CHAPTER

Trusted execution environment with Intel SGX

8

Somnath Chakrabarti, Thomas Knauth, Dmitrii Kuvaiskii, Michael Steiner, Mona Vij
Intel Labs, Intel Corporation, Hillsboro, OR, United States

1. Introduction

Big data analysis has become a common trend in many fields. Analyzing aggregated data across organizations has a huge potential benefit to business and research. For instance, if data from hospitals and research institutes were combined, data analytics could have a far greater chance of leading to accurate diagnostics and discovering well-targeted treatments that could improve the quality of life or even save lives. However, the privacy and security of aggregated data are critical due to regulatory requirements and the concerns of mutually distrustful data owners of losing control over these assets. As a result, huge amounts of data remain locked in silos as data owners cannot afford to reveal their data to each other for multiparty analytics.

To break big data-silos and realize the benefit of joint-data analysis, efficient and practical solutions are needed to support privacy-preserving, multiparty analysis, such that:

- Data owners retain control of their data during transfer, storage, and processing: no private data from any data owners shall be leaked; only authorized processing shall be performed and results released only to approved parties.
- There is support to scale and distribute pipelines of dependent jobs in a distributed computation and storage environment to handle the large data size and complexity of real-world analytic workflows.
- The utility of results is not unduly weakened throughout the processing.
- The existing programming and deployment model are supported to ease developing and deploying multiparty analytics applications.

We can decompose the challenge into two orthogonal subproblems:

- secure multiparty computation (MPC), and
- privacy-preserving computation.

The former ensures that the analysis on pooled datasets is executed correctly, and nothing but the final results are revealed and only to authorized parties. The latter ensures that the revealed final results of the joint analysis do not compromise anyone's privacy and helps in satisfying regulatory requirements.

Responsible Genomic Data Sharing. https://doi.org/10.1016/B978-0-12-816197-5.00008-5

The primary challenge with privacy-preserving computation is to balance privacy with utility, the third requirement above. Privacy depends on the type of input data and the algorithmic properties of computation. A common technique to achieve privacy-preserving computation is differential privacy. In its common usage scenario, a trusted server guards the data and executes differentially private algorithms that ensure that (untrusted) clients get only sanitized replies to queries; in particular, no combination of query results will enable the client to deduce any information of individual records of data owners. Although distributed versions of differentially private algorithms exist, they are considerably more challenging to design and deploy than "classical" client/server algorithms. We believe a more promising approach is to convert a client/server differentially private algorithm into a distributed version by running it on top of an MPC environment. In the rest of this chapter, we will focus on how to provide an MPC environment in a scalable and user-friendly manner.

MPC allows a set of *N* mutually distrustful parties, each with a private input, to securely and jointly evaluate an arbitrary function over their *N* inputs and to be provided with *N* outputs, one per party (see Fig. 8.1). No party learns any information from the interaction other than their own output. This is formalized by an "ideal world" abstraction, where there exists an incorruptible trusted party to whom each participant sends its input privately. This trusted party computes the function on its own and sends back privately the appropriate output. Traditional MPC solutions rely on cryptographic protocols to realize a "real world" that provably emulates, based on some cryptographic assumptions, the security properties provided by the "ideal world." In this chapter, such MPC protocols are called as crypto-MPC. Chapter 6 discusses such approaches in detail. However, even with the tremendous progress in practical general-purpose crypto-MPC protocols, they cannot handle the scale of datasets available today, for example, genome data can be measured in terra-bytes and might require days to process even if unprotected. Furthermore, crypto-MPC toolkits require rewriting applications in unfamiliar programming environments and with still fairly immature tooling. This prevents leveraging large existing efforts in big-data analytic tools across different domains, such as the genome analysis toolkit (GATK) [1] widely used in genomics.

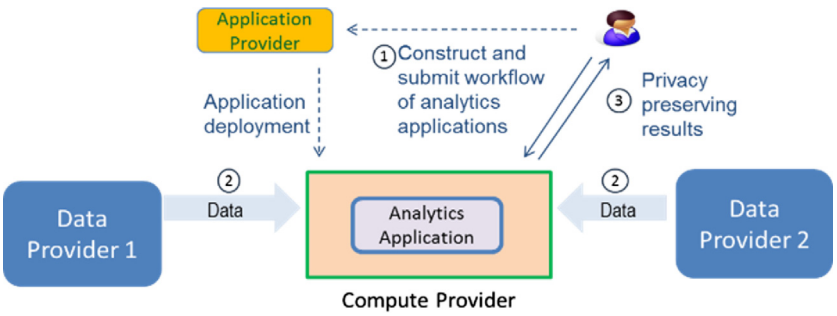

FIGURE 8.1

Multiparty analytics.

To address the earlier issues, we take notice of the rise of trusted computing technologies. In such an approach, trusted execution environments (TEEs) are constructed to host execution such that code outside of the TEE and its trusted computing base (TCB) can neither compromise the execution integrity of the TEE nor the confidentiality of the data processed inside the TEE. Over the years, some hardware security features have been introduced by the industry, such as Intel Software Guard Extension (SGX)/Trusted Execution Technology (TXT) technologies and ARM TrustZone to support setting up TEEs on compute platforms. By using TEEs with hardware-rooted trust, even the cloud provider is moved out of the trust domain. Although no computing system can be completely secure, by leveraging such hardware-based TEEs, one can emulate and provide security properties approaching the "ideal-world" similar to crypto-MPC. We call such a TEE-based approach HW-MPC.

To achieve HW-MPC for a simple system is quite intuitive as a TEE somewhat matches the "ideal world" abstraction of cryptographic MPC protocols. Building scalable MPC solutions on top of TEEs remains a challenge, though. The existing TEE abstraction and interfaces mainly support constructing and attesting TEEs on individual server platforms. Abstractions and services are only emerging on how to (1) compose TEEs to serve complex analytic workflows and meet scalability demand of computation, (2) define and enforce policies for controlling the flow of information (private data) through a workflow of analytic jobs and across TEEs, and (3) provide publicly verifiable proofs for the trustworthiness of the composite TEEs. Developing and deploying TEE applications are nontrivial tasks.

In the following, we will define in more detail the abstractions and properties offered by TEEs (Section 2), explain the realization of the TEE abstraction in Intel SGX (Section 3), explore the deployment of SGX in the cloud (Section 4), and finish with an outlook (Section 5).

2. Trusted execution environment

Execution environments refer to a collection of processors, memory, storage, and peripherals. A TEE provides computation isolated and integrity protected from the normal execution environment. The protections offered by TEEs must be rooted in hardware as it provides minimal TCB and needs to provide protection against hardware as well as software attacks. TCB is the set of hardware and software components that are critical to a TEE's security and must be trusted. The ultimate goal is to minimize the TCB in a TEE, as larger codebases are error prone and difficult to reason about. Fig. 8.2 contrasts the classical untrusted execution environment with a smaller TEE on the example of Intel SGX.

A TEE typically consists of several components: *secure bootstrapping* to help ensure the system starts at a secure initial state; *isolated execution* to protect data and computation; and secure I/O, for example, *sealed storage* to protect data at rest and, in particular, *remote attestation* to allow a remote party to identify and verify

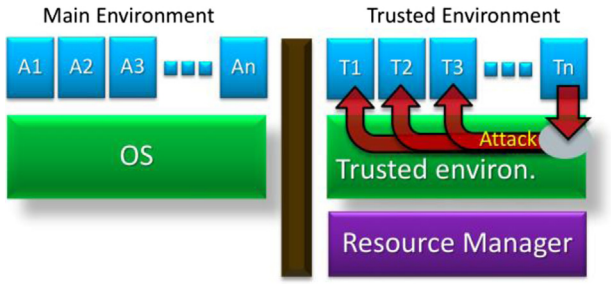

FIGURE 8.2

Classical untrusted execution environment versus trusted execution environment (Intel SGX).

the trustworthiness of a communication peer inside a TEE. Remote attestation enables the establishment of a secure channel that can be used to provision secrets, receive authenticated results, etc. TEEs should provide protections from operators as well as other privileged users accessing restricted data. There is also a strong desire to provide protection against various side-channel attacks and from physical tampering.

TEEs provide a place to stand for building trustworthy software. Some examples of TEEs in production today are Crypto-coprocessors, TEE based on trusted platform module (TPM) and related technologies (Intel TXT, AMD SVM, Intel TME/MKTME, AMD SME), Keystone Enclave for RISC-V, ARM TrustZone, AMD SEV, and Intel SGX.

The earliest secure system in the industry dates back to coprocessors such as IBM 3848 and IBM 4758 [2]. Their purpose was mostly to provide cryptographic processing capability and a means to securely store cryptographic keys. However, the 4758 and later models would also have allowed running arbitrary programs.

Since 2000s, a security coprocessor called *TPM* [3] was standardized by a trusted computing group and widely deployed in PCs. The TPM chip allows the CPU to take and store security measurements of the platform state thereby enabling secure or authenticated boot and subsequent platform integrity of a such-booted TEE. Based on these measurements, TPM's chip-unique RSA key can then be used for platform device authentication and remote attestation. Additionally, TPM provides monotonic counters for rollback security and various other cryptographic services. TPM chips usually also include some security mechanisms to make physical tampering difficult. Originally, TPM required, in a static-root-of-trust, that the BIOS and boot software are part of the TCB. Intel later added *TXT* [4] to CPUs that provide a dynamic root of trust and allows removing BIOS and initial boot software from the TCB. As part of AMD's virtualization extension SVM, an extension similar to TXT was added to AMD chips. Subsequent hardware extensions like *Intel Total Memory Encryption (TME)* and *AMD Secure Memory Encryption (SME)*, allow processor memory to be visible outside of the CPU only in encrypted form and enable further hardening of TPM-based TEEs. They provide *only* heightened confidentiality of data and *not* protection against tampering and rollback attacks.

Furthermore, as the design goal of TPM and related technologies was focused primarily on software adversaries and the overall security against physical adversaries is not very high, TPM-based TEEs might not be sufficient for the need of multiparty security settings where the owner of the computer cannot necessarily be trusted. Additionally, TPM-based TEEs always include an operating system or hypervisor in the TCB and hence have fairly large TCBs. This makes it challenging to get an assurance of code correctness high enough for sensitive operations such as processing of medical data.

The Keystone project [5] is an open-source TEE implementation for RISC-V processors. It was unveiled in 2018 and aims to provide an open-source, highly customizable TEE that can be optimized for resource usage of a particular application. Keystone is influenced by the previous commercial designs including ARM TrustZone and Intel SGX. Therefore, it implements the same TEE components as other techniques, including remote attestation, memory isolation, secure bootstrapping, and a security monitor. Different from Intel SGX, Keystone does not currently provide a memory encryption/integrity engine though such a component can be implemented as an extension. As of this writing, the Keystone project is in its early stages of development (version 0.1), and its performance, usability, deployability, and other characteristics are still unclear.

ARM TrustZone [6] goes beyond coprocessors and enables the development of isolated execution environments directly in the main CPU by supporting two domains (known as worlds): Normal and Secure. Normal operating systems and applications run in the normal domain; trusted OS and applications run in a secure domain. These domains are isolated by the CPU and components running in a normal domain cannot access any memory or other resources running in the secure domain. This model provides strong isolation enforced by hardware but does not scale well as there is only a single TEE. Running inside the secure domain a secure OS, which provides, for example, GlobalPlatform [7] based TEEs, can address this but does so at the cost of increased TCB size and, depending on OS and TEE abstractions, a restricted programming environment. This together with the fact that TrustZone is more widely available only on mobile devices but not on server systems makes TrustZone less suitable for big data analytics in multiparty security settings.

AMD Secure Encrypted Virtualization (SEV) [8] extends SME to AMD virtualization, allowing individual VMs to run SME using their own secure keys and providing remote attestation. Similar to TPM-based TEEs, it requires large TCBs including a complete virtual machine monitor (VMM). Although SEV provides confidentiality, it does not provide strong integrity protection, in particular, no protection against replay attacks. Additionally, applying SEV in the context of collaborative analytics requires overcoming the challenge that SEV's remote attestation is a-priori verifiable only by the provider of the VM and is not publicly verifiable, as required in multiparty security settings.

Intel SGX [9] provides a trusted execution environment that is intended to provide a scalable and attestable secure execution environment in a mainstream platform. In the following sections, we will describe Intel SGX in detail, explore the deployment of SGX in the cloud, and finish with an outlook.

3. Intel Software Guard Extensions

In 2015, Intel introduced a new secure extension to the upcoming Skylake CPUs called Intel SGX. Intel SGX is an actual hardware realization of a TEE providing confidentiality and integrity protection, with a particular focus on small TCB, support for legacy applications, ease of application development and deployment, and acceptable performance overhead.

Intel SGX was designed to protect secure computations even in the presence of a powerful attacker. The attacker can control the whole software stack including the operating system and the hypervisor. She can also have access to the physical machine: she can observe and modify all data in main memory (RAM), snoop on the memory bus, and control I/O devices. The *only* hardware part that the attacker is unlikely to hijack is the CPU package. Thus, with Intel SGX, the end user needs to trust only the Intel CPU.

The central concept in Intel SGX is the so-called *enclave*. Enclaves are opaque regions of memory carved out of the normal application (process). They securely execute sensitive code on sensitive data, such that no other part of the "host" process, nor any other process, nor even the privileged software (operating system, hypervisor) can access enclave data or subvert enclave code execution (see Fig. 8.3).

Importantly, SGX enclaves execute on the same hardware as other software. An SGX enclave can be thought of as a "privileged" part of a user application: the application performs its normal noncritical computations most of the time, but for sensitive critical computations it switches to the enclave mode (enters the enclave), the enclave securely runs these computations and then exits, thus switching back to the normal application mode.

From the hardware perspective, Intel SGX introduces several new features. First, a new CPU mode: whenever a user process wants to enter an enclave, the CPU

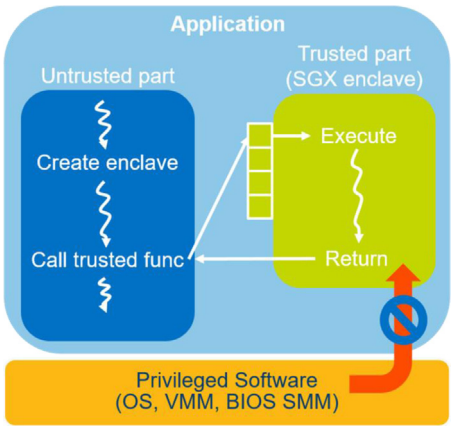

FIGURE 8.3

Interaction between untrusted and trusted parts of the application in Intel SGX.

switches to the enclave mode. In this mode, only the enclave code is allowed to execute, some security-sensitive CPU instructions are forbidden, and all enclave data are transparently encrypted/decrypted when moved out/to the CPU caches. The second new feature is an enclave page cache (EPC)—a region of RAM dedicated to storing enclave code and data pages. The enhanced memory access controller on the CPU allows memory accesses inside the EPC only for the corresponding SGX enclave, preventing attacks from other applications and privileged software. EPC also contains enclave-specific metadata to prevent subtle attacks discussed in the next sections. The third new feature of Intel SGX is the memory encryption engine (MEE)—a separate component on a CPU that transparently encrypts all data traveling from CPU to EPC and transparently decrypts data in another direction. MEE uses cryptographic keys for encryption to help provide confidentiality, message authentication codes to provide integrity, and versioning to provide freshness, for example, protect against rollback attacks.

From the software perspective, Intel SGX provides a familiar development and deployment environment. Recall that pure software solutions for secure computations (homomorphic encryption, multiparty computations) rely on complex cryptographic protocols that incur high-performance overheads and are built using specific, limited software primitives. In contrast, SGX enclaves execute normal $\times 86$ instructions on a CPU and compute over plaintext data in CPU caches. This allows to run legacy unmodified code inside enclaves over normal unencrypted data with near-native performance. (As some instructions are forbidden inside SGX enclaves, it may be sometimes necessary to adapt legacy code; see next sections.) In fact, enclaves are usually programmed as *shared libraries* and loaded inside the EPC at process initialization so that the rest of the application invokes library functions to execute secure in-enclave functionality.

Finally, the end user (who may want to run secure computations on a remote machine in another part of the globe) must gain trust in enclave execution. In particular, the user wants to ensure that the enclave executes the code she expects/trusts and that the enclave executes on a real SGX-enabled Intel CPU. To this end, Intel SGX provides a cryptographic measurement over all code and data at the enclave initialization as well as a special "report" on the CPU signed by a CPU-specific key. With the help of the remote attestation (RA) procedure of Intel SGX, the user can obtain these measurements, compare and verify them against the expected values, and by doing so gain trust in the remote enclave.

To put it all together, we will use the example of an aggregate association analysis application over shared genomic data. In our scenario, illustrated in Fig. 8.4, several parties owning genomic data are pooling their datasets to allow for higher-quality aggregate association analysis. However, as the data providers do not completely trust each other nor the researchers performing the analysis, the computation is performed in a multiparty secure manner inside an SGX enclave. In preparation (step 0), the application is provisioned and the (large) data-sets are encrypted and uploaded. To perform an analysis, the analyst launches the analysis application in an enclave at the cloud provider (step 1). After assuring themselves

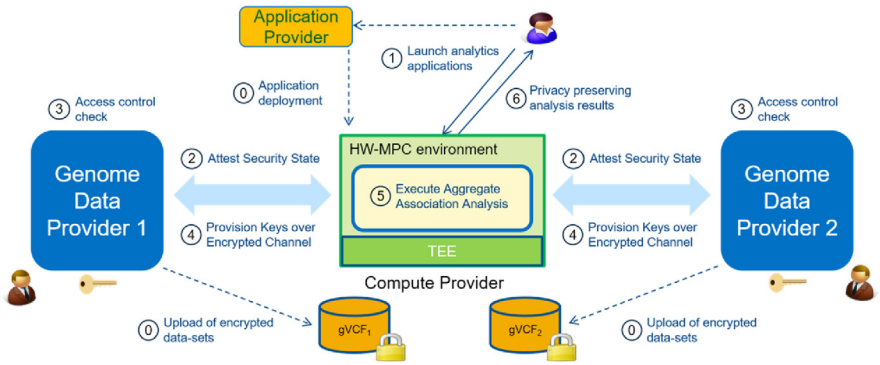

FIGURE 8.4

Example application performing collaboratively aggregate association analysis over genomic data.

that the application is genuine and appropriately protects their privacy as well as approving all involved parties (steps 2 and 3), the data providers release the decryption keys for the previously uploaded encrypted data files containing the DNA sequences to the enclave (step 4). Having now access to the data, the enclave performs the analysis (step 5) and returns the results to the analyst (step 6). Thanks to transparent memory encryption in hardware, all genomic data and all intermediate computations are confidentiality- and integrity-protected. With the help of remote attestation, the data providers and analysts can gain trust in the application code and the Intel SGX platform. This allows the establishment of a secure channel to the enclave so the decryption keys will be visible only to the enclave and the result can be authenticated and privately delivered to the analyst, ensuring the required multiparty security requirements.

3.1 Hardware architecture

Intel SGX provides a secure execution environment for applications using a standard programming model. However, Intel SGX does not trust any hardware or software components for its operation except the CPU package but strives to provide a secure and trusted execution environment. To achieve this level of protection, several new capabilities have been added to the Intel CPU architecture.

As previously mentioned, when enclave code and data reside inside registers, caches, or other logic blocks within the processor package, then any unauthorized access may be prevented. This is done by access control mechanisms built inside the CPU's SGX microcode. In addition, when enclave data leave the CPU caches to be written into protected main memory (EPC), the data are automatically encrypted and integrity protected preventing memory probes or other techniques to view, modify, or replay data or code contained within an enclave. This is done by the memory encryption engine. Fig. 8.5 shows a high-level hardware architecture of Intel SGX, with the CPU package as the trust boundary.

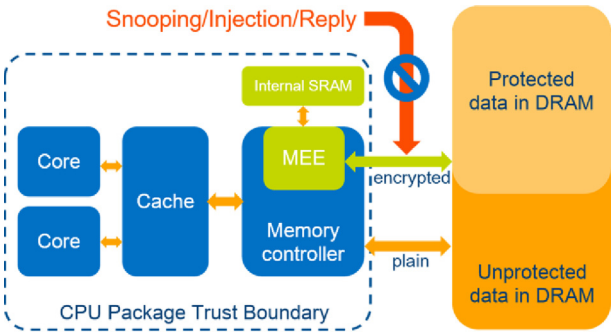

FIGURE 8.5

Intel SGX: hardware architecture.

3.1.1 Memory encryption engine and protected memory region

The MEE is a hardware unit on the CPU package that encrypts and integrity protects enclave traffic between the processor package and the main memory (DRAM). The overall memory region that an MEE protects is called the MEE region and is protected using CPU PRMRR (processor reserved memory range register). The MEE uses a single encryption key to help protect all enclaves on the platform that is automatically generated every time the CPU resets. Thus, attempts to modify an enclave's contents are detected and either prevented (during software attacks) or execution is aborted (during hardware attacks).

An SGX enabled Intel CPU implements the MEE region as a cryptographically protected DRAM region and supports the ability for the BIOS to reserve a range of memory called processor reserved memory (PRM). The BIOS allocates the PRM by configuring a set of range registers, collectively known as the PRMRR.

3.1.2 Enclave page cache and SGX data structures

To implement SGX memory protections, new hardware logic, microcode, and data structures are required. In particular, Intel SGX introduces the EPC. It is a protected memory region where enclave pages and SGX data structures are stored. This memory is designed to be protected from unauthorized hardware and software access. Code and data from all the enclaves on the platform reside inside the EPC. When an enclave performs a memory access to the EPC, the processor decides whether or not to allow the access. The processor maintains security and access control information for every page in the EPC in a hardware structure called the enclave page cache map (EPCM). This structure is consulted by the CPU and SGX instructions only, and it is not accessible by code running inside enclaves. The security attributes for each EPC page are held in this internal micro-architecture data structure.

The EPC is divided into 4KB chunks called EPC pages. EPC pages can either be valid or invalid, and a valid EPC page contains either an enclave page or an SGX structure.

Each enclave instance has an enclave control structure, SECS. Every valid enclave page in the EPC belongs to exactly one enclave instance. The system software is required to map enclave virtual addresses to a valid EPC page. In addition, each enclave has at least one thread control structure (TCS) that stores metadata for an enclave thread. Multithreaded enclaves require several TCS structures.

EPC pages and their corresponding EPCM metadata are physically stored in PRM. The following table summarizes all the earlier data structures.

Data structure	Description
Enclave Page Cache (EPC)	Contains enclave application code and data
Enclave Page Cache Map (EPCM)	Contains metadata of all enclave pages
SGX Enclave Control Structure (SECS)	Metadata for each enclave
Thread Control Structure (TCS)	Metadata for each thread

3.1.3 Enclave creation and execution

The enclave creation starts by compiling the enclave code into a binary library. The binary is then loaded into the EPC and is assigned a unique enclave identity on the platform.

The enclave creation process is divided into multiple stages: initialization of enclave control structure, allocation of EPC pages and loading of contents into the pages, measurement of the enclave contents, and finally establishing the enclave identity.

These steps are supported by the following new CPU instructions: ECREATE, EADD, EEXTEND, and EINIT. Additionally, EENTER, EEXIT, and ERESUME are used to enter and exit the enclave (described in the following). The new CPU instructions are summarized in the following table.

Instruction	Description
ECREATE	Create SECS page
EADD	Add enclave pages
EEXTEND	Measure 256 bytes of enclave page
EINIT	Finalize enclave
EENTER	Enter into enclave at predefined entry point
EEXIT	Exit enclave
ERESUME	Enter into enclave after serving exception/interrupt

ECREATE starts the enclave creation process and initializes the SGX enclave control structure (SECS) that contains global information about the enclave. EADD commits EPC pages to an enclave and records the commitment (but not

the contents) in the SECS. The memory contents of an enclave are explicitly measured by EEXTEND.

Once the process of building the enclave in EPC is completed, the enclave needs to be securely locked down. Locking down an enclave in the EPC helps prevent any further changes or additions to enclave pages from outside the enclave. The CPU will only allow entry to the enclave once it has been locked down. The finalization process is executed by the EINIT instruction. Upon successful completion of the EINIT instruction, the enclave's cryptographic signature is recorded by the CPU. This signature uniquely identifies the code and initial data loaded inside the enclave.

Intel SGX also introduces EENTER and EEXIT instructions, to enter and exit an enclave programmatically. Enclave entry and execution can start only at fixed predefined entry points. Enclave exits, however, can occur due to a fault or an interrupt. Upon such asynchronous exits, the processor invokes a special internal routine called asynchronous exit (AEX) that saves the enclave state inside the enclave, loads a synthetic state in CPU registers, and sets the faulting instruction address to a value specified during EENTER. After the fault/interrupt has been serviced by the OS, it executes the ERESUME instruction that restores the state back to allow the enclave to resume execution.

3.1.4 Enclave paging

Intel SGX architecture also offers instructions to allow system software to oversubscribe the EPC by securely evicting and loading enclave pages and SGX data structures. This enables the system software to be able to run a large number of enclaves on a platform even if the total amount of enclave memory required is larger than the EPC size available on the platform. To achieve this, system software first identifies less frequently accessed EPC pages and evicts them to make space for more active EPC pages.

Intel SGX architecture enables the contents of an enclave page evicted from the EPC to main memory to have the same level of integrity, confidentiality, and replay protection as the contents residing inside the EPC.

To achieve the same security level, SGX architecture ensures that the following conditions are met while evicting or loading EPC pages:

- Enclave pages are evicted only after all CPU cache translations to that page have been cleared from all CPU cores.
- The contents of all evicted enclave pages are encrypted and integrity protected before being written out to main memory.
- When an evicted enclave page is reloaded into EPC it must have the same page type, permissions, virtual address, and content, and belong to the same enclave as at the time of eviction.
- Only the latest evicted version of an enclave page is allowed to be reloaded. Old copies of the evicted pages are invalidated every time a new eviction of a page is made.

The following table shows all SGX instructions used for paging and their use.

Paging instruction	Description
EPA	Create version page to track the latest version of evicted pages
ELD	Load evicted page from regular memory into EPC
EWB	Evict EPC page to regular memory
EBLOCK	Block EPC page from further access
ETRACK	Track all old CPU address translations to the EPC page
EREMOVE	Mark the EPC page as INVALID in EPCM

3.1.5 Enclave teardown

Once the enclave execution is over and the application no longer needs the enclave, it can be permanently removed from the EPC. Similar to the creation process, the removal process also must follow specific steps to maintain SGX security properties. The first step involves system software ensuring no threads are currently executing inside the enclave followed by removal of all process page table mappings to the enclave EPC pages. This will ensure that no further execution can take place inside the enclave. At this stage, all regular enclave pages (except SECS page) can be removed by system software using EREMOVE instruction.

EREMOVE instruction will mark the page as an INVALID page in EPCM and adjust the child page count number in the parent SECS page. Note that once an enclave page is removed using EREMOVE instruction it cannot be added again to the enclave and the only way the page can be added again is by recreating the enclave. The SECS page must be the last page to be removed from the EPC as EREMOVE instruction ensures that all child enclave pages are removed before removing the SECS page.

3.1.6 SGX platform stack

Fig. 8.6 shows a high-level hardware/software architecture of the SGX stack on a platform. Enclaves are the trusted parts of applications, communicating with the outside world via SGX user runtime. Page tables for enclaves are managed by the untrusted OS. In the hardware, enclave pages are mapped to EPC, and the EPCM is used to enforce the access control to EPC memory. All SGX instructions that are used to manage enclaves are actually leaves of two real CPU instructions, namely ENCLU and ENCLS. ENCLU is the user-level instruction that allows applications to enter/exit/resume the enclave and manages enclave keys. ENCLS instruction can only be executed by privileged software, which is used to create/remove/measure the enclave as well as perform paging operations.

3.1.7 Remote attestation

Another important aspect of Intel SGX is the support for remote attestation. Remote attestation provides the ability to remotely gain trust in an enclave and enables the remote party to share sensitive code, data, and secrets with trust and confidence using an authenticated and secure channel.

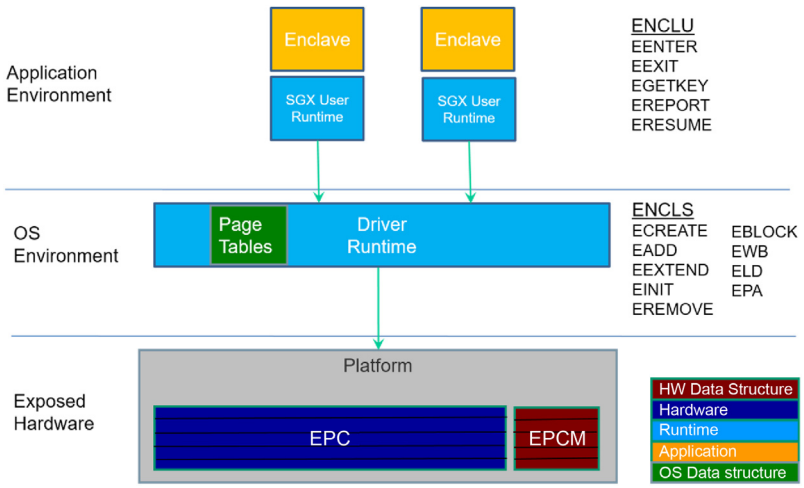

FIGURE 8.6

SGX platform stack and new instructions.

The process of establishing an authenticated and secure channel starts by generating a hardware-based attestation report identifying the enclave software as well as other security properties of the enclave including CPU's hardware and firmware properties. The measurement of the software, that is, code and data inside the enclave, is a vital parameter in the attestation report and is basically a hash digest over enclave memory as well as enclave configuration. Any attempt to change the code, data or enclave configuration will result in a different hash digest reflected in the attestation report.

Once generated, this hardware-based attestation report is then verified and signed as a quote by an Intel developed enclave called Quoting Enclave running on the same platform. This quote can then be cryptographically verified by the user at the remote end to gather authentic information about the software and the environment the enclave is running on.

The general flow of remote attestation is depicted in Fig. 8.7. An example of remote attestation and secure channel establishment with an SGX enclave is as follows:

- Once the application enclave is loaded on a platform, the remote user establishes a communication channel with the application and challenges the application enclave to prove that it is indeed the trusted enclave. Note that this initial communication channel is considered untrusted until a successful remote attestation takes place.
- The application enclave creates a manifest containing the response to the challenge and a self-generated ephemeral public key to be used by the remote user for secure communication.
- The enclave executes the EREPORT instruction with the hash digest of the manifest as an input to the instruction. The instruction generates the REPORT that binds the manifest to the enclave. The enclave sends the REPORT and the

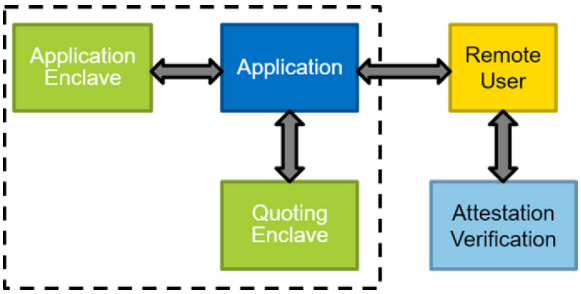

FIGURE 8.7

Remote attestation flow.

manifest to the application and it forwards the REPORT to the Quoting Enclave for signing.

- The Quoting Enclave (see Fig. 8.8) retrieves its Report Key using the EGETKEY instruction and verifies the REPORT. The Quoting Enclave then creates the QUOTE structure—which among other things contains the hash of enclave manifest, the enclave's identity and information on the platform TCB and QE identity—and signs it with its EPID key. The Quoting Enclave returns the signed QUOTE back to the application.
- The application sends the signed QUOTE and the manifest to the remote user.

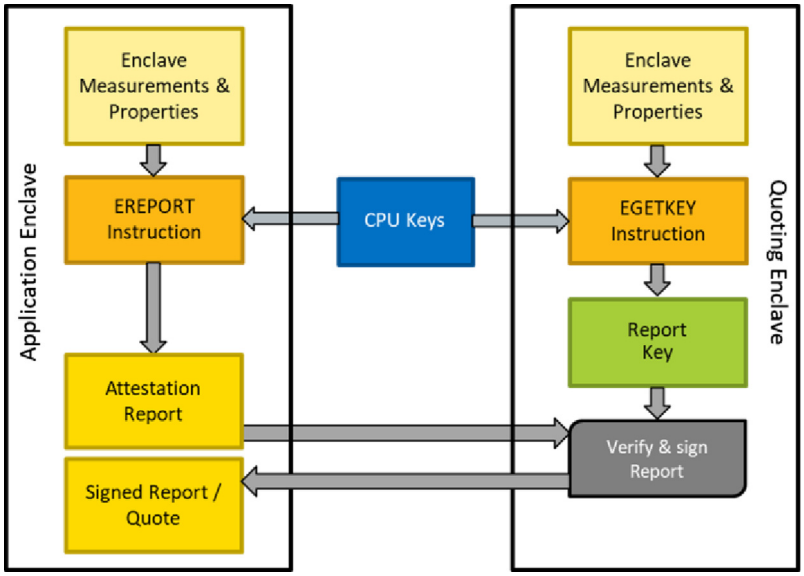

FIGURE 8.8

Signing of enclave's REPORT by Quoting Enclave.

- The remote user, with the assistance of an attestation verification service (Intel IAS by default), validates the signature of the quote. This attestation verification service is the ultimate root of trust in the enclave and the platform it runs on and provides, for example, information of up-to-date platform TCB versions and of revoked platforms. Based on this information, one can determine whether the particular platform and QE can be trusted and, if so, conclude that the (hash of the) manifest must come from an enclave with identity and properties as specified by the signature content.
- The remote user verifies the integrity of the manifest using the embedded hash digest and uses the data in the manifest as the response to the challenge that was sent in the first step.
- Once the remote user/entity is satisfied by the attested identity of the application enclave, it continues with the secure key exchange to complete the secure channel setup.

The following table describes two SGX instructions used for remote attestation.

Attestation instructions	Description
EREPORT	Generate REPORT containing enclave parameters like measurement, enclave signing key hash, and enclave properties
EGETKEY	Derive enclave- and platform-specific keys for various purposes, including the key for attestation (Report Key)

Remote attestation is an integral part of the deployment of our aggregate association analysis application (see Fig. 8.4). In particular, all the involved parties must attest the enclave: all data providers and the analyst must gain trust in the aggregate association analysis application code and the Intel SGX platform it runs on. For each involved party, remote attestation proceeds as described earlier. At the end of this process, each data provider as well as the analyst establish a secure channel to the enclave and provide it with data to perform the desired multiparty computation. For more information on remote attestation and the different types of remote attestation supported, see Refs. [10,11]. See Ref. [12] for further information on how to integrate it into secure channels.

3.2 Software development

The programming model for SGX enclaves is similar to the usual shared-library development. In particular, all sensitive code and data must be isolated from the rest of the application in a self-contained library with a few well-defined entry points. At runtime, this self-contained library is measured, loaded inside an SGX enclave, and remotely attested by end users of the applications. The application in which the enclave is embedded is called as the *host application* and is considered untrusted (see Fig. 8.9).

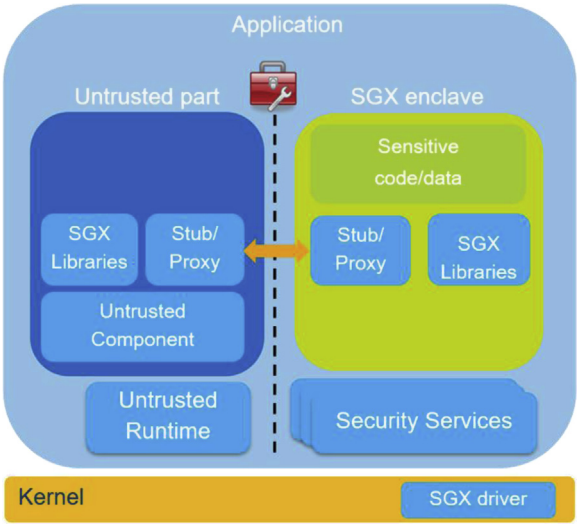

FIGURE 8.9

Intel SGX: software architecture.

The usual runtime flow is as follows. When the host application is started, it creates the enclave, populates it with sensitive code and initial data, measures the enclave, and starts it. After this initialization phase, the enclave is ready for secure computations. At runtime, whenever the host application needs to run enclave code, it passes control to the enclave by entering it, the enclave executes and exits back to the host application. At shutdown, the host application is responsible for the graceful enclave teardown.

As enclaves are executed on Intel CPU, legacy code can be ported to enclave code. Unfortunately, some CPU features are unsafe in enclave mode and are thus disabled, making it impossible to invoke system calls (to instruct the OS to send a network packet, for example) from inside the enclave. Therefore, if an enclave needs some OS functionality, it must pause its computations, exit so that the untrusted part of the application performs a system call on its behalf, and be reentered to continue execution with the result of system call. All enclave communication must be channeled through the address space of the host application.

Just like shared libraries, enclaves must be entered only through a well-defined API. The entry points (also called *entry functions* or *call gates*) of the enclave are fixed by SGX, such that the malicious host application cannot simply jump to an arbitrary point in enclave code and disrupt normal execution flow. The act of entering the enclave through one of the entry points is called *ECALL*, or enclave call. Similarly, the enclave may exit to ask the host application to perform some untrusted computation—this is called as *OCALL*, or outside call. ECALLs are akin to normal library function calls, while OCALLs are akin to callbacks. Input arguments and output results are communicated between the untrusted host

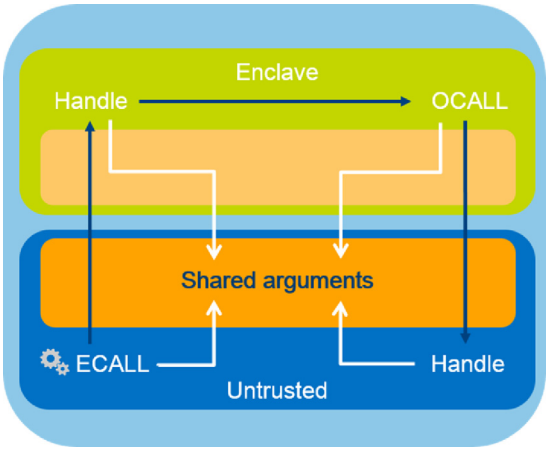

FIGURE 8.10

OCALL and ECALL interfaces in Intel SGX.

application and the enclave through untrusted shared memory. The flow of ECALLs and OCALLs is summarized in Fig. 8.10.

The internal mechanics of ECALLs and OCALLs are similar to Remote Procedure Calls (RPCs) and include complicated marshaling/unmarshalling of arguments, their verification, and enclave entry/exit. For this reason, special software tools simplify the development of the ECALL/OCALL interface, automatically generating stubs/proxies from the user-supplied high-level interface description (this is similar to, e.g., protobuf [13]).

One such tool is the Intel SGX software development kit (SDK) tailored for C/C++ applications. It provides the "edge interface" generator that creates ECALL/OCALL boilerplate code and links it into the host application. It also provides the "trusted runtime" used inside the enclave: helper SGX libraries for trusted memory management, file I/O, encryption mechanisms, random number generator, sealing of data on persistent storage, etc. The SGX SDK also provides the "untrusted runtime" used by the host application: helper SGX libraries to load, start, initialize, and measure enclaves (more specifically, these libraries instruct the SGX kernel driver to perform these actions). Finally, the SGX SDK provides additional security services such as enclave configuration, in-enclave debugger, and IDE extensions, as well as generating the expected measurement of the enclave used by the end user as a reference during remote attestation. In general, Intel SGX SDK allows developers to create and port C/C++ enclave applications with minimal effort.

SGX enclaves can only be started on an SGX-enabled hardware. On hardware without SGX support, any attempt to start the enclave will fail. The attacker can still try to emulate the SGX enclave and SGX hardware, but remote attestation in emulation mode will fail to verify.

Operations to create, initialize, and measure enclaves must be done by privileged software and are provided via an SGX driver. SGX drivers must be installed to use these operations.

Let us now describe the development process for our SGX-enabled aggregate association analysis application. We assume a simple analysis with an existing library that can be put as-is inside the enclave. This code should be separated from the rest of the application in a shared library. The library will contain several ECALL functions, for example, "prove_enclave" for remote attestation, "receive_keys" that provides the (encrypted) data file decryption keys from remote data-providers as inputs, and "run_analysis" that runs the aggregate association analysis and returns the (encrypted and integrity protected) result. The library will also use several OCALLs, for example, "print_stats" to periodically output current progress, "persist_analysis" to checkpoint intermediate results on a hard drive for fault tolerance, and "read_data" to access the encrypted data files. The developer uses the "edge interface" generator of SGX SDK to automatically create the necessary proxy code. The untrusted part of the host application will only serve as a proxy between the enclave and the user. Additionally, it may output statistics and current progress for the local administrator. In-enclave code must call SGX SDK helpers to perform sealing, generate encryption keys, etc. Because system calls are prohibited inside the enclave, our in-enclave library must not try to directly communicate with the operating system. Finally, the resulting library must be measured by the SGX SDK and can be shipped to remote machines for secure execution.

Many bioinformatics applications are much more complicated than our simple aggregate association analysis example. They may invoke system calls extensively, perform many I/O operations, and request trusted counters/timestamps. In this case, partitioning and porting applications to SGX enclaves may be challenging and time consuming, even with the help of Intel SGX SDK.

Fortunately, there are several existing frameworks to run unmodified legacy applications inside SGX enclaves, including SCONE [14], SGX-LKL [15], and Graphene-SGX [16]. They were shown to achieve good trade-off in terms of security, porting effort, and performance. Using a framework like Graphene-SGX, our aggregate association analysis could be run as-is inside the enclave, with no modifications required. This "push-button enclavization" comes at a price of a considerably increased TCB, possibly larger attack surface, and slight degradation in performance.

3.3 Security properties

Thanks to its minimal hardware TCB of only the CPU package, Intel SGX can thwart an array of hardware and software attacks. The general attack vectors are shown in Fig. 8.11. In the following, we outline the attacks and explain how the SGX architecture prevents them.

Hardware attacks. Intel SGX is one of the few TEE technologies that can mitigate direct hardware attacks. The attacker may have complete control over the physical machine and can attach malicious devices to, for example, snoop on the memory bus. Alternatively, the attacker can launch a DMA attack by attaching her device to an expansion port to directly observe physical memory. The underlying

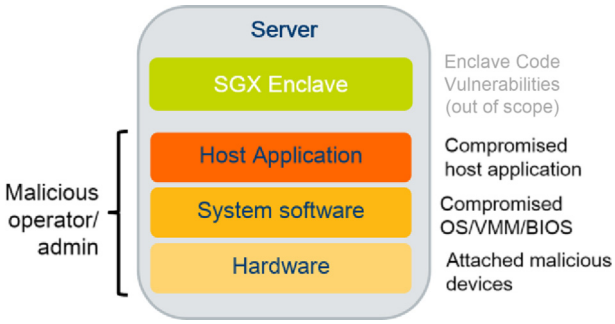

FIGURE 8.11

Attack model of Intel SGX.

memory encryption engine for Intel SGX stores all enclave data encrypted and integrity-protected in memory, thus complicating hardware attempts to read or modify enclave contents.

System software attacks. Intel SGX can protect against a range of attacks by privileged system software: firmware, hypervisors, and operating systems.

- *Reading/modifying physical memory.* Modern computers are shipped with omnipotent firmware, invisible even to operating systems and hypervisors. This firmware (system management mode and BIOS/UEFI) is a frequent target of attacks. Such compromised firmware may read and modify all physical memory of a computer. Similarly, OSes and VMMs (or hypervisors) are privileged software that have full access to both virtual and physical memory of applications. Thus, a compromised OS or VMM may read and overwrite any application data. Intel SGX helps protect against system software's attempts to read or modify enclave data because SGX strives to protect all enclave data leaving the CPU package with encryption. In addition, at the software level, access to EPC contents is allowed only for the corresponding enclave; SGX disallows access to EPC for any other software.
- *Address translation attacks*. Malicious operating systems and VMMs may launch more insidious attacks on the enclave by tampering with page table mappings. Recall that the OS is responsible for the virtual-to-physical-pages mapping of application pages (this mapping is removed when a physical page is evicted from RAM to secondary storage and recreated when the physical page is swapped back into RAM). The malicious OS could silently jumble up these mappings and subvert enclave's execution flow. Intel SGX introduces a special mechanism to verify that the OS maintains correct page tables, therefore preventing this indirect attack.
- *Replay attacks*. As the attacker cannot modify enclave data due to SGX encryption, she can try to feed the enclave its own old data in the hope of subverting the enclave's execution. For example, the attacker might replace intermediary statistics for the aggregate association analysis with previously

computed stale one and the enclave would then compute misleading results. To prevent this attack, the integrity-protection mechanism of SGX generates unique version numbers and verifies them when bringing enclave data back to the CPU (thus achieving data freshness).

Host application attacks. The host application that runs the enclave can be compromised by the attacker to run arbitrary malicious logic. The possible attacks on the enclave, in this case, are the control flow attacks.

- *Control flow attacks*. The malicious host application can try to subvert enclave execution by jumping not to the beginning of the requested enclave entry, but to some arbitrary place in enclave code. This could lead the enclave to "forget" to perform some computations. Intel SGX helps prevent such random jumps, allowing to enter the enclave only at a predefined set of entry points.

Attacks by malicious actors. All the earlier attacks can be launched by an insider operator who has full access to both hardware and software running on the physical machine. An additional attack vector is impersonating the SGX enclave on an attacker-controlled machine and directing end users to it.

- *Malicious operator/administrator.* A malicious operator or administrator has physical access to the machine where the user enclave runs and has complete control over its privileged software. Thus, the operator can launch any of the aforementioned attacks. As we discussed earlier, however, all these attacks are undermined by Intel SGX.
- *Impersonation attack*. A malicious remote party may pretend to run a secure enclave on a genuine Intel SGX-enabled machine, while in reality it would emulate enclave execution and apply all attacks described earlier. However, the remote attestation procedure of Intel SGX ties the identity of the server and the enclave to the unique secret known only to the SGX-enabled CPU. Thus, for a successful impersonation attack, the hacker needs to obtain/guess the secret that is practically impossible.

Although current Intel SGX implementations help prevent a wide range of malicious behaviors, they are still susceptible to the following attacks:

System software attacks. Although Intel SGX protects against most of the attacks launched by privileged system software, some attacks are out of the scope of the HW protection. Enclave developers and users must be aware of these attacks and harden SGX applications as deemed appropriate.

- *Denial-of-service attacks*. As all privileged software is not in the TCB, the server can simply refuse to run the enclave or pause it frequently. This would lead to deteriorated QoS guarantees or complete denial of service. This attack is out of SGX scope, but we note that it is also not in the interest of a service provider.
- *Iago attacks*. Any SGX enclave needs to communicate with the outside world. At the very least, the enclave relies on network I/O from the underlying operating system to receive requests from the remote user and send his replies back.

This is achieved by ECALLs/OCALLs with specific arguments. However, the untrusted host application/OS can tamper with arguments and return values of these calls, which in some cases may subvert enclave execution [17]. Enclave developers must design the ECALL/OCALL interface carefully and double-check all potentially malicious return values to thwart Iago attacks.

- *Side-channel attacks*. Side-channel attacks are attacks that do not access sensitive data directly but can infer it to some degree by observing changes in the microarchitectural state. Powerful side-channel attacks such as cache attacks [18], timing attacks [19], Rowhammer DRAM attacks [20] are especially threatening in the SGX context as they leak enclave-protected data. Intel SGX is also susceptible to a specific type of side-channel attack called controlled-channel attacks: a malicious operating system can induce page faults on every page accessed by the enclave and observe enclave memory accesses at page granularity [21]. Intel CPUs (including SGX-enabled CPUs) provide microcode updates to patch Intel hardware against some of these attacks. Various software techniques help mitigate these attacks.

Attacks on enclaves. Finally, the SGX technology provides no security guarantees regarding code running inside the enclaves themselves. Thus, attackers can still launch attacks on buggy software executing inside an SGX enclave.

- *Vulnerabilities in enclave code*. Intel SGX helps protect the enclave from the malicious outside world, but it cannot protect a malicious enclave from itself. If the enclave code contains bugs that can be exploited by an attacker, SGX provides no confidentiality and integrity guarantees. Enclave developers must rely on well-established techniques for software reliability such as code reviews, formal verification, extensive testing, dynamic memory safety [22], and CFI/CET [23].

3.4 Performance properties

Thanks to the fact that Intel SGX executes native enclave code directly on a CPU using plaintext enclave data in CPU caches, it significantly outperforms software-only cryptographic schemes such as fully homomorphic encryption, garbled circuits, and MPC. For example, the PRINCESS [24] privacy-preserving genomic data analysis framework based on Intel SGX performs orders of magnitude better than homomorphic encryption and garbled circuit solutions.

Improving confidentiality and integrity of applications always comes at a cost of performance degradation. In the case of Intel SGX, there are three main sources of performance overhead.

First, additional hardware checks on memory accesses inside EPC, as well as transparent encryption, authentication, and replay protection of EPC pages inevitably degrade performance. Note that cache-hit accesses incur no performance penalty because they do not propagate outside of the CPU package, while cache-misses are already rather expensive and thus amortize the overhead of EPC accesses.

The second source of overhead is transitioning between the enclave and the host application. Entering and exiting the enclave is notoriously slow, and frequent ECALLs/OCALLs can lead to high overheads. However, this problem can be solved by a switchless design where the enclave never exits and runs in parallel to the host application [25]. In detail, the enclave runs on one CPU core and communicates with the untrusted application running on another CPU core via a shared queue where both store ECALL/OCALL requests and responses. This switchless design can be enabled in most SGX tools including SGX SDK [25], Graphene-SGX [16], SCONE [14], and SGX-LKL [15].

The third source of overhead stems from a limited size of EPC. If the working set of the enclave exceeds the size of the EPC, a slow EPC paging mechanism is employed. This can be a serious performance bottleneck for genomics applications that operate on huge datasets. To alleviate this issue, the developer must employ locality-aware techniques to operate on small chunks of data at a time (similar to cache blocking).

4. HW-MPC and SGX in cloud

Bioinformatics today tackles problems that are infeasible to compute on a single machine. Therefore, bioinformatics applications require cost-effective scaling where thousands of (virtual) machines are employed to collaboratively compute over application data. The *cloud computing* paradigm emerged to solve the problem of agile and cost-efficient scalability. With cloud computing, a large fleet of virtual machines can be acquired in a short time and decommissioned after the problem at hand is solved.

Unfortunately, a user who wants to offload computations to the cloud must trust the cloud service provider. Numerous attacks on cloud platforms raise privacy concerns over offloaded user data [26]. Trusted hardware architectures such as MPC or TEEs bring back control over private computations in untrusted clouds to the end user. Among the trusted execution environments surveyed earlier, Intel SGX is unique in the security properties, versatility, and performance it provides. Due to its high performance and minimal TCB comprising only the CPU chip and the enclave, Intel SGX is a perfect match for untrusted cloud environments. Even a cloud administrator with physical access to a cloud server cannot deduce what code and data are running inside an enclave, whereas this remains a threat with cloud providers that do not offer Intel SGX.

To make Intel SGX in the cloud a viable option, it must integrate with existing technologies for scalable and distributed data processing. Established big data processing frameworks like Spark and MapReduce are indispensable to the bioinformatics practitioner. Keeping familiar abstractions and being able to reuse existing code with Intel SGX is important for its adoption. In the following, we will present how Intel SGX can be successfully deployed in the cloud, how it integrates with existing cloud technologies and briefly touch on open challenges.

4.1 TEEs in the cloud

Because of security concerns when moving computation to the cloud, it is only sensible to use TEEs to protect cloud applications. Although cloud providers go to great lengths to secure their infrastructure, TEEs offer an additional layer of security that the application developer controls exclusively. TEEs provide a secure environment for confidential computing in an otherwise untrusted public cloud. Even the proverbial disgruntled employee with physical access to the servers cannot extract customer data, for example, a genome that is currently analyzed, from within a TEE.

Nevertheless, the cloud provider retains full control over the cloud infrastructure. It is still responsible for allocating resources and running the associated services reliably. The rich ecosystem of integrated cloud services (database, key-value store, virtual network, coordination service, domain name service, message queue, load balancer, etc.) significantly reduces the operational overhead of deploying applications in the cloud. As genomic data processing routinely requires to analyze terabytes of data, it is a perfect fit for the cloud where computational resources scale conveniently with the size of the problem.

As Intel SGX-enabled processors were released in 2015, cloud providers have begun to integrate Intel SGX into their service offerings [27,28]. Due to SGX's novelty, we expect the integration into the existing cloud ecosystem to evolve over time. Now that different TEE implementations from multiple hardware vendors are available, cloud providers might be inclined to offer a unifying programming interface to abstract the details of the individual TEEs. We may also see software-only implementations of existing hardware TEEs that can be used as a fall back if the actual hardware might be unavailable. Cloud providers will continue to innovate in that space and it will be interesting to see how the integration of TEEs into cloud platforms evolves over time.

Even though SGX provides a strong trust anchor, the cloud user may still have to trust the provider to some degree with other aspects. For example, if it is important for the data to stay within certain geographic boundaries, the cloud user must trust the cloud provider to correctly implement "geo-fencing." SGX does not offer any mechanisms to enforce this. Today, the tier-one cloud providers typically have a global presence. Cloud users decide at which point-of-presence to instantiate a workload. As there is no way to verify the actual location of the virtual resources, the cloud user must trust the provider. Similarly, to truly leverage the power of the cloud, the cloud user might have to trust the cloud provider with other tasks too. Cloud architects are just beginning to reason about the security implications and trade-offs of TEEs in the cloud.

4.2 Developing with SGX in the cloud

Developing SGX applications to run in the cloud is similar to developing regular SGX applications in many aspects. The application developer splits the application in a trusted and untrusted part, defines the interfaces at the trust boundary, and follows established practices when developing security-sensitive SGX applications.

However, in other aspects, the cloud is also a very different environment from executing a single SGX application on a local machine.

For one, as multiple cloud applications from different cloud users share the same infrastructure, they must be isolated from each other. This is typically done using virtualization, possibly in combination with containers. Although SGX was designed with virtualization in mind, the hypervisor must support it. Patches exist for the popular open-source Xen and KVM hypervisors [29], and cloud providers are adding virtualized SGX support to their own hypervisors. Supporting SGX inside containers is less problematic as containers share the underlying host operating system. The container only needs access to the SGX device and be able to communicate with SGX daemons running on the host. If this is set up correctly, SGX applications can be launched inside containers without problems.

Because containers are a popular vehicle to package and deploy cloud applications, some initial work exists on automatically converting an existing container into an "SGX container" [30]. Static analysis of the container reveals the main binary and its runtime dependencies. With the help of Graphene-SGX [16], which allows to run existing binaries transparently on SGX, the main binary is hoisted into an SGX enclave. A new container image is constructed in which Graphene-SGX replaces the existing main binary as the entry point. Ultimately, if the resulting "SGX container" image is instantiated, the main binary will automatically run inside an SGX enclave. In this way, existing genomics workflows, including WDL-based execution pipelines [31], maybe converted with low effort to run on SGX.

Running an existing program on SGX using one of the mentioned solutions has another benefit. The solutions work by wrapping the unmodified program and mediating all the actions that are not permitted in an enclave. As SGX introduces a new processor mode, certain operations typically available in user mode are disallowed in enclave mode. The wrapping code can work around some of these restrictions. The wrapping code can also transparently encrypt any data leaving the enclave. As existing applications trust the environment, they typically output clear text data, either by writing it to a file or sending it over the network. The wrapping code can thus act as a "shield": it encrypts all data exiting the enclave and decrypts all incoming data. Besides transparent encryption, the wrapping code can enact a variety of security policies to retrofit security onto otherwise security-oblivious applications.

In summary, developing SGX applications for the cloud is similar to developing stand-alone SGX applications. However, virtualization, packaging, deployment and the desire to preserve existing workflows create additional challenges that must be overcome. The technical building blocks exist today, but it will take additional time for them to mature before cloud providers will want to integrate them into their infrastructure.

4.3 Deploying SGX applications

As already mentioned earlier, cloud applications tend to be a distributed collection of cooperating services rather than a single monolithic application. To help with the

instantiation and orchestration of multiple cooperating services, cloud users write a "recipe" that states which services should be launched together. The recipe includes information on service dependencies (e.g., the database must be up before application server), their communication topology (e.g., the application server can talk to the database) and configuration information for each service. An orchestrator utility takes the recipe and starts all the listed services. The orchestrator also configures the (virtual) network to enable their communication and restarts services if they crash. Examples of these orchestrators include Docker Swarm [32], Kubernetes [33] and cloud-platform specific implementations like AWS CloudFormation Templates [34], Google Cloud Composer [35], and Microsoft Azure Automation [36].

For confidential computing in public clouds, it would be highly desirable to have a "secure" version of this orchestrator. The developer provides an existing recipe to the secure orchestrator. The secure orchestrator takes the existing applications and wraps them inside SGX enclaves; similar to the automatic conversion of containers into "SGX containers." The secure orchestrator uses the communication topology from the recipe and installs an equivalent security policy. The security policy is enforced at the enclave boundary where the wrapping code interacts with the untrusted outside world. For example, if the recipe states that the database can communicate with the application server, the policy would disable all network communication by default and only allow these two services to establish a network connection.

Besides restricting communication to known services listed in the recipe, the in-flight data must also be secured. To protect the in-flight data, the SGX-enabled application establishes an attested and authenticated communication channel between the participating endpoints. Attestation ensures that the enclave is, in fact, a genuine enclave running on an up-to-date SGX-enabled platform. The attestation serves to authenticate the enclave to interested parties. With RA-TLS [12] there already exists a solution to establish an attested secure channel between two SGX enclaves. RA-TLS uses the standard Transport Layer Security (TLS) protocol and integrates SGX remote attestation into it. RA-TLS solves the problem of secure interactive interenclave communication. RA-TLS can be used not only to send secrets, such as encryption keys, to the enclave but to protect communication between cooperating enclaves in general. For example, RA-TLS can secure the communication of a multistage WDL-based workflow. In this scenario, RA-TLS ensures that only the correct next stage (as measured through MRENCLAVE/MRSIGNER) is able to receive the input data from the previous stage.

Besides the ability to run custom applications in the cloud, providers also host an ever-increasing set of common processing frameworks. Map/Reduce, Function-as-a-Service (FaaS), stream processing (e.g., Spark), and data flow-based frameworks are sufficiently popular among cloud users for the provider to offer them as hosted solutions. Researchers have already started to integrate Intel SGX with these existing programming models to make confidential computing easily accessible to a wider class of users and application domains. For example, Microsoft Research ported Map/Reduce-style computations to Intel SGX and called the resulting system VC3 [37]. Similarly, Opaque [38] is an SGX-enabled Spark engine developed by

the University of California, Berkeley. Opaque not only helps protect the confidentiality and integrity of the data but also obfuscates the communication patterns between components to prevent an attacker from learning sensitive information via side channels. The result is an oblivious database system where the query execution hides information about the distribution of the underlying data.

Unfortunately, current SGX-enabled data processing frameworks (e.g., both VC3 and Opaque) require to rewrite the code put inside the enclave in C/C++. Rewriting a large code from a high-level language like Python or Java into low-level C code may be impractical; in addition, genomics scientists may not have skills for such a venture. For wide adoption, it is imperative that SGX enclaves allow to run logic written in common data-scientist languages. In the future we expect more and more processing frameworks and language runtimes to integrate natively with trusted execution technologies. Cloud providers will then be able to offer secure, SGX-enabled versions of popular data processing frameworks.

4.4 Open questions on SGX in the cloud

Although most aspects of the current SGX ecosystem are easily transferable to the cloud, some specific concepts do not translate as well. We briefly highlight some facets of SGX that must still be aligned with the cloud's operational model. The topics mentioned here require further exploration and are areas of active research.

Sealing. Intel SGX helps protect the integrity and confidentiality of a program at runtime. For a complete solution, a program's inputs and outputs must also be protected at rest. To this end, SGX supports *data sealing*. Before storing data, the enclave encrypts it with a key only available on this particular platform. In this way, even if an attacker manages to exfiltrate the sealed data, the attacker will be unable to decrypt it, as instantiating the same enclave on a different platform will result in a different sealing key. However, sealing to the platform is of limited use in the cloud, as the infrastructure is virtualized. When an enclave is restarted, there is no guarantee that it will come up on the same physical server as before. Hence, protecting data from exfiltration and the ability to only be decrypted by authorized enclaves requires a different approach.

One option is to deploy a key management enclave. Application enclaves remotely attest to the key management enclave to access their decryption keys. Cloud service providers already offer key management services. They could either run their existing service unmodified on SGX, for example, Barbican [39] or develop a new key management service with a small TCB from scratch.

In general, for a distributed scalable SGX application like a key manager itself, instead of sealing the secrets to enclave, SGX sealing can be used to seal a shared master key. This key gets provisioned during instance creation. Each instance of the SGX application then seals its secrets using the shared master key.

Deep attestation. Sealing also involves the notion of enclave identity and deciding which enclave should be able to derive specific encryption keys. We already touched upon the topic of cloud applications often being distributed. Although an enclave's identity is defined through a cryptographic hash measurement of its code, a cloud

application includes multiple SGX enclaves potentially with different identities/code measurements. Attestation is used by a relying party to decide on whether to trust an enclave. In the cloud, the relying party is deciding on whether to trust an ensemble of cooperating SGX enclaves. One possibility to achieve a deep attestation is to have the secure orchestrator issue a "cloud service attestation." This attestation is essentially a signed statement by the orchestrator that it faithfully instantiated and successfully attested each component. A client of the cloud service can inspect the "cloud service attestation" to determine for themselves the trustworthiness of the cloud service. In the genomics domain the aggregate service may be an entire processing pipeline where each pipeline step is a single SGX enclave. To gain trust in the entire pipeline, all stages must be attested. A potential solution must trade off the complexity and frequency of attestations as well as the desire to hide the cloud service's implementation details, among other things.

4.5 Putting it all together

Coming back to our example of aggregate association analysis, the application description given earlier was overly simplistic: instead of a single simple application, the analysis is more likely a workflow of chained transformation steps, for example, using the GATK [1], written in WDL [31], a language allowing to describe workflow graphs of steps running in containers. Additionally, analysts would not directly launch the application but use runtimes such as Cromwell [31] to dispatch and schedule the workflow in a cloud environment.

Fig. 8.12 illustrates how we could compute such a workflow in an untrusted cloud. Individual steps $f_x()$ will run as-is inside earlier-discussed SGX containers. A small shim in the container enforces that communication between steps is

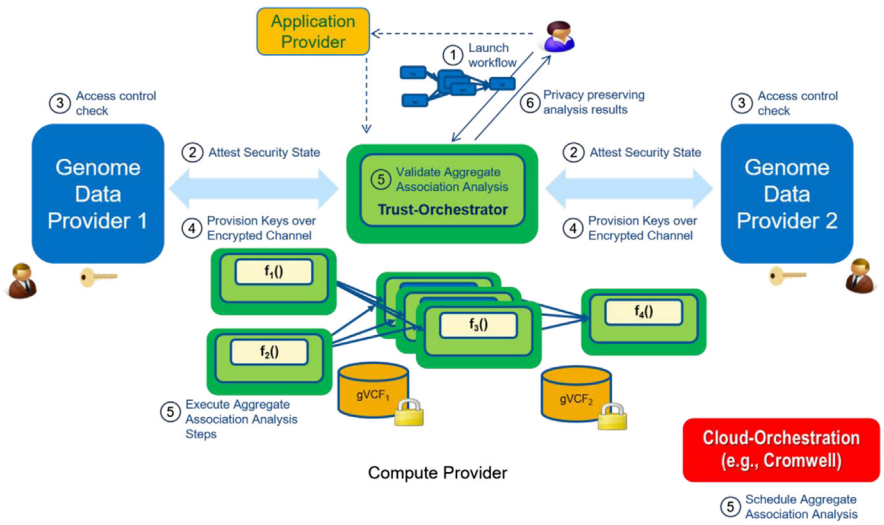

FIGURE 8.12

Example of application in Cloud.

properly secured using RA-TLS. The shim also handles the application-transparent encryption and decryption of data files. A separate new component, dubbed trust-orchestrator, proxies between all involved SGX enclaves and the different parties, whereby providing them with implicit deep attestation over the whole workflow. This simplifies the interaction and hides unnecessary details of the cloud topology from external parties. In conjunction with the shim in the container, the trust-orchestrator handles key management and, importantly, also enforces the integrity of the workflow graph while still leaving the scheduling decisions to the existing untrusted scheduler. This reduces the TCB size while also allowing the cloud operator complete control to optimize overall cloud resources.

5. Outlook

Intel SGX is available on current generation Intel client platforms as well as low end servers. There is lots of traction in the academic community with over 170 papers published on SGX over last 3 years. On the client side, the usages include password managers [40], secure web browsers [41] and SGX-enabled DRM [42]. Cloud computing is where a lot of the academic work is focused and usages include Genomics privacy-preserving analytics [24], distributed ledger [43], encrypted databases [44], middle boxes [45], and many more.

In addition to academic traction, there are a number of startups sprouting that are productizing SGX-based HSMs [46], key managers [39], network middle boxes [47] and more. There is an emerging trend called decentralized cloud computing [48] that relies on SGX TEE for running arbitrary computations on any computer participating in decentralized cloud computing. Microsoft Azure recently announced Azure cloud computing deploying Intel SGX as well as a software SDK called Open Enclave SDK [27]. Google Cloud Platform has announced Asylo [28], an open-source framework for confidential computing. Asylo allows development of applications at a higher-level abstraction and supports various backend like Intel SGX and AMD SEV. With various SDKs, cloud users have tools to build their secure applications, but still there is a strong desire to run unmodified legacy cloud workloads on SGX. There are a number of academic/production projects like SCONE [14], SGX-LKL [15], and Graphene-SGX [16], which provide support for running unmodified legacy applications inside an enclave. Going forward these environments will provide automatic protection of complex workloads using SGX, but users still have the flexibility to partitioning workloads that are sensitive to the size of TCB.

Even though there is clear traction in the industry and among the cloud providers and a clear path forward for applications such as secure outsourcing, there are still numerous technical challenges to the widespread adoption of TEEs. Adopting TEEs in multiparty settings, in particular for sensitive and highly regulated data such as health information, also raises additional nontechnical challenges: as any new technology, there is legal and regulatory uncertainty, for example, whether absolute or relative identifiability is required in the EU has a large impact on whether MPC,

based on either hardware or crypto, is sufficient or not [49]. Additionally, a public acceptance that is based as much on technical aspects as it is on perception has yet to be tested.

Nevertheless, we consider TEEs and HW-MPC a very promising avenue to allow the pooling of sensitive data and unleashing of the vast potential of collaborative analysis.

References

[1] https://software.broadinstitute.org/gatk/.
[2] https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2677.
[3] https://trustedcomputinggroup.org/work-groups/trusted-platform-module/.
[4] https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-infrastructure-overview.html.
[5] https://keystone-enclave.org/.
[6] https://developer.arm.com/technologies/trustzone.
[7] https://globalplatform.org/.
[8] https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/.
[9] https://software.intel.com/en-us/sgx.
[10] Johnson, et al. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. Intel Whitepaper; 2016. https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.
[11] Scarlata, et al. Supporting Third Party Attestation for Intel® Software Guard Extensions Data Center Attestation Primitives. Intel Whitepaper; 2018. https://software.intel.com/en-us/download/supporting-third-party-attestation-for-intel-sgx-data-center-attestation-primitives.
[12] https://github.com/cloud-security-research/sgx-ra-tls.
[13] https://developers.google.com/protocol-buffers/.
[14] Arnautov S, Bohdan Trach, Gregor F, Knauth T, Martin A, Priebe C, Lind J, Muthukumaran D, O'Keeffe D, Stillwell ML, Goltzsche D, Eyers D, Kapitza R, Peter P, Fetzer C. SCONE: secure Linux containers with Intel® SGX. OSDI; 2016.
[15] https://github.com/lsds/sgx-lkl.
[16] Tsai C-C, Vij M, Porter D. Graphene-SGX: a practical library OS for unmodified applications on SGX. USENIX ATC; 2017.
[17] Stephen C, Shacham H. Iago attacks: why the system call API is a bad untrusted RPC interface. ASPLOS; 2013.
[18] Wang W, Chen G, Pan X, Zhang Y, Wang XF, Bindschaedler V, Tang H, Gunter CA. Leaky cauldron on the dark land: understanding memory side-channel hazards in SGX. CCS; 2017.
[19] Van Bulck J, Frank P, Strackx R. Nemesis: studying microarchitectural timing leaks in rudimentary CPU interrupt logic. CCS; 2018.
[20] Jang Y, Lee J, Lee S, Kim T. SGX-Bomb: locking down the processor via Rowhammer Attack. SysTEX; 2017.
[21] Xu Y, Cui W, Marcus P. Controlled-channel attacks: deterministic side channels for untrusted operating systems. S&P; 2015.

[22] Szekeres L, Payer M, Tao W, Song D. SoK: eternal war in memory. SP. 2013.
[23] Ruan de Clercq, Verbauwhede I. A survey of hardware-based control flow integrity (CFI). arXiv; 2017.
[24] Chen F, et al. PRINCESS: privacy-protecting rare disease international network collaboration via encryption through software guard extensions. Bioinfornatics 2017;33(6): 871−8.
[25] https://github.com/intel/linux-sgx.
[26] https://www.hackread.com/amazon-suffers-security-breach/.
[27] https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.
[28] https://cloudplatform.googleblog.com/2018/05/Introducing-Asylo-an-open-source-framework-for-confidential-computing.html.
[29] https://01.org/intel-software-guard-extensions/sgx-virtualization.
[30] https://github.com/cloud-security-research/graphene-sgx-secure-container.
[31] https://software.broadinstitute.org/wdl/.
[32] https://docs.docker.com/engine/swarm/.
[33] https://kubernetes.io/.
[34] https://aws.amazon.com/cloudformation/aws-cloudformation-templates/.
[35] https://cloud.google.com/composer/.
[36] https://azure.microsoft.com/en-us/services/automation/.
[37] Schuster, et al. VC3: trustworthy data analytics in the cloud using SGX. In: IEEE symposium on security and privacy; 2015.
[38] Zheng, et al. Opaque: a data analytics platform with strong security. NSDI; 2017.
[39] Chakrabarti S, Baker B, Vij M. Intel® SGX enabled key manager service with OpenStack Barbican. arXiv; 2017. https://arxiv.org/abs/1712.07694.
[40] https://pdfs.semanticscholar.org/ec40/833215b3d415c9525940690d0a94d2a178ca.pdf.
[41] https://software.intel.com/en-us/articles/hardening-authentication-tokens-in-browsers-using-intel-software-guard-extensions.
[42] https://software.intel.com/en-us/articles/using-innovative-instructions-to-create-trustworthy-software-solutions.
[43] Brandenburger M, Cachin C, Kapitza R, Sorniotti A. Blockchain and trusted computing: problems, pitfalls, and a solution for hyperledger fabric. ArXiv; 2018. https://arxiv.org/abs/1805.08541.
[44] Priebe C, Vaswani K, Costa M. EnclaveDB − a secure database using SGX. S&P. 2018.
[45] Poddar R, Chang L, Popa RA, Ratnasamy S. SafeBricks: shielding network functions in the cloud. NSDI; 2018.
[46] https://fortanix.com/.
[47] https://arxiv.org/pdf/1706.06261.pdf.
[48] https://enigma.co/enigma_full.pdf.
[49] Damiani E, editor. Evaluation and integration and final report on legal aspects of data protection; 2016. Deliverable D31.3, EU Project PRACTICE (Privacy-Preserving Computation in the Cloud).