutshell, the SGX technology enables software developers to protect sensitive code and data in enclaves that are secured at the hardware level. The protection includes encryption, integrity, and anti-replay. No software on the host, regardless of its ring and privilege, is allowed to touch others' enclaves. Moreover, the hardware can measure the trusted code in an enclave and generate attestation, so that a trusted entity, such as a service provider, is able to confirm the integrity of the code and provide premium services.

Notice the word "innovation" appearing in the titles of all three papers. Running sensitive portions of a software program in the trusted world is not a new idea. However, compared to existing solutions, the SGX's innovation is its capability of managing multiple secure enclaves, mutually untrusted, concurrently in the untrusted world. The CPU-based attestation and sealing are also innovative creations, which function like a dedicated TPM for each individual enclave.

In September 2013, Intel officially announced the SGX feature and published a programming reference manual.[10] The SGX is seemingly a very promising technology that is tasked with resolving long-lasting security problems for the software vendors and consumers. Its design is not trivial. Behind the scene are a number of hardware, firmware, and software components working together to make the SGX a reality. The security and management engine also plays a pivotal role in the solution.

The SGX architecture makes use of the engine through the generic DAL interface. Individual enclaves can invoke the engine's wide range of capabilities, including the cryptography driver, monotonic counter, secure timer, PAVP, and so forth. As the development of SGX continues, other services available from the engine may also be leveraged.

More Excitement to Come

The future development of the security and management engine can move forward in two directions. The first is to expand the family of platform-level features. By their nature, these features cannot be implemented on the host operating system because either the software stack is not trusted or the function must be available even though the host is not active. The engine's unique characteristics of isolation environment should be further utilized to realize security enforcements for the platform, as well as nonsecurity applications that require operations in the sleep state.

Second, the DAL is a milestone development that opens the door of the security and management engine to the external world. It has been used for Intel IPT and will be used for SGX. With the increasing openness of the engine, software vendors and computer manufacturers should be able to develop proprietary and innovative features that make use of the engine's infrastructure.

References

1. Kumar, Arvind, Purushottam Goel, and Ylian Saint-Hilaire, "Active Platform Management Demystified – Unleashing the Power of Intel vPro Technology," *Intel Press*, 2009.

2. National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules," http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf, accessed on April 15, 2014.

3. Intel Identity Protection Technology, http://ipt.intel.com/, accessed on April 20, 2014.

4. Digital Content Protection LLC, "High-Bandwidth Digital Content Protection," www.digital-cp.com, accessed on May 10, 2014.

5. Trusted Computing Group, "Trusted Platform Module Library," www.trustedcomputinggroup.org, accessed on March 20, 2014.

6. Intel Remote Wake Technology, www.intel.com/support/motherboards/desktop/sb/CS-032989.htm, accessed on May 10, 2014.

7. McKeen, Frank, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar, "Innovative instructions and software model for isolated execution," Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 2013.

8. Anati, Ittai, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata, "Innovative technology for CPU-based attestation and sealing," Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 2013.

9. Hoekstra, Matthew, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo, "Using innovative instructions to create trustworthy software solutions," Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 2013.

10. Intel, "Software Guard Extensions Programming Reference," https://software.intel.com/sites/default/files/329298-001.pdf, accessed on May 10, 2014.