TYCHE: Trusted Boot

Maëlys Billon
maelys.billon@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

In this study, we aimed to examine the measuring launch environment (MLE) process in Intel trusted execution technology (TXT) in order to implement a trusted boot for our system. We identified three main components of this process: the SMX instruction GETSEC [SENTER], an authenticated code module, and the measured launch environment (MLE). Our analysis also revealed that TXT introduces multiple memory spaces to enhance security between components. While implementing TXT on top of QEMU, we encountered several issues that gave us a deeper understanding of the complexity of the TXT architecture. Our work highlights the importance of thoroughly examining the measuring launch environment process in order to ensure the security of our system and its ability to trust applications and prevent self-propagating malware, data breaches, and other attacks.

Key words - *TYCHE, Virtual machine monitor, Trusted boot, Intel trusted execution technology, measuring launch environment, authenticated code module, dynamic root of trust, rootkit*

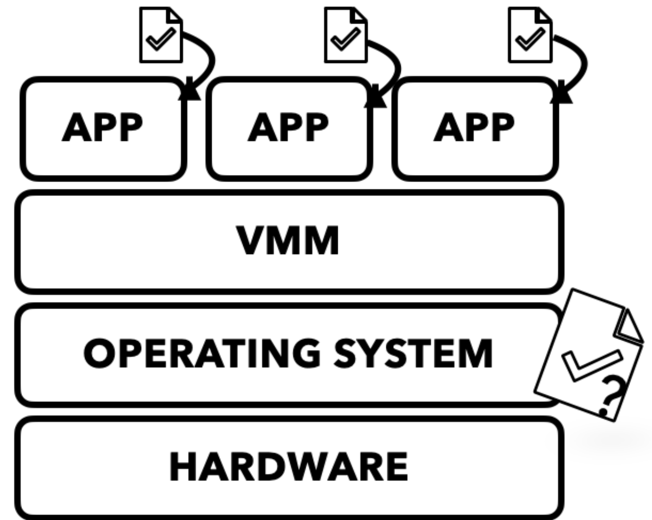

Figure 1: Global picture of our goal

1 INTRODUCTION

As cyber threats become increasingly sophisticated, organizations must implement strict security measures and carefully examine every aspect of their execution environment to protect against them. One type of malicious threat is the rootkit, which modifies or replaces core system files and programs with malicious versions in order to conceal itself and its activities. Rootkits can be introduced to a system through various means, such as exploiting security vulnerabilities, using social engineering tactics, or being bundled with other software. To ensure the security of the operating system and applications, it is necessary to establish and maintain isolation between them, so that the compromise of one does not affect the others. This is where the virtual machine monitor (VMM) comes in. By enforcing isolation, the VMM ensures that the security of the operating system and applications is maintained, even if one of them is compromised. If the VMM is compromised, the attacker could potentially bypass the isolation mechanisms and gain access to sensitive information stored in other parts of our system.

In this report, we will delve into Intel trusted execution technology (TXT) [2] and explore how it works. TXT adds security features such as measured launch and protected execution to the digital office platform, and is supported on certain Intel processors and chipsets (supported since Intel Core 2 Duo (released in 2006)/ICH9(released in 2007)). In order to utilize this technology, both hardware and software requirements must be met. At the hardware level, systems must be based on Intel Xeon processors with support for Intel TXT and Intel VT-x, and must also have a trusted platform module (TPM) added to the chipset to ensure that secrets are not spoofed. On the software side, the operating system and hypervisor must support Intel TXT. Our BIOS therefore needs to enable the trusted platform module and Intel TXT. On the software side, we just need to make sure that our operating system and hypervisor support Intel TXT.

One aspect of TXT that will be particularly useful in implementing a trusted boot is the measured launch environment (MLE). In this report, we will explore how to use MLE to measure a trusted part of the operating system (OS) and use this measurement to provide a certificate to our platform if

the subsequent boot measurement matches the trusted measurement. As illustrated in Figure 1, our goal is to certify that the operating system is booting in a trustworthy manner, serving as the trusted root of our system. Using Intel TXT, specifically MLE, we will aim to jump from one stage of the boot process (stage 1) to another (stage 2) (depicted in Fig.2) and store the measurement in the TPM. This certificate will enable us to request certificates for all applications running in our system from our operating system.

In order to achieve our goal, we will first gain an understanding of Intel trusted execution technology (TXT) and how to launch a measurement of it. We will then demonstrate the process for implementing TXT on our platform and identify any issues that arose during this implementation due to the use of QEMU, an open-source virtual machine and emulator that can be used to run operating systems. To address these issues, we will delve deeper into the interactions between the various components of the measuring launch environment process and examine how they can work together securely.

2 RELATED WORK

Trusted execution environment: TrustVisor

TrustVisor [7] is a type of hypervisor that is designed to create a trusted execution environment (TEE) on a computer system. A hypervisor is a software layer that allows multiple operating systems to run on a single physical host, by abstracting the underlying hardware and creating virtual machines (VMs) for each operating system to run on. TrustVisor is a specialized type of hypervisor that is designed to provide a higher level of security than traditional hypervisors, by creating a TEE for running trusted applications.

The TEE provided by TrustVisor is intended to be isolated from the rest of the system, and is designed to protect sensitive data and operations from tampering and external threats. TrustVisor is designed to be used for contexts where security is a high priority, such as in government, military, and financial applications. It is implemented as a type 1 hypervisor, which means that it is running directly on the machine, as we want Tyche to do. The type 1 hypervisors run at a lower level in the system than traditional hypervisors (type 2, such as Linux's KVM), and are able to provide a higher level of security and isolation.

Trustvisor uses a two-stage attestation process. It boots its hypervisor through the AMD Trusted Boot mechanism and creates an attestation for the enclaves. Clients can then verify the enclave's authenticity by checking both Trustvisor's hardware attestation and the enclave's software attestation.

This trusted boot and two-stages attestation mechanisms is what we are aiming to do with Tyche. One main difference is that it we will not use AMD trusted boot mechanism but the Intel one (TXT). The rest of the report will delve into how we plan to utilize Intel technology to achieve this goal.

3 INTEL TXT

3.1 Definitions

Definition 1 (Root of trust) :

Root of trust (RoT) in cybersecurity refers to a hardware or software component that serves as a starting point for determining the trustworthiness of a system. As RoT's are considered inherently trustworthy, they must be secured by design. These components can be used to verify the integrity and authenticity of firmware and software, ensuring only trusted software is executed on a device or system.

There are two types of root of trust: static and dynamic.

The difference between a static root of trust (SRoT) and a dynamic root of trust (DRoT) lies in their ability to be modified. SRoT, which is established at the time of manufacturing or deployment and cannot be altered, is used to establish a secure boot process and verify the authenticity of firmware and software. Although effective in maintaining the security of a device or system, it may not be able to respond to new security threats.

On the other hand, a DRoT can be updated to address new security threats or improve security features, providing ongoing security for the device or system. However, this flexibility comes at the cost of dependability, as a DRoT can be modified.

Another distinction between the two is the scope of the trust measure. SRoT covers the entire boot process from start to finish, while a DRoT can only be applied at a specific point in the boot process and may not include the firmware (such as BIOS or UEFI).

Definition 2 (Trusted boot) :

A system boot, where aspects of the hardware and firmware are measured and compared to known good values to verify their integrity and thus their trustworthiness. The trusted boot prevents rootkit malwares to stay unnoticed. Unlike secure boot, that only focus on hardware and provides a static root of trust for measurement, trusted boot provides a dynamic root of trust for measurement. In the DRTM the trust anchor is not a static value, but rather a constantly evolving and verifiable reference point that is used to establish trust in the system. If any of the measured values do not match, the boot process is halted and the user is alerted.

Definition 3 (Trusted platform module) :

It is a hardware component that provides security-related functions, such as secure boot and storage of cryptographic keys, passwords, and digital certificates. Trusted platform modules are used to enhance the security of a system by providing a hardware-based root of trust.

More specifically for our project, Intel TXT uses the TPM to compute hash of the measurement and to store it.

Definition 4 (Measuring Launch Environment) :
The measuring launch environment (MLE) will generally consist of three main sections of code: the initialization, the dispatch routine, and the shutdown. The initialization code includes code to setup the MLE on the ILP and join code to initialize the RLPs. After the initialization the MLE will behave as an unmeasured code. MLE prepares for shutdown by again synchronizing the processors, clearing any state, and executing the GETSEC[SEXIT] instruction.

The launching of the measuring launch environment of a computing system is the process of examining the system's firmware and configuration data to verify the authenticity and integrity of the system at launch. This process is typically used to establish a root of trust at launch and ensure that the system is running in a known good state.

Launch measurement involves generating cryptographically signed hashes of the system's firmware and configuration data, which are known as launch measurements. These launch measurements can then be verified against a trusted reference to ensure their authenticity and integrity. If the launch measurements are found to be valid, it can be assumed that the system's firmware and configuration are authentic and have not been tampered with.

Definition 5 (Authenticated Code Module) :
The ACM creates the hash of the measurement and store it into the trusted platform module, which serve to verify the integrity of the trusted execution environment (TEE) and the software running within it. By ensuring that only trusted code is permitted to run within the TEE, the authenticated code module helps to protect against malicious software attacks and contributes to the overall security of the system. Typically, the authenticated code module is implemented as a small piece of code stored in a secure location on the processor, where it verifies the integrity of the trusted environment's software and hardware components. There is two type of authenticated code modules used in Intel TXT (BIOS ACM and SINIT ACM).

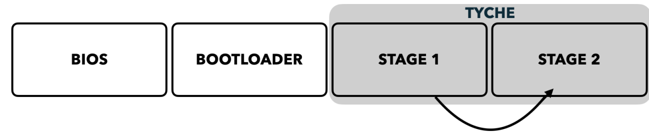

Figure 2: Boot of our system

3.2 Problem Statement

The goal of this research is to explore the use of Intel trusted execution technology [2] for implementing a trusted boot in the context of the TYCHE project. Specifically, we aim to answer the following questions: (1) How can we utilize Intel trusted execution technology to establish a trusted boot process? (2) How does the trusted boot process work using this technology? (3) What are the requirements for implementing a trusted boot using Intel trusted execution technology in our project? (4) How can we incorporate the use of Intel trusted execution technology for a trusted boot in the TYCHE project?

4 TRUSTED BOOT OF TYCHE

4.1 Our system boot

It is important to have a thorough understanding of the boot process of our system and the desired measurements. The boot process can be divided into four parts: BIOS, bootloader, stage 1, and stage 2 (Our boot is represented in Figure 2). Stage 1 is focused on properly configuring the CPU and launching the measured launch environment. Stage 2 follows stage 1 and is a defined code with a known location in memory (including the start and end points). In the context of trusted boot, stage 2 is the part of the boot process that we want to measure.

Currently, we are emulating our system on QEMU because it is too time-consuming to boot and debug on a real machine. It's worth mentioning that this emulation has caused some issues in implementing a trusted boot using Intel technology.

4.2 Intel documentation

In order to better understand how Intel trusted execution technology works we need to go through its documentation. From the first reading, we can spot that the launch of an Intel measuring launch environment is mandatory. Intel TXT enable us to trust our system using two concepts: The static chain of trust that measures the platform configuration, and the dynamic chain of trust that measures the system software, software configuration, and software policies. Our goal here is to implement a trusted boot to create a DRTM. As stated before the part we want to measure is stage 2 so the complex initialization code from stage 1 will not be part of this measurement. We will introduce in more details all the actors that

enable us to create such thing, the MLE, the ACM, the SMX instruction that will start the measurement. We will then see how these actors work together.

4.2.1 Measuring Launch Environment.

The measuring launch environment measure and sets up aspects of various hardware parts of our system: BIOS, chipset, processor, and the trusted platform module. At the beginning of our project it was then mandatory to add a TPM to the machine emulator. The trusted platform module role is to encrypt and store the measurement of our boot. Our operating system needs to communicate with the TPM in order to encrypt the measurement (thanks to hash functions) and store it. This stored hash will enable us to make a comparison with subsequent boot measurements. To begin the measuring process and store the measurements in the trusted platform module in a way that prevents software spoofing, it requires hardware such as specific chipset registers (PCRs).

Intel TXT measurement process begins with hardware, more specifically a microcode designed into the Intel processor. All parts of the firmware and hardware in their current state, can be seen after, the measurement, as a dynamic root of trust until the next boot of our machine.

4.2.2 Authenticated Code Module.

The authenticated code modules that are supported by Intel TXT are micro-code, signed by Intel. A platform aiming to use Intel TXT should use two different ACMs : the BIOS ACM and the SINIT ACM. The first one must be present in the boot flash, and referenced by FIT (Intel firmware interference table). The FIT allows to run code before the actual IA32 reset vector is executed by the CPU. It resides in the BIOS region. BIOS authenticated code is also known as STARTUP ACM. This AC module is used for performing subordinate tasks such as TXT Opt-in preparation, clearing the memory, alias checking, etc.

The AC module that will be more important in the implementation of our trusted boot is the second one, SINIT Authenticated Code module. This AC module is loaded into the internal RAM (referred to as authenticated code execution area or ACEA) within the processor, execution of the module therefore does not rely on or access any data stored in external memory or any activities that may be occurring on the external processor bus.We say that it is executed in isolation, which is key to enforce security of our system. The module is able to run independently, without being affected by or interacting with these external factors.

For this module to be executed it should firstly be authenticated. To authenticate the AC module, a digital signature is included in its header. The processor then calculates a hash of the AC module and compares it to the signature to determine if the AC module is valid. If the calculated hash matches the signature, the AC module is authenticated. Since we are using SMX instruction this authentication is necessary for the processor to execute this micro-code. This authenticated code module is a crucial part for the establishment of the dynamic root of trust.

Index (EAX)	Leaf function	Description
0	CAPABILITIES	Returns the available leaf functions of the GETSEC instruction.
1	Undefined	Reserved
2	ENTERACCS	Enter
3	EXITAC	Exit
4	SENTER	Launch an MLE.
5	SEXIT	Exit the MLE.
6	PARAMETERS	Return SMX related parameter information.
7	SMCTRL	SMX mode control.
8	WAKEUP	Wake up sleeping processors in safer mode.
9 - (4G-1)	Undefined	Reserved

Figure 3: GETSEC leaf functions

4.2.3 SMX instructions.

In order to launch the MLE and securely store and protect measurements from external parties, our platform must use safer mode extensions (SMX). The SMX interface includes multiple functions that support these goals, such as a measured launch of the MLE, mechanisms for protecting and securely storing measurements, and protection mechanisms that allow the MLE to control attempts to modify itself. Safer mode extension comes with numerous instructions to make our system or enclaves secure. The one instruction that enable us to launch the measurement of the measured launch environment is GETSEC.

The GETSEC instruction takes an immediate value as an operand, which specifies the particular operation to be performed, we call them GETSEC leaves. These leaves functions are selected by the value in EAX register at the time GETSEC is executed. Figure 3 [6]shows all the different GETSEC leaves and their EAX corresponding values. A GETSEC leaf can only be used if it is shown to be available as reported by the GETSEC[CAPABILITIES] function. If the leaf function is not available it will throw an undefined opcode exception.

The one leaf function we are focusing on is GETSEC SENTER. This is the instruction enabling us to launch the MLE (through the authenticated code module). To execute the SENTER leaf of GETSEC, the value 4 is set in the EAX register. The base address of the AC module to be loaded and authenticated is stored in the EBX register, and the size of the module in bytes is stored in the ECX register. The EDX register controls the level of functionality supported by the measured environment launch. If we want to enable full functionality for the protected environment launch, we must set the value of EDX to zero. SENTER leaf will load and authenticate the authenticated code module required by the measured environment, and enter authenticated code execution mode, verify and lock certain system configuration parameters. It will then measure the dynamic root of trust and store into the PCRs

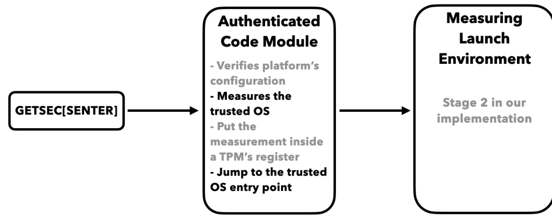

Figure 4: Launch of the measurement

in TPM and will transfer control to the MLE with interrupts disabled. Since the software part that we want to measure (the MLE) is Stage 2, GETSEC[SENTER] will jump into this stage after the execution of the Authenticated code module.

4.2.4 *Launching the measurement.*

After the more detailed presentation of actors of the measurement we can start to draw a high level view of the launching of such a measurement, depicted in Figure 4. The GETSEC[SENTER] instruction is used to initiate the measurement of code being executed and to establish a chain of trust for the code. When this instruction is executed, it sends messages to the chipset. The processor that initiates this process is called the initiating logical processor (ILP), it must be the system bootstrap processor (BSP), it is identified by setting the BSP flag in the IA32_APIC_BASE MSR to 1 (bit 8).

The ILP loads, authenticates, and executes the AC (authenticated code) module. The AC module checks the configuration of the chipset and processors to ensure that it meets certain standards, and then launches the measuring launch environment (MLE). The MLE initialization routine finishes configuring the system, including redirecting certain types of interrupts. At this point, all processors and the chipset are properly configured.

5 IMPLEMENTATION

Now that we have a clearer view on how the measuring launch environment works in Intel Trusted execution Technology, we want to implement it on our system. To be able to test and directly see this implementation we are, as a first step, emulating it on QEMU. We started by adding a trusted platform module to our platform, without it no MLE can be launch. We made sure that the rest of our platform enables the version of the added TPM.

5.1 GETSEC emulation

In our project, we encountered a problem with the SMX instructions, specifically the GETSEC one, which was not supported by QEMU. We therefore were not able to run GETSEC[SENTER] and start the measurement of our operating system software. We thus needed to try to emulate GETSEC. As a result, we decided to focus on emulating the SENTER and CAPABILITIES leafs, which seemed most important for our goals. We were able to successfully emulate the CAPABILITIES leaf, but encountered difficulties when attempting to create a simplified version of the SENTER leaf. Our goal in this simplified version was to catch errors, verify that the necessary conditions were met, and then jump to the authenticated code module that will proceed to stage 2.

To do so, in the way that imitates the most the true instruction we try to zoom in inside GETSEC SENTER and the SMX instructions in general.

5.2 Memory

5.2.1 *SMX interactions with the platform.*

To enhance the interaction between SMX and our platform intel TXT implements configurations registers[3], they are a subset of the chipset registers. Those registers are mapped into two different memory regions. The two regions separate the public and private configuration spaces, the private one can only be accessed after a measured environment has been established. Each regions come with its own permissions for a given registers (i.e. some registers can be read only on private mode). The registers are defined as 64 bits.

Two of this registers are useful for our simplified version of GETSEC:

- TXT.HEAP.BASE – TXT heap base address, contains the physical base address of the Intel TXT heap memory region. The BIOS initializes this register.
- TXT.SINIT.BASE – SINIT base address, the physical base address of the memory region allocated by the BIOS for loading an SINIT AC module is stored at this location. If the BIOS has provided an SINIT AC module, it can be found at this address. System software that includes an SINIT AC module must place it at this location.

After the AC Module authentication has completed successfully, the private configuration space of the Intel TXT-capable chipset is unlocked. At this point, only the authenticated code module or system software executing in authenticated code execution mode is allowed to gain access to the restricted chipset state for the purpose of securing the platform.

5.2.2 *TXT Heap memory.*

One of the components of Intel TXT is a special heap memory area that is used to store sensitive data and code. It is a region of physically contiguous memory that is set aside by BIOS.This heap memory is isolated from the rest of the system's memory and can only be accessed by code that has been authenticated and authorized to do so, in our case the

authenticated code module. The system software is responsible for filling in the table contents prior to executing the SENTER instruction. It passes data between the AC module and the MLE.

The part of heap memory that will be interesting for our simplified version of GETSEC is the part where the system software data passed to the SINIT AC module (OSSinitData). This part partly contains format and physical base address of the launch control policy (LCP), PMRs (protected memory regions). And more importantly for our emulation information about the MLE, such as: physical address of MLE page table (the MLE page directory pointer table address), size in bytes of the MLE image, linear address of MLE header (linear address within the MLE page tables).

As stated before the heap region of Intel TXT is only accessible by AC module or authenticated code. OSSinitData is where the ACM will take the necessary information to be able to know where to start the measurement and where to jump to execute the trusted OS.

5.3 Header

The previous registers only give us the address of the header of the AC module and MLE, but we need to know the entry point of those components to be able to execute them.

5.3.1 ACM Header.

An authenticated code module (AC module) is required to conform to a specific format. Every ACM has a fixed size header [4], it contains critical information necessary for the processor to properly authenticate the entire module, including the encrypted signature and RSA-based public key. The processor also uses other fields of the AC module for initializing the remaining processor state after authentication.

5.3.2 MLE Header.

SINIT AC module uses the MLE Header structure to set up the correct initial MLE state and to find the MLE entry point. The header is part of the MLE hash. The structure of this header is shown in Figure 5 [1]

5.4 Limitations

Seeing all the underlying layers and adds into the memory (adding a fake ACM, implementing fake registers...) needed to perform even our simpler version of the GETSEC SENTER leaf, we decided to save our tears, and our sanity. Our emulation of the measuring launch environment in QEMU performs validity checks to report errors that would arise on a real CPU and ease development. This more detailed version of the measurement is pictured in Figure 6

Field	Offset	Size (bytes)	Description
UUID (universally unique identifier)	0	16	Identifies this structure
HeaderLen	16	4	Length of header in bytes
Version	20	4	Version number of this structure
EntryPoint	24	4	Linear entry point of MLE
FirstValidPage	28	4	Starting linear address of (first valid page of) MLE
MleStart	32	4	Offset within MLE binary file of first byte of MLE, as specified in page table
MleEnd	36	4	Offset within MLE binary file of last byte + 1 of MLE, as specified in page table
Capabilities	40	4	Bit vector of MLE-supported capabilities
CmdlineStart	44	4	Starting linear address of command line
CmdlineEnd	48	4	Ending linear address of command line

Figure 5: MLE Header structure

5.5 Remaining questions

One of the questions addressed in this paper is the lack of information or literature on the implementation of Intel TXT trusted boot on QEMU. The paper provides an explanation for this gap in the available resources on the topic. It is simply too complex to be something common. There is, however, still some questions awaiting for an answer.

We did not went too deep in the interaction between the platform and the trusted platform module. We know these interactions happen between a dynamic root of trust and the TPM. Without the establishment of such a root of trust no information can be exchanged. We also know that the sets of interface registers accessible within a TPM device are grouped by a locality attribute. They each corresponds to a separate set of address ranges from the Intel TXT public and private spaces. For example there is:

- Locality 0 : Non-trusted and legacy TPM operation
- Locality 1 : An environment for use by the trusted operating system
- Locality 2 : MLE access
- Locality 3 : Authenticated code module
- Locality 4 : Intel TXT hardware use only

We need to understand how or with which instruction we can exchange information with the trusted platform module.

We can also try to critic the implementation of the measuring launch environment in Intel TXT. Even if the implementation seems to be meeting the security and trustworthiness expectations, why does it need to be this complex. Why the AC Module needs to be in a RAM "very secure" area and not inside the ROM. Since ROM can be considered somewhat isolated. It is not directly accessible by a computer's user or CPU and cannot be easily modified. It certainly have to be because it cannot be easily modified. A good point of the SINIT ACM is that it can be loaded from the internet (Intel

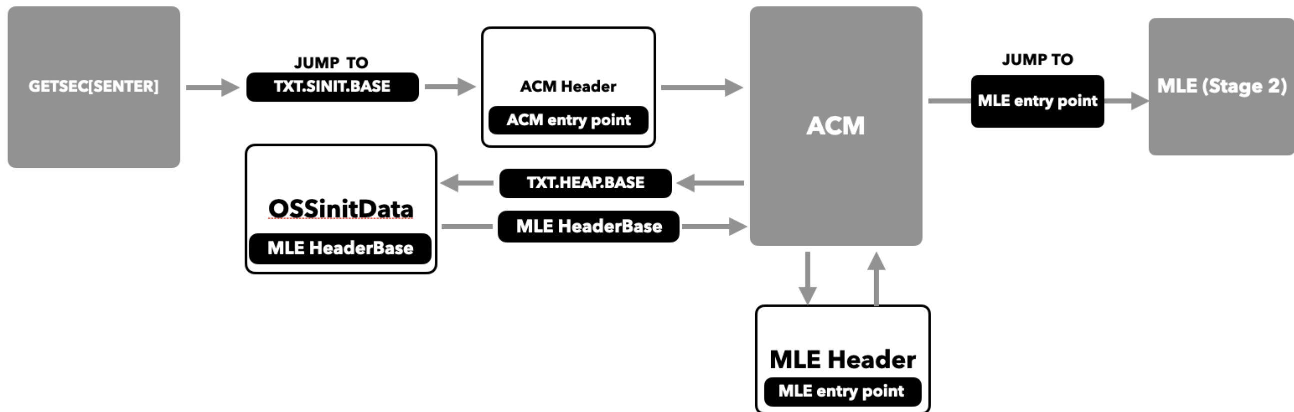

Figure 6: GETSEC[SENTER] emulation on QEMU

offers a platform [5] to download ACM depending on the platform configuration) and thus can fit to different configuration without having to replace the chipset.

Our project progress was hindered due to the absence of enabled SMX instructions in QEMU. A question that still remains is how SMX instructions could be effectively integrated into the QEMU framework. Our investigation delved into the underlying complexities of this implementation, specifically focusing on the GETSEC[SENTER] instruction and the various components of memory that it affects. Although our exploration was limited to a high-level and theoretical aspect, it would be intriguing to delve further and implement the different types of memory and their interactions within the framework of Intel TXT.

6 CONCLUSION

In conclusion, our work has allowed us to gain a deeper understanding of the measuring launch environment process defined in Intel trusted execution technology (TXT). Through our analysis, we identified the three main components of this process: the SMX instruction GETSEC [SENTER], an Authenticated Code module designed by the chipset vendor that runs in a secure area of RAM, and the Measured Launch Environment (MLE).

Our efforts to emulate TXT under QEMU uncovered a number of issues that prompted us to examine the underlying architecture in more details. We discovered that TXT introduces a range of different memory spaces to enhance the security of interactions between various components. These insights highlight the complexity of the architecture underlying TXT and provide valuable guidance for organizations seeking to utilize this technology to secure their systems. Overall, our work demonstrates the importance of carefully examining the measuring launch environment process in order to effectively implement TXT and ensure the security of digital office platforms.

REFERENCES

[1] Intel txt - 2.0 measured launched environment. Table 1 MLE Header structure.
[2] Intel txt - documentation. All documentation.
[3] Intel txt - documentation. Appendix B.1 Intel® Trusted Execution Technology Configuration Registers.
[4] Intel txt - documentation. Appendix A Intel® TXT Execution Technology Authenticated Code Modules.
[5] Intel txt - production sinit acm download. SINIT AC Modules.
[6] Intel® 64 and ia-32 architectures software developer's manual - 6.2.2 smx instruction summary. Table 6-2.
[7] Jonathan M. McCune Yanlin Li Ning Qu Zongwei Zhou Anupam Datta Virgil Gligor Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. Link.