[

image

1920×1043 135 KB

](https://collective.flashbots.net/uploads/default/original/2X/b/bacaace03c01bc2e10fe437d2827ca195caa3aea.jpeg)

At Flashbots we have been experimenting and building products utilizing Intel Software Guard Extensions (SGX) as described in Running Geth within SGX, Block Building inside SGX, and most recently Sirrah: Speedrunning a TEE Coprocessor.

This post is one of a series of upcoming posts dedicated to confidential computing with TEEs. Here we will provide an introduction to Intel SGX that includes use cases, benefits, and limitations. Note: The content will be a mix of technical and high-level explanations of the workflow. A more in-depth and hands-on session will be conducted at 4pm CET Wednesday (15th May)

(Youtube, Zoom).

[

](https://www.youtube.com/watch?v=TVmeuJ_HgYg)

What are TEEs?

Cloud providers offer a range of benefits to their customers such as scalability, performance, geographic distribution, and flexible maintainability costs. However, when it comes to security-critical application that handles sensitive data like personal and financial data, users would need to trust their cloud provider to be benign and act honestly. In response, cloud providers are expanding their offering with enhanced security features through Trusted Execution Environments.

T

rusted E

xecution E

nvironment (TEE) is a hardware-based security technology that protects sensitive data and applications' execution from a malicious host or hypervisor. It protects the code execution and data at rest, transit and during

execution. Moreover, TEEs provide an attestation mechanisms to ensure the authenticity of the hardware and the correctness of the running application. Currently, there are several TEE offerings from different vendors such as Intel, AMD, ARM and AWS. In this post, we will focus on Intel's S

oftware G

uard E

xtensions (SGX) because it is one of the first TEEs that became widely available with a complete set of security features (sealing, attestation, confidentiality and integrity protection) and also because we have experimented a lot of designing and building products with it.

Intel SGX

In 2015, Intel extended the x86 instruction set to support their new hardware security feature, S

oftware G

uard E

xtensions (SGX). It provides confidentiality and integrity protection to the execution of code and data inside isolated and encrypted memory regions called Enclaves

.

These enclaves reside inside a processor reserved memory area, called the E

nclave P

age C

ache (EPC). Additional details on EPC will be presented during the live session next week.

SGX's trust model is very strict and consists of only the CPU, its firmware and the enclave code. This is referred to as the T

rusted C

omputing B

ase (TCB). Everything else is considered untrusted and is protected against, including the admin, the operating system, privileged processes or any co-located applications in the system.

[

image

1145×1008 60.1 KB

](https://collective.flashbots.net/uploads/default/original/2X/4/4d204106fc2da2d0eb4c64471602d6b37d31c795.png)

Intel also provides a SGX SDK with a set of tools to improve the SGX application development experience. However, it still requires a specific architectural approach which will be described in the technical section below.

Intel's SGX SDK introduces a new notion to enter and exit the enclave through the wrapper functions ecall

and ocall

. From the perspective of the developer, they seem like normal function calls. However, they actually have a longer flow to ensure the security during the transition into and outside of the enclave.

Ecall execution flow

[

image

2131×867 78.8 KB

](https://collective.flashbots.net/uploads/default/original/2X/f/ff8f40bba0624a838f59e6909469c3b420d5db3e.png)

The figure illustrates the flow of an ecall function that resides in the enclave.

When the untrusted side (App) calls ecall_critical_func()

, it first executes wrapper code (enclave_u

) that is generated by the sgx_edgr8r

tool. This wrapper code does the args marshalling and pointer security checks. Then it calls sgx_ecall()

that is provided from the sgx untrusted runtime system (libsgx_urts.so

) that does the transitioning into the enclave mode.

Afterwards, the flow is passed to the sgx trusted runtime system (libsgx_trts.so

), which in turn forwards the flow to generated wrapper code (enclave_t

) that does the args unmarshalling and finally executes the intended ecall_critical_func()

inside the enclave.

This execution flow and context switch have performance ↔ security trade-offs as several researchers have shown (hotcalls, sgx-perf).

The ocall execution flow works very similarly but in the opposite direction and has some intermediate steps, such as clearing the translation lookaside buffers and storing CPU state in the Save State Area (SSA), before transitioning to the untrusted side.

Technical steps to build SGX APP with SGX SDK

Here are some recommended steps when you develop an application with the Intel SGX SDK.

1. Separation of concerns:

You need to separate your code into 2 parts, untrusted (App) and trusted (Enclave).

Then define the interface that transition the call flow in- and outside the enclave by writing the enclave.edl file.

enclave { / * *ecall_critical_func - invokes ECALL to do the transition into the enclave * and executes the security sensitive code.* / trusted { public void ecall_critical_func(void);
}; / * *ocall_print_string - invokes OCALL to display string buffer inside the enclave. * [in]: copy the string buffer to App outside. * [string]: specifies 'str' is a NULL terminated buffer.* / untrusted { void ocall_print_logs([in, string] const char *str); }; };

1. Enclave configuration:

Define the metadata that your enclave should have. More accurately, enclave's maximum heap size, amount of threads and their stack size, and write them inside the enclave.config.xml file. Below is an example of an enclave configuration:

0

0

0x40000

0x100000 0x1000

0x40000 10

10 1

0

1 0xFFFFFFFE

1. Wrapper code generation:

Once finished implementing the application and the enclave logic, you need to call the sgx_edgr8r

tool to generate the wrapper code enclave_u

and enclave_t

files, which are responsible for marshalling the args and calling the sgx_ecall

and sgx_ocall

, respectively.

Those calls are provided by the libsgx_trts.so

and libsgx_urts.so,

which handle the security checks when transitioning in and out of the trusted enclave.

1. Finally, you need to build the enclave and the untrusted app side separately. The latter will load the enclave binary on startup and then can utilize the aforementioned generated wrapper code to call inside the enclave. Here are some code samples to start with [1] [2].

For more in-depth details such as the supported data types, enclave.edl file syntax, enclave configuration settings and much more can be found in the SGX SDK developer guide .

Remote attestation

Intel provides a remote attestation as a way to prove the integrity of the SGX application execution on the cloud. In other words, it ensures the user that the application is running on an authentic trusted SGX hardware, running the correct code and processing the sensitive data securely.

There are two types of remote attestations, E

nhanced P

rivacy Id

entifier (EPID) and D

ata C

enter A

ttestation P

rimitives (DCAP). The former relies on Intel's infrastructure for launching enclaves, while the latter came to support the deployment of enclaves without relying on Intel's infrastructure.

We will briefly explain the workflow and differences between the two below.

[

image

2312×489 35.4 KB

](https://collective.flashbots.net/uploads/default/original/2X/c/c292ce334f92b37b54487ff2d8e444b948577ff9.png)

Alice wants to start a secure communication channel with the SGX App and requests remote attestation first to verify its integrity. The SGX app creates an SGX report that includes metadata, such as the enclave measurement (MRENCLAVE), and sends it to the quoting enclave to generate an attested SGX quote. The quoting enclave uses the EPID key that was provisioned to it at the initial deployment of the SGX machine by Intel Provisioning Service (PCS). Afterwards, the signed quote is sent back to Alice which she forwards to I

ntel's A

ttestation S

ervice (IAS) for verification. Based on the returned result, Alice could decide to proceed and initiate a secure communication channel with the SGX app.

[

image

2795×884 178 KB

](https://collective.flashbots.net/uploads/default/original/2X/c/cb06c147a0d69bc46ccb38ca2c0cff305b9e33fb.png)

DCAP processes are similar at the start but instead of using EPID key to sign the attestation, it uses P

ublic K

ey I

nfrastructure (PKI) and X.509 certificate chains instead. In essence, upon first deployment of the SGX machine, the Provisioning Certificate Enclave (PCE) fetches the attestation certificates and revocation list from another service, Intel Provisioning Certification Service. The quoting enclave would be talking with the PCE instead.

Alice doesn't need to consult the IAS anymore but rather she periodically fetches the DCAP certificates and revocation lists and caches them locally. Subsequently, when she receives the quote, she can compare the embedded certificates with the ones she cached and verify directly.

The explanation and figures are inspired from Intel's DCAP paper and gramine attestation docs.

Use cases

As you might have built a mental image so far about possible scenarios where TEEs are relevant, here are some examples:

- Storing and processing of sensitive data, such as financial & health institutes or communication privacy (likeSignal)
- In blockchain and cryptocurrency, e.g., keeping the bidding auction safe from tampering as well as not leaking sensitive information.

Also, it could potentially help improve consensus protocols (e.g. BFT protocolsSplitBFT, Hybster) since trusted hardware might potentially decrease some intermediary steps or nodes that were introduced for security reasons.

(You can ask yourself the question do we need relay nodes if builders and proposers are both running in TEEs ? maybe not ?)

- In general for cases where integrity is very important to ensure that the state wasn't tampered with through remote attestation. For examples use your imagination
- Check out some of our experiments withRunning Geth within SGX & Block Building inside SGX

Limitations

Now, you would have probably built your opinion about possible drawbacks and limitations that SGX might have, here are some of the hot ones:

- Performance: it depends mostly on the workload and the application's architecture.

For example, too frequent IO operations will impact the performance due to the expensive context switch we discussed above. In case of interest, I recommend watching this video.

- Memory constraints: SGX enclave are by design limited by size. Only recently (2021), Intel releasedScalable SGX and increased the enclave size limitations from 128Mb up to 512Gb.

However, for applications that require enclave memory bigger than those limits, then a secure paging process will trigger. This process extremely impacts the performance which might be critical for timely sensitive applications.

- Code refactoring: as you saw earlier, developing a SGX application with intel SGX SDK requires specific architecture in mind and most probably code refactoring for existing code that needs to be ported into SGX. Porting efforts might be very high depending on the requirements and the structure of the vanilla code.

However, there are alternatives that you can utilize to deploy your legacy code directly in an SGX enclave. For example, library operating systems (like gramine) which we will talk more about below.

- Side-channel attacks: researchers have conducted heavy analysis and sophisticated attacks to compromise the security guarantees of Intel SGX. Some of those attacks like spectre and foreshadow have been patched by Intel through microcode updates which consequently impacted the performance (sgx-perf). Other researchers, have done access pattern, reply attacks and other software(e.g. SGAxe, Plundervolt) and hardware attacks (VoltPillager).

This could be a lengthy topic on its own and is out-of-the-scope of this blog post. By interest, it could have its own blog post and discussions around it. Nonetheless, just wanted to mention it for transparency.

Gramine (libOS) [

](https://gramineproject.io/)

What is a libOS?

A library operating system (libOS) is a lightweight operating system that provides application-level abstractions without the need for a full operating system kernel.

Gramine is an example of a libOS that abstracts away most

of the system calls that an application would require to run within its minimal runtime.

[

image

911×865 54.1 KB

](https://collective.flashbots.net/uploads/default/original/2X/a/ad716f551e3409904534c842e21809e6549fbd03.png)

This way, it enables users to deploy their whole application inside an enclave with very minimal efforts instead of refactoring it manually. As we will see in the hands-on live session, one would have to write a separate gramine manifest to describe the app's requirements, such as enclave size, amount of threads, access to paths, etc…

This flexibility comes at the cost of an increased TCB size and potentially

less security due to a wider attack surface. Furthermore, despite it hassupport to many system calls (~170 out of 360). It is still limited in case your application uses any of the unsupported or partially supported features.

Keep in mind that gramine is continuously under development and incrementally increasing their supported features.

At Flashbots, we have also experimented with gramine. For example,Sirrah smart contracts demos are executed in SGX kettles using gramine.

TLDR:

- TEEs provide confidentially and integrity guarantees for data at rest, transit and during execution in untrusted

environments.

- Intel SGX is an example of such TEE, with benefits as well as limitations
- Gramine provides an alternative way to utilize SGX with minimal efforts