POLITECNICO DI TORINO

Master degree course in Cybersecurity

Master Degree Thesis

Integration of Trusted Execution Environments and Digital Forensics for the Secure Preservation and Analysis of Digital Evidence

Supervisors
prof. Andrea Atzeni
dr. Grazia D'Onghia

Candidate
Andrea Geuna

Academic Year 2025-2026

Contents

Chapter 1

Introduction

1.1 Problem Statement and Motivation

In recent years, the digital world has become such a natural and pervasive part of our daily lives that it is now impossible to separate it from social, economic and even legal activities. Everyone owns at least a smartphone, uses online services, stores data in the cloud and interacts with applications that automatically record information. In the same way, cyberattacks have become more sophisticated and frequent, affecting not only large organisations but also individuals and home devices. In this context, digital forensics plays a central role: it is the discipline that enables us to collect, preserve, and analyse the traces left by digital systems, so they can be understood during an investigation and, if necessary, used as evidence in court.

The nature of traditional physical evidence, however, is different from that of digital evidence: they do not weigh, they do not occupy tangible space, and, most importantly, they can be modified without leaving visible traces. A file can be altered in a millisecond, a timestamp can be modified by malware, and memory can be overwritten simply by turning the device on. The guidelines published by the National Institute of Standards and Technology(NIST), an international benchmark in the field of information security, emphasise that preserving digital evidence requires attention and specific procedures precisely because this type of data is extremely vulnerable to inadvertent or malicious modification during all phases of the investigation, from the collection to the analysis, until the long-term preservation. [1]

Further complicating the situation is the growing variety of devices from which evidence originates: traditional computers, smartphones, IoT devices such as smart cameras or voice assistants, home automation systems, cloud services, and social media platforms. Each of these environments uses different technologies, different formats and storage methods that are not always transparent to investigators. Digital forensics must therefore contend with exceptional technological complexity, where investigators often find themselves working on compromised, outdated, or even intentionally designed systems that obstruct analysis. The latter category, where systems are intentionally manipulated, is referred to as “antiforensics”.

Traditional forensic methods were initially designed for devices on which the analyst could assume a certain level of trust in the operating system. But today, this trust is increasingly unjustified: the device may have been compromised by sophisticated malware, altered by an attacker, or improperly configured by the user. In such scenarios, there is no guarantee that forensic tools will operate securely, because a compromised system could interfere with data acquisition, modify it or prevent it from being properly read. So the risks are clear: a forensic image could be altered, rendering it unusable in court.

1.2 Trusted Execution Environments as solution

To tackle these limitations, in recent years, there has been a growth of interest in a technology seemingly unrelated to forensics but actually perfectly suited to solving some of its critical issues:

Trusted Execution Environments (TEEs). TEEs are secure environments integrated directly into the device hardware that allow code execution and data manipulation in a completely isolated manner from the operating system and other software. Intel SGX, ARM TrustZone and the latest solutions found in mobile processors or cloud services are examples of technologies that enable the creation of a sort of digital "safe" where to run sensitive applications, including forensic code and data. The idea of using a TEE is both intuitive and revolutionary: if evidence analysis could be performed in an isolated environment, many of the risks associated with a compromised system could be avoided. The investigator could rely on a protected environment, both hardware and software-based, in which critical operations cannot be observed or manipulated by external processes. Furthermore, TEEs offer a built-in attestation mechanism that allows showing not only that the data has not been altered, but also that the operation was performed correctly and with reliable tools. From a procedural perspective, this would constitute substantial added value, as it would strengthen the credibility and integrity of the evidence throughout the chain of custody.

1.3 Research Aims and Contributions

Despite the potential, there is currently no consolidated solution that integrates TEE and digital forensics tools at the structural level. Some studies and prototypes have explored individual aspects, such as the use of enclaves to protect the analysis of sensitive files or to verify data integrity, but a complete framework that allows existing forensic tools, such as Autopsy or The Sleuth Kit(TSK), to employ these protected environments in a transparent and standardised manner is still lacking. This thesis aims to contribute to this research gap. From an experimental perspective, the work involves building a prototype based on Intel SGX to verify the approach's feasibility in terms of both performance and reliability. One of the most interesting aspects will be evaluating how a TEE can improve the chain of custody; for example, an analysis performed in an enclave can offer greater guarantees than a traditional analysis phase on a potentially compromised system.

This thesis aims to provide an alternative solution in digital forensics regarding the protection of evidence during the analysis and exportation phases of a digital evidence chain of custody. As a solution, Forensic-SGX is proposed: a framework that combines the security features of Intel SGX with digital forensics, specifically by integrating The Sleuth Kit (TSK), a widely used forensic tool for analysing disk images across different filesystems, to ensure the preservation of evidence throughout an investigation. To further guarantee the integrity and authenticity of the evidence, the framework incorporates sealing, ECDSA signature, and DCAP remote attestation, making the forensic report verifiable by external parties, including an initial study of the implementation on the verifier side. This is essential in an investigative context, where evidence must remain verifiable over time and by trusted entities, such as a court, to allow accurate decisions about a case or further investigation.

The framework offers two modes of analysis: Standard Mode and Secure Mode. The Standard Mode represents the core contribution of the framework, providing the actual integration of TSK within an SGX application through a hybrid approach, where execution involves communication between the untrusted and trusted parts of the SGX application. The Secure Mode, on the other hand, is a complementary solution that replicates the TSK functionality through a custom in-enclave parser, currently supporting only the FAT32 filesystem.

During the testing phase, the limitations and vulnerabilities present in the framework will be highlighted. A comparative evaluation of the two modes was also conducted in order to determine which offers the best trade-off between security and performance.

1.4 Structure

The thesis is organised as follows.

Chapter 2 focuses on TEE. In particular, both Intel SGX and ARM TrustZone are presented. For Intel SGX, the architecture and memory isolation mechanisms that protect code and data

within the enclave are analysed in detail, along with the security features such as Remote Attestation and Sealing. The threat model, describing what SGX protects against and its known limitations, is also discussed.

Chapter 3 introduces the fundamental concepts of digital forensics, starting from its origins to the present day. Key concepts are defined and discussed, including digital evidence, chain of custody, data acquisition, and hashing. The section then explains the forensic process across its four principal phases, and concludes with a panoramic of the main forensic tools, with particular focus on TSK and its architecture. The chapter concludes with an analysis of the FAT32 filesystem architecture.

Chapter 4 examines existing approaches that integrate TEE academic research with practical implementations. The works cover a range of different use cases: as a first case, TEE-BI leverages the ARM TrustZone for remote forensic investigation; SGX-Log and CUSTOS protect the integrity of system logs using Intel SGX; Prov-Trust combines SGX and blockchain for data provenance; T3E provides trusted time services to SGX enclaves. Then, a dedicated study analysing the forensic challenges encountered in SGX isolation is reported. The chapter concludes by presenting the proposed Forensic-SGX framework, which draws inspiration from these existing works by addressing a different aspect: the protection of forensic evidence during the analysis phase.

Chapter 5 presents the Forensic-SGX framework in detail, describing its architecture, implementation, and three operational modes: Standard Mode, Secure Mode, and Export Mode. The chapter concludes with an initial implementation of the verifier, which acts as a verification simulator to demonstrate the correctness of the framework, and which can be used by any third party (such as a court or an independent auditor) to verify the integrity and authenticity of the forensic evidence without requiring access to the original disk image or the enclave.

Chapter 6 presents the testing activities conducted to evaluate the correctness, robustness, security, and performance of the Forensic-SGX framework. The tests are executed to assess input validation and robustness, sealing and unsealing integrity, hash chain verification, ECDSA signature correctness, and DCAP remote attestation. Subsequently, the security vulnerabilities and limitations identified during testing are also discussed, including rollback attacks, evidence metadata manipulation, and time manipulation attacks. The section concludes with a performance evaluation of the Standard Mode, Secure Mode, and the TSK Analyser (executed outside the Intel SGX environment) using different disk images, highlighting the differences between the security guarantees and efficiency offered by the modes.

Chapter 7 concludes the thesis by summarising the results obtained, discussing the identified vulnerabilities and limitations, and proposing possible solutions for future work.

Chapter 2

Trusted Execution Environment(TEE)

2.1 Security Challenges and Modern Requirements

As history shows, the need for a TEE predates the invention of computers, as sensitive activities require reliability and protection against malicious actors, eavesdroppers, thieves, and impostors. We can see the concept of TEE as a castle, with strong walls to defend against physical attacks, where sensitive information can be stored, and messages can be passed into the castle. A first example of TEE was introduced in the 1990s and is known as smart card chips [2]: they mimicked the protection of the castle, with secure storage, execution, communication, and sophisticated hardware and software defences against side-channel, physical, and fault attacks. Examples of this technology, such as smart cards, use dedicated security hardware for hosting sensitive data and applications. They are designed, developed, and certified as independent systems.

The major advantage of the isolated platform is that its security is independent of the main devices (e.g., mobile phone). Let's consider the following example: an attacker who wants to recover sensitive information on a SIM card cannot do so by gaining root access or by simply placing the device in another device. That is because direct access to critical assets is not granted (e.g., pre-module authentication keys or the International Mobile Subscriber Identity (IMSI) used to identify the SIM to a mobile network operator). [2]

Since the early 2000s, significant improvements have been seen in Big Data, Artificial Intelligence (AI), and Machine Learning (ML) domains. The increase of social media, Internet of Things (IoT) devices, smart watches, and the expansion of the big data analytics market have further contributed to this growth. Increasingly, organisations engage in data sharing to gain a more comprehensive understanding of users, products, and services, as seen in supply chain management, advertising, and pharmaceutical development. However, this approach raises substantial concerns regarding information security.

A 2021 survey commissioned by ARM [3] revealed that 96% of enterprise executives lack confidence in their organisation's ability to protect data from third parties. As a confirmation of this untrust, there are high-profile data breaches, such as the 2020 Marriott incident, in which more than 330 million guest records were exposed, and the LinkedIn breach that occurred in 2021, which affected more than 500 million accounts. Such incidents not only attract international media attention but also result in serious legal consequences under regulations such as the General Data Protection Regulation (GDPR), which can impose sanctions of up to 20 million euros or 4% of annual global revenue [2].

To address these complex security challenges, organisations must adopt a multilevel defence-in-depth strategy. This approach is adopted to organise security mechanisms in a hierarchical way across various system levels, including network infrastructure, host configuration, and application vulnerabilities, thereby enabling the identification and mitigation of attack paths at each level.

By requiring attackers to independently compromise multiple layers, this architecture significantly reduces the attack surface and provides multiple opportunities for detection before critical data are compromised [4].

2.1.1 Insecurity of Operating Systems

The x86 and x86-64 processor architectures implement a hierarchical protection mechanism consisting of four distinct privilege levels, designated as rings 0 through 3, as illustrated in Figure 2.1. Although hardware provisions exist for all four levels, Linux kernel design employs only two: level 0 for kernel-mode execution and level 3 for user-mode execution [5]. When applications in the user space that operate at level 3 need access to protected resources, such as I/O operations or memory management, they should pass through the kernel via the system call interface. This mechanism enables the kernel, executing at the privileged level 0, to perform critical operations on behalf of requesting applications while maintaining system integrity.

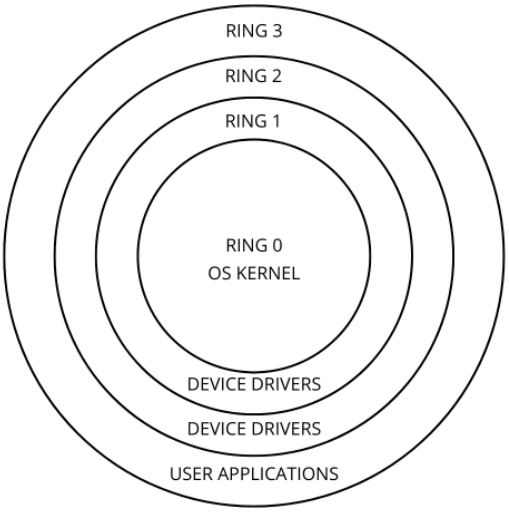

Figure 2.1. Operating system privilege rings

Therefore, operating systems face fundamental security challenges at the kernel level. Once attackers obtain kernel privileges, they gain unrestricted access to all system resources, including user-space processes, memory, network interfaces, filesystems, and I/O operations. This is highlighted by the fact that only the Linux Kernel has registered more than 13.500 vulnerabilities in its history [6], with a discovery rate in recent years drastically increased.

Among these, memory corruption vulnerabilities represent the wide threat category, with over 3,300 CVEs; instead, buffer and integer overflows contribute with an additional 419 vulnerabilities. The situation is further complicated by the fact that many vulnerabilities remain undetected for extended periods, leaving systems exposed to unknown threats during this latency window.

Why do these vulnerabilities continue to emerge?

- Growing complexity: the Linux kernel has had an exponential growth since its creation in 1991, expanding from 20 million lines of code in 2015 to over 40 million lines with version 6.14 in 2025. This huge codebase now powers diverse computing environments, including servers, supercomputers, smartphones, and IoT devices, making comprehensive security analysis and formal verification practically infeasible [7].

- Wide attack surface: these dimensions make formal verification impossible

- New hardware attack vectors: some micro-architectural attacks like Spectre or Meltdown exploit vulnerabilities at the hardware level that are difficult to mitigate via software [8]
- Legacy problems: OSs written in memory-unsafe languages (C, Assembly) with security mechanisms added reactively rather than proactively

Consequently, the TEE is proposed as a solution, offering a protective barrier in an era characterised by widespread insecure software development practices. The concept of a "trusted" environment takes on these complex challenges by providing a separate, protected area in which sensitive code is stored, processed, and safeguarded against a range of security threats.

The frequency of security and privacy attacks continues to rise globally, prompting manufacturers to enhance the protection of personal devices. Increasingly, manufacturers are adopting TEE to secure their products. TEEs are designed to prevent malicious smartphone applications from stealing users' fingerprint data, protect media content distributed by streaming platforms from piracy, securely store payment credentials, and safely process high-risk data in cloud environments. Furthermore, the construction and implementation of a TEE depend on the specific underlying technology.

2.1.2 Confidential Computing

Confidential computing has emerged as an essential security technology for addressing security and privacy challenges. This approach leverages both hardware and software security mechanisms to build a TEE that ensures confidentiality and integrity protection for data in use, where data in use is "data that is currently being updated, processed, erased, accessed, or read by a system, application, user, or device. This type of data is not being passively stored, but is instead actively moving through parts of an IT infrastructure" [9]. The rapid development of the digital economy has driven its adoption as a popular technology, supported both by government policy and industry initiatives. Moreover, both in technologies and products, confidential computing is also in a rapid growth stage. More specifically, the Confidential Computing Consortium (CCC) defines it as the protection of data in use by performing processing in a hardware-based, attested TEE [10]. However, it is important to understand that confidential computing is not simply a set of technology architectures that rely on TEE hardware, but a new type of secure computing model encompassing hardware, system, and data security. It is defined as a "computing paradigm for protecting and securing data in use" [11].

Confidential computing relies on a processor-based, modular architecture where hardware, firmware, and software work together. In this architecture, the hardware is the most critical component, with processors that support TEE capabilities (e.g., Intel SGX and AMD SEV). Hardware-enforced isolation of memory areas establishes isolated memory regions (enclaves/secure worlds) whose boundaries are protected by the hardware itself. Additionally, hardware and firmware components (e.g., secure boot and microcode) work in conjunction to establish a trusted execution baseline that verifies the integrity of the computing environment during device boot. On the software side, the stack comprises secure runtimes, TEE SDKs, remote attestation mechanisms (which allow external observers to verify that code is running in an authentic, unmodified TEE before sharing sensitive data with it), and application-level logic that utilises TEEs for secure execution. This layered architecture guarantees that even when vulnerabilities in higher-level software components are exploited, sensitive workloads remain isolated and protected within the enclave.

TEE Position in the Computing Stack

TEE plays a fundamental role in confidential computing by establishing secure compartments within the chip. This behaviour allows applications to run in an isolated, protected environment that is separate from the host OS, the hypervisor, and other applications. TEE guarantees confidentiality and integrity through hardware mechanisms, such as memory encryption, access control, and tamper verification. TEEs establish trust directly in the hardware, shifting the

focus of trust away from the software stack. This is especially important in cloud computing environments where infrastructure is shared and often opaque. Modern TEE implementations, such as Intel SGX, ARM TrustZone, AMD SEV, and specialised solutions like Apple's Secure Element and AWS Nitro Enclaves, enable secure enclaves to run unmodified applications with low performance overhead while providing attestation services that prove to remote parties that sensitive workloads are being executed in a secure and trusted environment [12].

Security Properties of Confidential Computing

We can define confidential computing as a framework designed with security properties to secure sensitive workloads throughout the execution lifecycle. The most important security properties are:

- Confidentiality: it makes certain that an unauthorised party cannot recover data processed inside a TEE (we obtain this by both hardware and through encryption of memory)
- Integrity: ensure that the code and data inside a TEE are not tampered with (this is achieved through secure boot, cryptographic measurements, attestation procedures)
- Isolation: hide communication between the trusted and untrusted sections of the system
- Resilience: the property to resists from a series of attacks

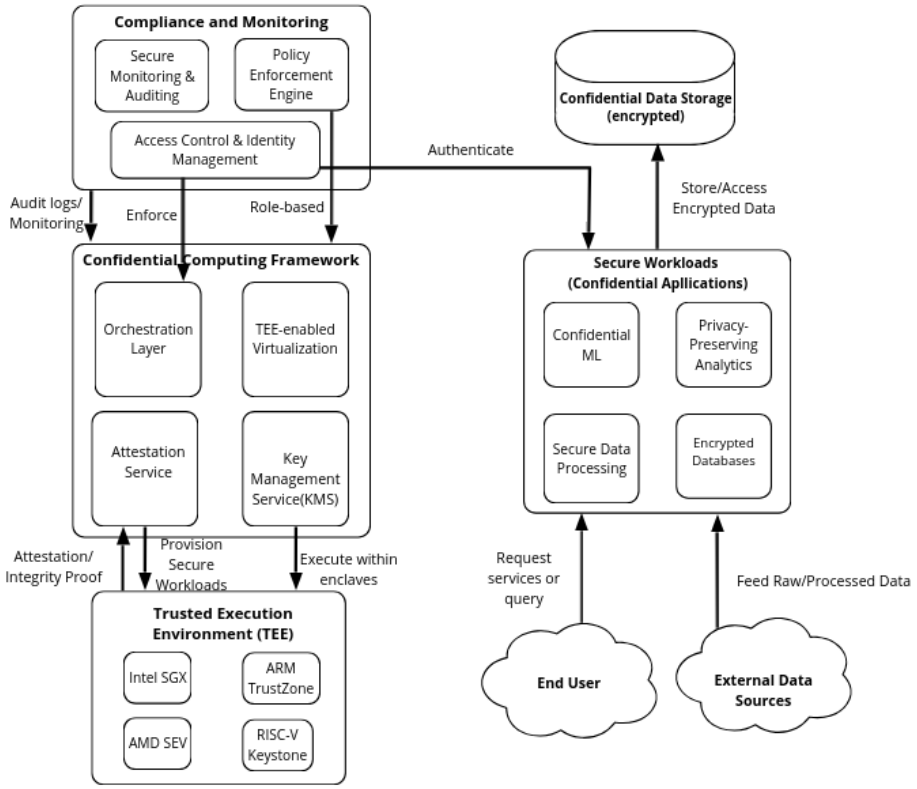

Figure 2.2. Confidential Computing Architecture Using Trusted Execution Environments (TEEs)

The confidential computing architecture depicted in Fig. 2.2 integrates multiple layers to ensure end-to-end data protection during processing [13]. At the core of this architecture is TEE, which, as shown in the figure, is implemented using technologies such as Intel SGX, ARM TrustZone, AMD SEV, and RISC-V Keystone. This is essential, as a secure workload is needed to execute sensitive operations securely.

The Confidential Computing Framework acts as a coordinator with the scope to secure execution via multiple components: an orchestration layer for workload deployment, a TEE-enabled virtualisation used for isolated runtime environments, attestation services to validate enclave integrity, and Key Management Services (KMS) for handling cryptographic keys. This framework allows Secure Workloads to execute confidential operations, such as secure data processing and preserving data confidentiality throughout the entire computation lifecycle.

Compliance and Monitoring mechanisms enforce security policies through access control, identity management, and permissions based on user roles. Audit logging and secure monitoring ensure regulatory compliance without jeopardising confidential data.

End users interact with the system by requesting services or submitting queries, while external data sources send raw or processed information into secure workloads.

All sensitive data resides in Confidential Data Storage, is encrypted both at rest and in transit, and is accessible only after successful authentication through the confidential computing framework.

2.2 ARM TrustZone

2.2.1 Secure World Architectures

At the core of trusted architecture emerges a fundamental issue, that is, the intrinsic difficulty of creating a single execution environment that can be fully trusted. This approach encounters two critical limitations. First, managing TCB that evolves frequently over time poses considerable difficulties, particularly when attempting to maintain TCB minimisation: " A trusted computing base (TCB) refers to all hardware, firmware, and software components of a system that provide a secure environment. The components inside the TCB are considered critical. If one component inside the TCB is compromised, the entire system's security might be jeopardised. A lower TCB means higher security" [14]. Second, organisational heterogeneity affects deployment: financial teams require accounting software, developers need programming tools, and sales personnel depend on customer relationship management systems. This variety of operational requirements makes it impractical to maintain a uniform, minimal TCB across an organisation. Moreover, users install and remove software as needed, so changes are frequent and continuous, and the heterogeneity of devices includes laptops, servers, and mobile devices with different OS, hardware, or firmware versions. The tracking is also impractical: tracking what an expected platform configuration, like using a TPM's PCR values, becomes so difficult.

The solution to this problem is to adopt a different design approach, using a "trusted" separate environment with a TCB significantly reduced (the trusted world). This trusted environment is designed to host rigorously tested software components for security-sensitive applications and applications that are too sensitive to be protected solely by OS-level controls. Indeed, the untrusted system is untouched in practice: it contains just the ordinary OS and the user applications used daily.

The key components of the trusted world are:

1. Root of Trust: initialises the TEE in the trusted world, and launches the security monitor
2. Security Monitor: handles the transitions between the main OS (in the untrusted world) and the trusted world in execution. It makes sure that the transitions are valid, authorised, and executed safely
3. Security-Enhanced Kernel: the trusted world TEE contains a security kernel that is developed to resist a set of defined threats; it can also be certified independently under the Common Criteria framework [2]
4. Trusted world TEEs use the same system-on-chip, which has been extended to allow the secure execution of a trusted world (the SoC is extended in order to prevent unauthorised access to cache entries, RAM address regions, and interrupts associated with the trusted world)

Definition of Trusted World: "A trusted world contains an independent security kernel, drivers, and related components for hosting a set of security-sensitive applications. Importantly, the world is self-contained and operates without relying on an external operating system for maintaining its security. The isolation of the trusted and non-trusted worlds is implemented using a hardware-based root of trust" [2].

2.2.2 GlobalPlatform TEE

GlobalPlatform TEE (GPTEE) [2] is a suite of specifications that defines the architecture, the functionality of trusted world-based TEEs, the requirements, the initialisation, run-time operation, and management of the TEE. A GPTEE implementation is formally recognised as compliant with the Initial TEE Configuration [15], which comprises all the necessary specifications to guarantee a uniform and integrated framework. There are 2 mandatory specifications:

- GPTEE Client API Specification: it defines the interfaces for applications in the non-secure world to connect and use the services implemented by Trusted Applications (TAs: applications that reside inside the TEE)
- GPTEE Internal Core API Specification: it specifies the programming interfaces at the function level to permit TAs to access services provided by the TEE (some examples of services are access to dynamic memory allocation, use of AES encryption, and data submission for secure storage)

Regarding security requirements, a GP TEE-compliant implementation should satisfy the security requirements established in the GP TEE Protection Profile document [16]. This document details the protection scope for the TEE, the security requirements for TEE assets (TAs and associated data), and the threats and vulnerabilities that should be addressed. However, GlobalPlatform TEE offers important implementation flexibility: it does not specify how the trusted world should be implemented in hardware and software, and does not impose the use of specific operating systems, programming languages, SoCs, or TAs that must exist on the target of evaluation.

ARM TrustZone is a solution for implementing a GPTEE on mobile devices and embedded systems where ARM processors are widely used: the TrustZone goal is to provide security for those platforms. TrustZone is a hardware security extension that provides a secure execution environment by dividing computer resources into two execution worlds: the normal and secure worlds. Note that the idea is to partition all of the SoC's hardware and software. The secure world encompasses everything that executes when the processor operates in a secure state, while the normal world includes everything that runs when the processor is in a non-secure state. Hardware-based barriers are implemented to block normal world components from accessing secure world resources; conversely, the secure world faces no such restrictions. More specifically, the memory system blocks the normal world from accessing: i) physical memory regions marked as secure; ii) system controls that govern the secure world; and iii) state transitions outside of a limited set of authorised mechanisms. This partitioning can be implemented physically and/or virtually. For example, a physical processor core is shared between the normal and secure world in a time-multiplexed manner, creating the illusion for both worlds that each owns the processor exclusively. The secure world allows the creation of an isolated programmable environment capable of executing a diverse range of security applications. Finally, ARM TrustZone has the goal to defend against privileged non-secure world software attacks (e.g., kernel-mode adversaries); note also that secure worlds are used to carry out the GPTEE Internal APIs, the trusted OS, the TAs, and any drivers that are used to communicate with the REE [17].

TrustZone for Cortex-A

In the ARM architecture, there are four distinct privilege levels: EL0 is the user space applications (least privilege), then there is EL1, which is intended for the OS (here are present all the components that are used to execute memory management, process isolation, and scheduling). The next layer is EL2 that is used by the hypervisor to support multiple OSs and user applications. Finally,

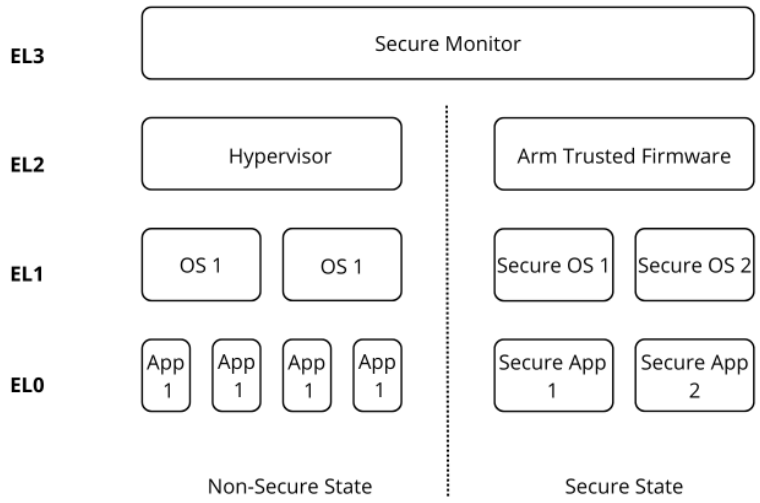

Figure 2.3. ARMv8 TrustZone Architecture

EL3 is the last level and provides secure monitor mode (used to implement security-critical functions like secure boot). The secure monitor is also used for transitioning between the secure and the untrusted worlds, also called "world switch", and it is activated by using the Secure Monitor Call instruction (SMC). After this instruction, the monitor transitions to EL2, which moves to the secure world OS (EL1). Authorisation procedures should be implemented in the monitor or in the secure world component to prevent unauthorised requests arriving from the REE. After the operation is executed in the secure world, the secure monitor uses the exception return instruction (eret) to switch back to the non-secure world (resuming normal execution). To denote the current world of execution, the NS bit of the Secure Configuration Register (SCR) in the System Control Space (SCS) is used, which contains security configuration settings (interrupts, exceptions, ...), and the NS state is transmitted throughout the different system-on-chip components via the associated system buses. When:

- NS is set to 0: the processor is operating in secure mode (NS is set to 0 every time the processor executes the smc instruction) b
- NS is set to 1: the processor is in non-secure mode (typically after eret is executed)

However, also by looking at the image 2.5, we can see that there are a lot of sub-components that are needed to handle secure world accesses to different SoC components, and that are used to mediate secure and non-secure world accesses to different types of peripherals and memory units, and a SoC could use only some or all of these. They are called TrustZone security controllers:

- TrustZone Protection Controller(TZPC): SoC component that is used for guaranteeing that only the appropriate world can access on-chip peripherals (e.g., over UART, SPI, I2C) and bus accesses to external, off-chip components.
- TrustZone Memory Adapter(TZMA): is used for partitioning static memory units used by the secure and non-secure worlds. This mechanism enables the designation of a fixed, secure memory region by specifying its base address and size at chip initialisation time. When configured, TZMA controls access to the secure world memory region by preventing all non-secure accesses (note: it is not used for dynamic RAM).
- TrustZone Address Space Controller(TZASC): it is used for enforcing complex access permissions for memory units, making sure that the secure and non-secure worlds can only access their designated memory area. Like TZMA, it segregates the memory space. It is also re-configurable, so developers can define the memory regions and their access at runtime. It allows dynamic setup of multiple secure and non-secure memory regions with

variable sizes and base addresses, granting enhanced granularity and flexibility in memory protection.

The previous controllers are placed to protect the different components of the SoC from unauthorised access from the untrusted world. By the way, it is also important to defend the states of the processor itself:

- protecting cache contents: cache partitioning divides the processor cache into separate regions for trusted and non-trusted environments, preventing unauthorised access to protected data memorised in the cache. Special privileged instruction allows the safe word to clean, invalidate or remove the line from the cache, guaranteeing correct cache maintenance. When the non-secure world needs cache operations that affect the secure world, it must request that the secure monitor perform them on its behalf. In the ARMv8-A architecture, cache partitioning is achieved using the Security Attribution Unit (SAU) and the Memory Protection Controller (MPC).
- Protecting TLB Entries: Translation Lookaside Buffer(TLB) entries require protection in addition to CPU caches. The TLB stores recent translations of virtual memory addresses to physical memory addresses, and it is partitioned into secure and non-secure zones on the TrustZone system. Special privileged instructions enable explicit invalidation, flushing, or synchronisation of TLB entries within the secure world. When the non-secure world needs TLB maintenance operations, it must request the secure monitor to perform them through the SMC instruction
- Secure boot: it is essential for the TrustZone secure world initialisation. The boot process of a TrustZone-enabled system-on-chip begins in a dedicated ROM module that stores cryptographic keys for initiating a chain of trust. Executing at the processor's highest privilege level, the ROM code authenticates and subsequently loads the bootloaders and the images for both the secure and non-secure worlds.

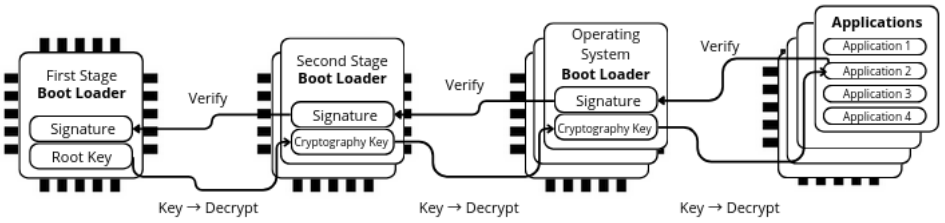

Figure 2.4. Secure Boot Process

- Interrupt Handling: separates fast interrupt requests (FIQs) from standard interrupts (IRQs), typically assigning FIQs to the secure world and IRQs to the non-secure world. The ARM Generic Interrupt(GIC) Controller configures interrupt routing to either world. When triggered, the processor saves its state and jumps to the corresponding interrupt handler, with secure and non-secure worlds maintaining independent handlers. An alternative approach traps both interrupt types in secure monitor mode for increased control.

2.3 Intel Software Guard Extension (SGX)

2.3.1 From Secure Worlds to Enclaves

As we saw in the previous chapter, ARM TrustZone uses an independent secure world to achieve confidentiality and integrity of data and applications. It contains its own firmware, operating system, and a set of trusted applications. However, this typology of TEE presents some issues:

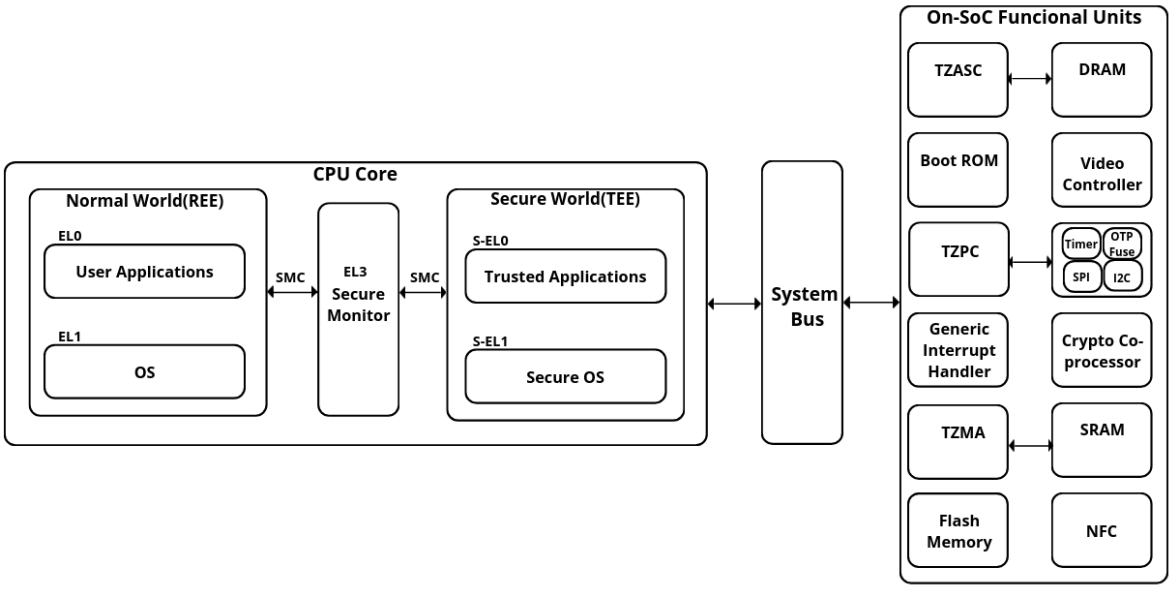

Figure 2.5. Example of an ARM TrustZone system-on-chip for Cortex-A

- Only one secure world is supported: no native support to create multiple hardware-isolated secure worlds belonging to different stakeholders
- Total control of the OEM (Original Equipment Manufacturer): it controls the secure world, by providing and provisioning the trusted OS, firmware, and the Trusted Applications. Moreover, the secure world image is signed by the OEM and loaded at boot time. So when a device is shipped, third parties cannot make any changes without first contacting the OEM.

To handle these problems, Enclave Computing emerged [2]. The idea here is to have multiple "worlds" coexisting on the same platform, each belonging to a different service provider. These environments are called enclaves. An enclave is an area that allows developers to delegate sensitive operations within the same application, using the same hardware and operating system. An enclave does not depend on a dedicated trusted OS to maintain its security, nor on a particular OEM to authorise its installation and utilisation. The isolation of these enclaves, which belong to different owners, is based on a hardware root of trust for their runtime protection and management.

Table 2.1 summarises the key architectural differences between ARM TrustZone and Intel SGX, showing how the enclave-based approach addresses the limitations discussed above. A detailed description of Intel SGX architecture and its security features will be provided later.

Moreover, software is nowadays deployed on remote computers that are not owned by a trusted party (e.g., within environments such as Infrastructure as a Service (IaaS), such as Amazon Web Services) [18]. For this reason, we may encounter security problems: an untrusted party could try to steal sensitive data from the platforms to which it has access, or the remote computer could be owned by an untrusted party. So the diffusion of these services created the need for a way to trust code running remotely and to guarantee the integrity of secrets stored remotely, and the solution for this is Intel SGX.

Intel SGX is a set of hardware extensions for x86 systems developed by Intel with the objective to improve the security of data and sensitive applications within environments containing untrusted system software [19]. In fact, with Intel SGX, even on compromised systems, the processor acts as a trusted component by isolating sensitive code and data from other software, which cannot access them without specific authorisation from the trusted hardware. However, to be efficient, the Trusted Computing Base of SGX should be limited: only the processor's microcode and the code running in the enclaves should be trusted [20]. Another important feature is remote

attestation [21]: an SGX enclave can be remotely attested to verify that everything is working correctly before receiving sensitive data and code. It can support multi-core processors and sustain high performance for advanced computations. Moreover, it is available on Intel processors from the 6th generation (2015) to the 10th generation on the client side, while it is available for server processors since 2015 (more specifically at the end of 2021, it was announced that Intel SGX would be deprecated on Intel Core processors, although it remains available on Intel Xeon processors designed for server workloads [18])

	ARM TrustZone	*Intel SGX*
Isolation Model	Two worlds: Secure World and Normal World	Multiple enclaves coexisting on the same platform
Isolated Environments	Single secure world; no native support for multiple hardware-isolated secure worlds	Multiple "worlds" belonging to different service providers
Control and Management	Total OEM control: provides and provisions trusted OS, firmware, and TAs	Enclave does not depend on a particular OEM to authorise installation
Deployment	Secure world image signed by OEM and loaded at boot time	Isolation based on hardware root of trust, independent from OEM
Trusted OS Dependency	Requires an independent security kernel, drivers, and components	Does not depend on a dedicated trusted OS
Trusted Computing Base	Includes firmware, trusted OS, drivers, Trusted Applications	Minimal: only processor microcode and enclave code
Remote Attestation	Not natively supported [21]	Native support: EPID and ECDSA/DCAP
Memory Encryption	Isolation via NS-bit; no native DRAM encryption [21]	Memory Encryption Engine (MEE) with AES
Sealing	Secure storage via software [21]	Native hardware support with MRENCLAVE and MRSIGNER policies
Target Platforms	Mobile and embedded systems (ARM processors)	x86 systems, Intel 6th-10th gen and Xeon processors

Table 2.1. Comparison between ARM TrustZone and Intel SGX.

2.3.2 SGX Threat Model and Security Properties

Intel SGX is designed to protect against an extensive threat model that includes adversaries with varying levels of system access. In the x86 architecture, software operates at different privilege levels, commonly referred to as rings [19]. Ring 0 represents the most privileged level, where the operating system kernel executes, while Ring 3 is the least privileged level, where user applications run. Typically, any privilege level can access and manipulate data at lower privilege levels. Transitions between these levels occur via well-defined mechanisms, such as the SYSCALL instruction, which allows ring 3 code to request services from ring 0, and the SYSRET instruction, which returns control to user space. This stratified model is fundamental to modern operating system security. SGX's threat model handles three primary categories of adversaries [18]:

1. Unprivileged software adversary (or ring-3 attacker): operates with limited permissions granted by the system software and can only execute user-level instructions (ring-3 instructions).

2. System software adversary: possesses full control over the operating system and can read or write to all available memory, schedule code execution, and even launch malicious enclaves.

3. Startup code adversary: has compromised the BIOS and possesses complete control over the platform during boot, including the ability to modify system management mode registers. Additionally, SGX assumes that attackers may have physical access to the platform, enabling them to observe and store all DRAM traffic.

SGX does not protect against all possible threats: in fact, some attacks, such as side-channel attacks that exploit observable processor statistics like power consumption, cache behaviour, or memory access patterns, fall outside SGX's security guarantees [22]. Similarly, availability threats like denial-of-service attacks are not addressed, as physical access inherently allows an attacker to disable the system [18].

2.3.3 SGX Memory Access Protection

During execution, Intel SGX use a TCB based on limited hardware on the CPU itself, introducing different specialised memory components to facilitate the management, isolation and attestation of the enclaves [19].

A fundamental component of SGX memory protection is the Processor Reserved Memory (PRM), a contiguous block of DRAM dedicated to hosting enclave data. The PRM size is configured at the start of the BIOS through a specific register of the model, known as PRMRR, and this configuration cannot be altered thereafter. Standard implementations support maximum sizes of 128 MB or 256 MB, though certain Intel Xeon Scalable processors (3rd generation and newer) can accommodate up to 512 GB. The specified size must be an integer power of two to allow efficient hardware-based boundary checking. Within the PRM resides the Enclave Page Cache (EPC), which contains the actual code and data used by enclaves along with associated metadata. The EPC is organised into 4 KB pages aligned at 4 KB boundaries (Fig. 2.8). Each EPC page is tracked by the Enclave Page Cache Map (EPCM), a protected structure that stores critical attributes, including whether a page is allocated to a particular enclave, access control permissions (read, write, execute), and a pointer to the enclave's owner through an SGX Enclave Control Structure (SECS). The EPCM recognises several page types: SECS pages containing enclave identity and measurements, Regular (REG) pages holding actual enclave code and data, Thread Control Structure (TCS) pages for multi-threading support, and Version Array (VA) pages for tracking evicted pages to prevent replay attacks. As depicted in Figure 2.7, the memory architecture contains the system DRAM (containing both general system memory and the protected PRM region), the EPC structure (used to store enclave components such as metadata, SECS, TCS, stack, heap, and code;) and the protection mechanism of the CPU (EPCM that is used to enforce access controll and the Memory Encryption Engine (MEE), essential for compute encryption enclave pages).

Access control to memory in SGX happens through a verification process divided into multiple steps. When a memory access occurs, the processor's Memory Management Unit (MMU) first translates the virtual address to a physical address using standard page tables, and then SGX-specific checks are performed based on the execution mode. In enclave mode, the CPU verifies that requested addresses reside within both the PRM and the Enclave Linear Address Range (ELRANGE). Each enclave designates ELRANGE as a dedicated region in its virtual address space, which is used to map the code and sensitive data stored in the enclave's EPC pages [19]. The CPU then consults the EPCM to confirm page ownership and access permissions. Any violation results in a page fault exception. In non-enclave mode, the process is more straightforward: any attempt to access addresses within the PRM is immediately blocked, preventing untrusted software from accessing enclave memory. Conversely, enclaves themselves can freely access non-PRM memory, maintaining necessary interaction with the untrusted system. This validation procedure is illustrated in Figure 2.6. To protect enclave data during transport from the CPU to DRAM, Intel provides the Memory Encryption Engine (MEE). In SGX's threat model, only the CPU is considered trusted; DRAM is treated as unreliable.

Using AES in counter mode at the cache-line level (512 bits), the MEE continuously encrypts and decrypts data as it moves to and from the EPC, with encryption keys generated at boot-time by an internal pseudo-random number generator. Message authentication codes are used in an encrypt-then-MAC paradigm to preserve integrity; verification failures cause the system to hang until a hard reset.

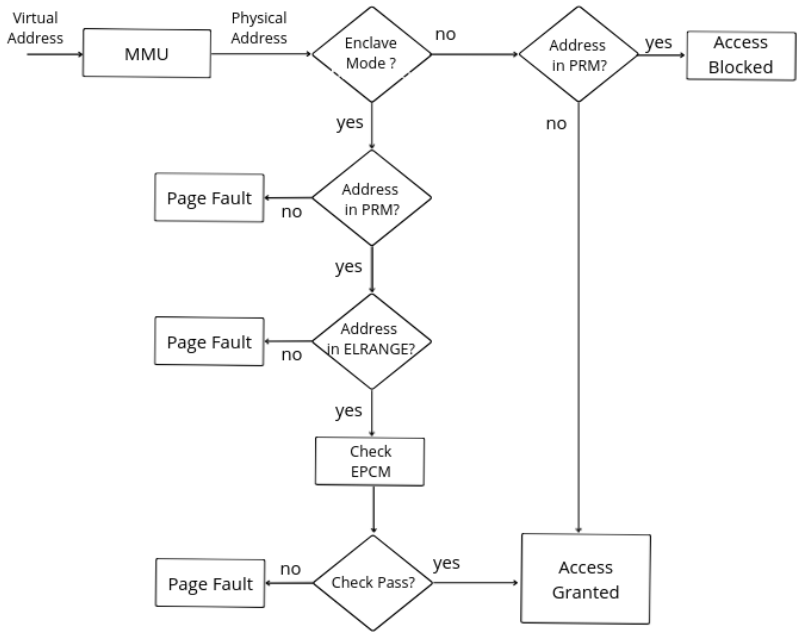

Figure 2.6. SGX Memory Access Verification Process

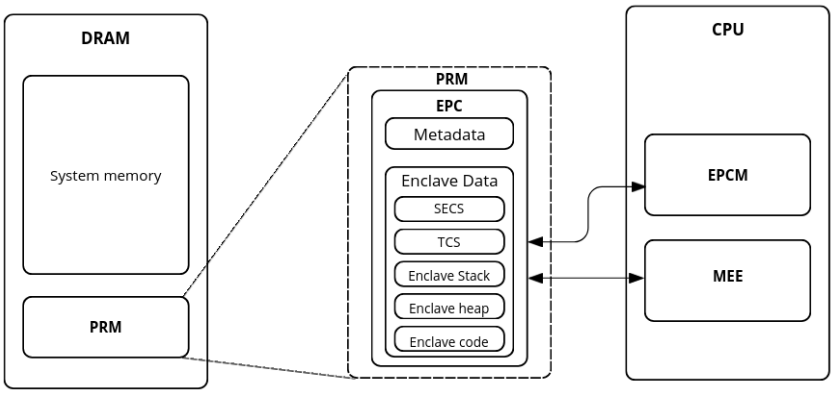

Figure 2.7. Intel SGX's memory organisation

2.3.4 Remote Attestation

Intel SGX supports two forms of attestation: local attestation and remote attestation. Local attestation (or intra-platform attestation) allows enclaves running on the same platform to verify each other's identity and integrity. Instead, remote attestation (or inter-platform attestation) provides the foundation for establishing trust between an enclave and a remote third party, enabling secure communication and data exchange across untrusted networks. In particular, remote attestation has the goal to authenticate an enclave that is running on a remote untrusted platform and to verify that it is behaving as expected.

Both mechanisms rely on the platform's ability to produce a cryptographic credential (the report) that accurately reflects the enclave's signature, including information about its security properties and current state. More specifically, an attestation report contains the identity of the software to be attested, all details of non-measurable states, the data associated with the software, and a cryptographic signature. The signature procedure in the attestation process ensures that the relying party can verify it is interacting with a genuine hardware-based enclave rather than a software simulation. However, to build trust within an enclave, it is necessary to have some hardware-based keys: two keys are permanently burned into the CPU's eFuses when an Intel SGX

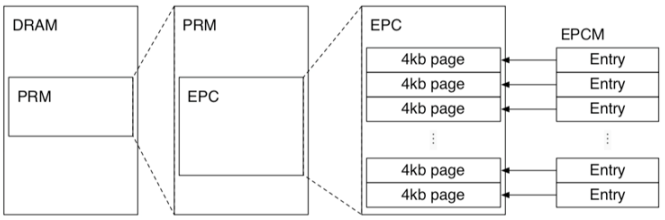

Figure 2.8. Processor Reserved Memory (PRM)(Source: SGX 101[23])

CPU is manufactured. The keys in question are the Root Provisioning Key (used to demonstrate that the CPU is a genuine Intel processor, which Intel also possesses) and the Root Sealing Key (used to implement the sealing infrastructure for securely storing persistent enclave data). Based on these hardware-rooted keys, Intel SGX implements a structured attestation bootstrap process that enables the secure generation and verification of attestation evidence. Figure 2.9 illustrates how the root keys embedded during manufacturing are leveraged to derive provisioning credentials, which are subsequently used to obtain platform-specific attestation keys. These keys are managed by dedicated system enclaves and are never exposed outside the processor. The Quoting Enclave uses the attestation key to generate a signed attestation report, commonly referred to as a quote, which contains both the enclave's identity and its security-relevant attributes. The quote is then verified by a trusted attestation service, which validates the signature and checks the platform's revocation and security status. This process establishes a cryptographic chain of trust from the hardware level up to the remote relying party.

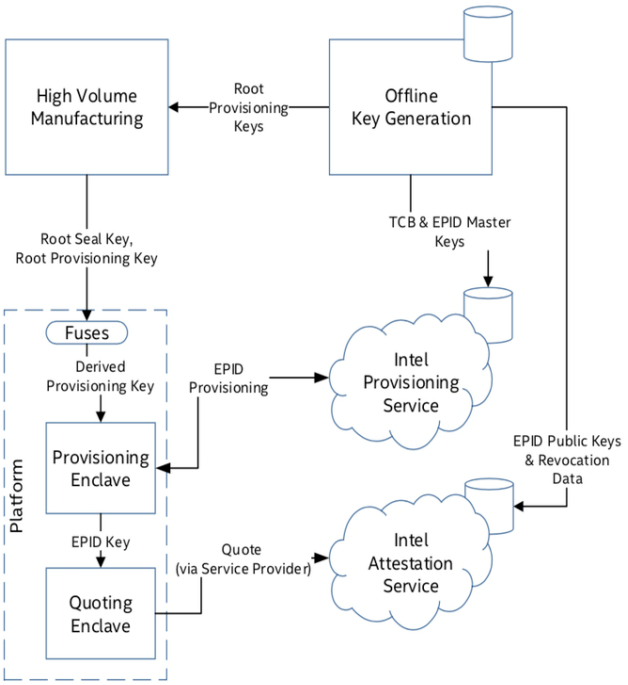

Figure 2.9. SGX Infrastructure Services (Source: SGX 101 [23])

In our case, we are more interested in remote attestation, so we will examine it in detail. An SGX-enabled processor calculates a cryptographic measurement of the code and data loaded into each enclave, similar to the TPM's measurement process. The objective of SGX remote attestation is to enable a remote party to verify that an enclave has been loaded into an authorised state. The process also allows developers to establish a secure channel between the enclave and the remote party. In this way, the challenge of secret provisioning is addressed. Rather than embedding

secrets directly into the enclave binary, SGX is designed such that this data must be provisioned after the enclave has been launched.

Remote attestation in Intel SGX relies on a combination of symmetric and asymmetric cryptographic mechanisms. Symmetric keys are used internally for local attestation between enclaves on the same platform, while remote attestation leverages asymmetric attestation keys, namely EPID keys, to enable external verification. The protocol involves three main entities: the service provider acting as the challenger, the user platform hosting the application enclave, the Quoting Enclave (QE), and the Intel Attestation Service (IAS) 2.10.

The attestation process begins when the application enclave contacts the service provider to declare its EPID group membership. If the service provider decides to proceed, it retrieves the latest revocation information from the IAS and issues a challenge containing freshness data and attestation parameters. The enclave responds by generating a locally verifiable report addressed to the QE, optionally embedding additional data required to establish a secure channel. The QE performs local attestation to verify the report, accesses the platform's attestation key, and generates a signed quote that binds the enclave identity to the platform state. This quote is forwarded to the service provider, which delegates its verification to the IAS. Upon successful validation, the IAS returns an attestation verification report, allowing the service provider to establish trust in the enclave and respond accordingly. Intel SGX supports two types of remote attestation: Elliptic Curve Digital Signature Algorithm (ECDSA) and Intel Enhanced Privacy ID (Intel EPID) Attestation [24].

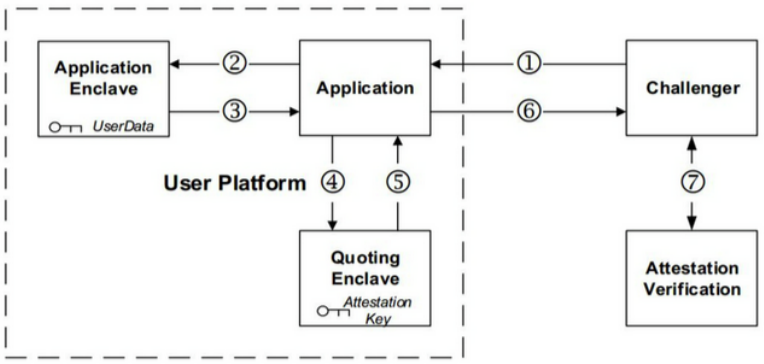

Figure 2.10. Remote Attestation Process (Source: SGX 101[23])

Intel Enhanced Privacy ID (Intel EPID)

Intel EPID is a Direct Anonymous Attestation signature scheme used to support the revocation of illegitimate members. EPID signatures are completely anonymous, even to Intel. Moreover, the EPID scheme allows SGX enclaves to demonstrate their execution on authentic Intel SGX hardware while preserving the anonymity of the underlying processor's unique identifier. During SGX software deployment, the Provisioning Enclave (PE) contacts Intel's EPID Provisioning Server to register the CPU in the EPID group using its Root Provisioning Key (RPK). Subsequently, an attestation key is delivered to the Provisioning Enclave, which is then utilised by the Quoting Enclave(special enclave) to cryptographically sign enclave reports throughout the attestation process. Following successful enclave attestation, the EPID implementation establishes a secure communication channel that can be used to provision secrets into the attested enclave

Elliptic Curve Digital Signature Algorithm (ECDSA) and Data Center Attestation Primitives(DCAP)

Large-scale SGX implementations, particularly in data centre environments, face a major challenge: sustaining continuous communication between every platform and Intel is often unfeasible.

To overcome this limitation in cases such as offline deployments or organisations that delegate trust management, Intel developed the DCAP. This solution allows organisations to perform local enclave certification while preserving cryptographic verification back to Intel's root of trust. Central to DCAP is the Provisioning Certification Enclave (PCE), a specialised enclave that relies on two cryptographic elements retrieved through the EGETKEY instruction. These items are the Provisioning Certification Key (PCK), which is bound to the specific trusted computing base configuration, and the Platform Provisioning ID, which serves as a unique platform identifier.

A key difference in DCAP-based attestation is the cryptographic approach: rather than using Intel's EPID group signature scheme, the Quoting Enclave employs ECDSA. This choice is strategic: ECDSA signatures can be validated independently without needing real-time verification from Intel's servers. The certification workflow operates as follows: the Quoting Enclave submits its attestation public key to the PCE for certification. The PCE responds by creating a signed certificate that cryptographically links the Quoting Enclave's identity with its public key, using the PCK for signing. To ensure the integrity of this decentralised certification model, Intel distributes certificate revocation lists to all platforms, maintaining an unbroken chain of trust.

2.3.5 Sealing

Upon instantiation, enclaves guarantee confidentiality and integrity by holding data within their protected boundaries. However, enclave developers must identify which sensitive data or state information must remain after the enclave's lifetime, particularly in three critical scenarios: voluntary enclave closure by the application, application termination, and system hibernation or shutdown. By default, secrets provisioned to an enclave are ephemeral and lost alongside enclave termination. When persistent storage of sensitive data is necessary for future enclave sessions, this information must be externalised before closure. SGX addresses this challenge through a sealing mechanism that enables communication between the enclave and the underlying platform. This key cannot be produced by any other enclave or on any other hardware. Using this key, the enclave can encrypt sensitive data before storing it externally (sealing) and subsequently decrypt it when needed (unsealing), ensuring that sealed data stays cryptographically bound to the original enclave and platform. By the way, SGX provides flexible key derivation policies via the EGETKEY instruction, allowing developers to choose how sealing keys are bound to enclave attributes. Two primary policies govern this binding mechanism: seal to the Current Enclave (Enclave Measurement) and seal to the Enclave Author.

Seal to the Current Enclave(Enclave Measurement)

The first approach, that is, a sealing based on measurement, cryptographically links the sealing key to the enclave's MRENCLAVE value, which is a hash computed during enclave creation, and represents its exact code and data. Hardware enforcement through EGETKEY ensures that only an enclave that owns an identical MRENCLAVE can derive the same unsealing key. Any modification to the enclave binary(through tampering or legitimate updates) alters this measurement and consequently changes the derived key, making previously sealed data inaccessible. While this provides strong integrity guarantees, it complicates software updates since each new version produces a different measurement.

Seal to the Enclave Author

This approach, that is, an author-based sealing, offers greater flexibility by binding keys to the enclave signer's identity (stored in MRSIGNER), combined with a Product ID, both of which are established during initialisation. This policy enables multiple benefits: enclave upgrades by the same author can access data sealed by previous versions without complicated migration procedures, and different enclaves from the same author can share sealed data.

To manage security across versions, authors assign Security Version Numbers(SVN) to the enclave. During key derivation, an enclave selects an SVN value constrained by its ISVSVN (set at creation). The enclave may derive keys using its current ISVSVN or any lower value,

but cannot derive keys associated with higher version numbers. This mechanism also supports backward compatibility for software updates: in fact, a newer enclave version can unseal data from earlier versions while preventing older, potentially vulnerable versions from accessing data sealed by newer ones. The SVN system allows controlled data migration during upgrades while preserving security boundaries across the software lifecycle.

Chapter 3

Digital Forensics

3.1 Fundamentals of Digital Forensics

3.1.1 What is Digital Forensics?

Digital forensics, also known as Computer Forensics, encompasses the scientific methods and techniques used to investigate cybercrimes, security incidents, and malicious activities involving digital systems. Law enforcement agencies, judicial systems, and organisations rely on digital forensic practitioners to identify, preserve, analyse, and present digital evidence following procedures in accordance with legal requirements.

The discipline emerged in the 1970s and 1980s when investigators recognised that digital storage media could contain crucial evidence for criminal investigations. Previously, examining suspect documentation was a time-intensive manual process. The advent of computer systems created both new investigative opportunities and challenges. In 1984, the Federal Bureau of Investigation established the Magnetic Media Program, representing the first official federal-level digital forensics initiative specifically designed to address crimes related to computer activity [25].

The concretisation of digital forensics as a distinct discipline was accelerated by significant cases and the creation of professional organisations. The case of 1989 United States vs Robert Tappan Morris, which led to the first conviction under the Computer Fraud and Abuse Act after the release of the Morris Worm, represented the first extensive application of the methodology of digital forensics for the analysis of computer logs and network activity in a criminal prosecution. [26]. That same year, the International Association of Computer Investigative Specialists (IACIS) was founded to develop guidelines for the acquisition and processing of computer evidence. This was followed by the creation of the International Organisation on Computer Evidence (IOCE) in 1995, which provided a collaborative forum for law enforcement agencies to exchange information on computer investigation techniques and forensic procedures [27].

The field has undergone rapid technological evolution over the subsequent decades. The 1990s saw the emergence of foundational forensic tools such as EnCase, which established standardised methodologies for the examination of digital evidence [28]. During the 2000s, digital forensics capabilities were widely adopted by law enforcement agencies worldwide. The 2010s introduced new challenges with the growth of cloud computing, the proliferation of mobile devices, and the application of automation and machine learning to support forensic investigations. More recently, 2020 introduced advanced challenges, such as high-level cryptographic schemas, blockchain forensics, and the integration of large-scale language models, which improve the effectiveness of digital forensics investigations.

The dependency of contemporary society on advanced technological computers, like cloud computing, IoT ecosystem, mobile devices, and network interconnectivity, has expanded in a significant way the panorama of digital threats. Cybercriminals exploit these technologies through various attack vectors, including identity theft, distributed denial-of-service attacks, malware

distribution, data breaches, and social engineering. These incidents inflict substantial harm on governments, organisations, and individuals [25].

Consequently, digital forensics has a fundamental role because it guarantees that the digital evidence remains unchanged from the collection phase until the final presentation in a courtroom, providing at the same time to the investigators, methodologies able to adapt to technological platforms in continuous evolution and criminal techniques always more sophisticated.

3.1.2 Digital Forensics Terminology and Relevant Concepts

Digital Evidence

The Scientific Working Group on Digital Evidence defines digital evidence as "information of probative value that is stored or transmitted in binary form" [29]. This broad characterisation encompasses data from various sources beyond traditional computers, including telecommunications equipment and multimedia devices. Moreover, the scope of digital evidence extends to all criminal investigations in which electronic data may be relevant, rather than being limited to cybercrime cases such as network intrusions or hacking incidents [30].

An alternative perspective is offered by the Australian Standards document HB171, "Guidelines for the Management of IT Evidence", which characterises IT evidence as "any information, whether subject to human intervention or otherwise, that has been extracted from a computer. IT evidence must be in a human-readable form or able to be interpreted by persons who are skilled in the representation of such information with the assistance of a computer program". However, this framework exhibits significant limitations by focusing exclusively on computer-based systems, thereby overlooking the increasing importance of non-traditional digital devices in forensic investigations [30].

However, collecting digital evidence presents significant challenges. Raw data must be interpreted and contextualised to establish its relevance to real-world events. The volatile nature of digital evidence requires careful handling, while its identification requires specialised technical knowledge. Therefore, expertise in file systems, network protocols, and encryption methods is fundamental to effective evidence collection.

Chain of Custody

The chain of custody constitutes a fundamental pillar of digital forensics investigation, acting like a mechanism to provide integrity of the evidence and, at the same time, as a complete protocol of documentation. This procedural framework guarantees that the evidence maintains its authenticity and admissibility from the initial discovery until the presentation in court.

Formally defined, the chain of custody is a systematic procedure that documents the chronological sequence of evidence handling throughout an investigation, addressing the fundamental questions of where, when, why, who, and how the evidence was managed at each stage [31]. The framework includes all the persons that are directly involved in the acquisition, collection and analysis of the evidence, including the case identifiers and the bodies responsible for processing.

However, digital evidence has unique characteristics that significantly complicate the maintenance of the chain of custody compared to physical evidence:

1. Digital data could be replicated and transferred instantaneously. It is easy to duplicate and transmit them.
2. The evidence could be altered without leaving any evident traces.
3. Data can be easily contaminated, and they are often sensitive to time.
4. Digital evidence frequently crosses international boundaries and different legal systems.
5. Unlike physical evidence, digital data requires technical interpretation to be properly understood and given meaning.

Interactions with physical evidence are relatively straightforward to document. In contrast, the ability to remotely access, copy, and transfer digital data, combined with investigator mobility, enables evidence examination across multiple locations and times. This complexity increases the difficulty of accurately tracking all investigative activities.

To be accepted in court, the chain of custody must demonstrate complete accountability by identifying all individuals who accessed the evidence, maintaining accurate time records, and tracking the evidence's geographic location throughout the investigation. Rigorous access controls and audit mechanisms must ensure that only authorised personnel interact with the evidence, ultimately providing demonstrable proof that it remained unaltered throughout all investigative stages [32].

Figure 3.1. Chain of Custody

Data acquisition

Data acquisition is the process of collecting digital evidence from devices without altering or damaging the original data. This process requires specialised expertise in forensic acquisition and analysis tools, such as FTK Imager, EnCase, and Volatility [33], to ensure proper evidence preservation.

Data acquisition is a crucial part of digital forensics, as its absence can compromise the success of a criminal investigation. However, the practice of collecting all the digital data available indiscriminately, nowadays, is always an object of discussion, in particular regarding the justification and proportionality in different investigative contexts. Recent debate highlights the intrusive nature of full data acquisition processes and their implications for the right to privacy, calling for the need to strengthen safeguards protecting fundamental rights in such practices [34].

Pollitt [35] highlights a fundamental constraint: digital forensics experts cannot examine all available data. Accepting this limitation requires a strategic reconsideration of data acquisition methodologies.

If forensic examiners cannot (and in some contexts should not) analyse the entire body of digital evidence, then the practice of acquiring complete datasets during the examination process seems counterintuitive. This approach raises several concerns: it introduces legal and ethical complications regarding the scope of data processing, reduces operational efficiency within forensic workflows, and increases the likelihood of interpretative errors and oversight of relevant evidence due to ineffective data management and cognitive overload.

Forensics Image

An image is a "bit-by-bit copy of the evidence (hard drive, USB device, shared network folder, etc" [36], which is an exact replica of the original device.

Hashing

Hashing represents a fundamental technique for managing large-scale data investigations, serving the dual purposes of validating data integrity and identifying known content. Defined as "the

transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string" [36], hashing provides cryptographic assurance of data authenticity throughout the forensic process. The integrity verification mechanism operates through one-way cryptographic algorithms applied during the creation of the forensic image of the evidence. When the hash values calculated before and after the acquisition process match, this cryptographic match proves that no alteration occurred during evidence collection, thus establishing the authenticity of the data and its admissibility in court.

In the context of digital forensics, the cryptographic functions of the SHA hash standard provide a level of assurance adequate for the preservation of evidence integrity. The secure model is not founded exclusively on the cryptographic robustness of the hash algorithm, but mostly on the rigorous application of the chain of custody procedures, finalised to prevent unauthorised access and improper modification. The primary threat vector is the possibility of a malicious actor simultaneously compromising both the evidence and the corresponding hash values, thus compromising the entire verification mechanism.

Contemporary forensic practice employs several specialised tools that implement hash-based integrity verification mechanisms.

Command-line utilities such as sha256sum allow direct calculation of the hash value. While other forensics platforms are more complete, like FTK Imager, Autopsy and The Sleuth Kit, they have integrated the hashing functionality into the flow of work, more broadly dedicated to the elaboration and analysis of the digital evidence. EnCase, a widely adopted commercial solution, implements multiple hashing algorithms, including MD5, SHA-1, and SHA-256, to establish and verify evidence integrity throughout the investigative process [33].

3.2 The Forensics Process

Digital forensics investigation constitutes a rigorous and systematic procedure that requires careful observation of the protocol established. The investigation outcome, positive or negative, depends decisively on rigorous adherence to standardised methodological frameworks. Contemporary digital forensic practice typically encompasses 4 principal phases that structure the investigative workflow [37]. Procedural variations may occur based on contextual requirements and organisational constraints; institutional policies, guidelines, and procedures should formally document any deviations from standardised forensic protocols:

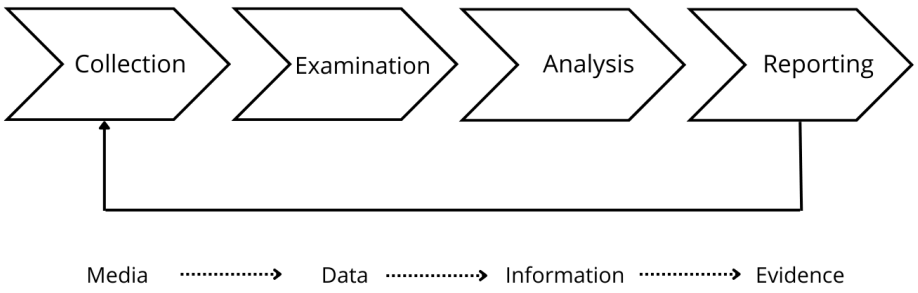

Figure 3.2. Forensics Process

Step 1 - Collection:

The collection phase constitutes the initial stage of a digital forensic investigation, in which investigators systematically identify all potential sources of digital evidence relevant to the case and acquire it from them. These systems integrated an internal optical unit for supporting CD

and DVD, enhanced by multiple ports of external interfaces, like USB, FireWire, and PCMCIA, that allow the connection of devices of external storage. Digital evidence, moreover, can be found within external entities, like Internet Service Providers (ISPs), that keep network log activity. Access to this data held by third parties generally requires legal authorisation through court orders, introducing procedural complexity and temporal delays into the investigation process. For any recorded data, its integrity should be preserved. [37].

Following a complete enumeration, the experts in digital forensics should evaluate which of the identified devices requires a detailed analysis to satisfy the investigative objectives. [33]. After the identification of the source, the investigators must proceed to the acquisition of the data through a precise methodology divided into 3 phases: acquisition planning, data acquisition and verification of the integrity of the acquired data. While this framework remains substantially unchanged between different surveys, the operating modes of the acquisition and verification phases must necessarily adapt to the different types of data and various storage formats. Cryptographic integrity verification constitutes a mandatory procedural step. Demonstrating conclusively that acquired evidence remains unaltered becomes critically important when the data may serve as courtroom evidence, as judicial proceedings require proof of data authenticity and freedom from tampering throughout the custody chain.

Step 2 - Examination:

After data has been collected, the next phase is to examine the data, which "involves assessing and extracting the relevant pieces of information from the collected data" [37]. The examination phase begins after data collection and focuses on identifying and extracting information relevant to the investigation from the acquired dataset. It typically consumes the majority of investigative resources, as well as the Analysis process discussed later, requiring deployment of numerous specialised techniques to access, locate, and extract acquired data into formats interpretable by humans [38]. The technical complexity arises from the need to circumvent or neutralise obfuscation mechanisms, including cryptographic protections, data compression techniques, and controls dictated by the Operating System.

The procedures of forensic examination, particularly within law enforcement contexts, must prioritise evidence preservation. Investigative protocols must eliminate or substantially minimise any possibility of modifying original evidence during analysis [39]. This principle requires operating exclusively on forensic copies, rather than on the original media, implementing write-blocking mechanisms and maintaining complete records (audit trail) of all analysis operations performed.

Step 3 - Analysis:

While the Examination phase focuses on extracting potential digital evidence from acquired data, the Analysis phase involves the methodical examination of this evidence and the reconstruction of events related to the incident. Analysis applies scientific methodology and critical thinking to address fundamental investigative questions: who, what, where, when, how, and why [38]. After extracting the relevant information, investigators enter the analysis phase, during which they conduct a methodical examination of the isolated data in order to formulate investigative conclusions and verify the facts.

The primary goal of the analysis phase is to enable investigators to reconstruct data fragments based on their relevance for the purposes of the evidence and determine the root cause of the incident. This phase typically represents the most time-consuming stage of the investigative process. This phase is typically the most time-consuming stage of the entire investigative process. Given the high volume, variety, and complexity of data that characterise actual digital investigations, evidence analysis shows significant challenges. Investigators should employ certified automated techniques to complement manual validation methods, thereby accelerating the analysis process and, at the same time, maintaining forensic integrity.

The analysis consists of the identification of the relevant entities (persons, places, items and events) and mapping their relationships in order to reach a conclusive investigation. This typically

requires correlating evidence across multiple data sources. For instance, network IDS logs might identify a compromised host, system audit trails could link activity to a specific user account, and host IDS data may reveal the actions performed under that account. [37]

Step 4 - Reporting:

The Reporting phase constitutes the culmination of forensic investigation, wherein findings are communicated to stakeholders in a manner that ensures clarity, accuracy, and legal admissibility. This phase involves communicating investigative conclusions to audiences beyond the examination team, including judicial bodies, corporate leadership, or other entities who will evaluate the evidence and determine appropriate action [40]. Reporting "is the process of preparing and presenting the information resulting from the analysis phase" [37].

The presentation deliverable comprises a comprehensive report documenting the complete investigative workflow, from initial evidence identification through final analytical conclusions. The essential components include: an accurate documentation of the chain of custody used to demonstrate the provenience of the evidence, chain of evidence records that demonstrate data integrity throughout all the phases of the processing, a detailed description of the investigative methodologies adopted, and the expert interpretation and opinion of an examiner, prepared for judicial or administrative consideration[40]. All additional documentation collected during the investigation that may influence the evaluation processes must be included in the final submission report to ensure full transparency and allow independent verification of the investigative procedures adopted.

Reporting responsibilities include identifying systemic weaknesses such as policy gaps or procedural errors that require correction. In addition to addressing any immediate gaps, digital forensics professionals must pursue ongoing professional development. Many certification bodies require periodic training updates on contemporary tools and techniques, storage technologies, and data format specifications in order to maintain accreditation status.

3.3 Standards and Methodologies

Digital forensics investigations require adherence to established methodological frameworks to ensure scientific rigour and legal admissibility. The development of formal protocols for analysing and documenting evidence has a dual purpose: first, supporting legal proceedings and second, enabling internal corporate investigations. [41].

The research has demonstrated that the combination of more forensic standards yields superior investigative results, rather than relying on a single framework. [42]. This integrated method allows for balancing the intrinsic limit of single standards, reducing risks in handling evidence and enhancing overall information security practices.

The selection of the most appropriate standards depends on the specific features of each investigation. Among the frameworks most adopted worldwide, the NIST Special Publication 800-86 and the ISO/IEC 27037 each address distinct aspects of the forensic process.

NIST SP 800-86 SP 800-86, which is published by the National Institute of Standards and Technology, presents a systematic methodology for digital investigations [37]. The framework defines six sequential phases: identification, preservation, collection, examination, analysis, and reporting. A central requirement is maintaining a complete chain of custody document, which ensures that every interaction with the evidence can be verified and traced.

ISO/IEC 27037 This international standard focuses specifically on the initial handling of potential digital evidence [43]. It provides guidelines for four key activities: identifying digital evidence sources, collecting materials without altering their state, acquiring forensic copies using validated methods, and preserving evidence to prevent unauthorised modification. The standard emphasises maintaining both integrity and authenticity from the moment evidence is discovered.

Benefits of Standards Adoption Implementing forensic standards offers several advantages. Moreover, standardised procedures guarantee that different investigators, during the examination of the evidence, follow coherent and uniform methodologies, and that is a fundamental element both for the reproducibility of the results and for the acceptance in a court.

However, the adherence to consolidated protocol reduces the likelihood of procedural errors that could compromise the integrity of the evidence. When investigators follow prescriptive methods, the results they obtain acquire greater solidity and weight in legal proceedings.

Finally, common frameworks facilitate professional collaboration across organisational and jurisdictional boundaries. Shared terminology and methodologies enable practitioners to exchange knowledge effectively and adapt to emerging technologies and threat landscapes.

3.4 Digital Forensics Tools

The exponential growth in digital data generation and storage has created a corresponding increase in vulnerability to malicious exploitation. Information residing on mobile devices, computer systems, and embedded controllers faces persistent threats from diverse attack vectors, necessitating robust investigative capabilities. Among the most significant advancements in the field has been the emergence of specialised software applications designed specifically for forensic examination [44]. These tools have substantially changed the investigators' approach to digital evidence, providing a systematic method for recovering evidence while ensuring its evidentiary integrity. A defining characteristic of forensic software is the implementation of write-protection mechanisms that preserve source data in its original state throughout the examination process. This capability enables verification of recovered content against original artefacts, ensuring that extracted information remains uncontaminated by the investigative procedure. The applications of digital forensic tools extend to various domains of criminal and civil investigation, including financial fraud detection, privacy violation cases, unauthorised system access to computer systems, and different forms of electronic harassment or exploitation, where each digital artefact constitutes primary evidence. Fundamental operational principles regulate the correct use of these instruments. In the first place, the acquired data must remain unvaried with respect to its original state. Secondly, the experts must maintain a complete and detailed documentation of all the investigative activities performed. Third, access to original evidence sources requires strict controls to prevent inadvertent or intentional alteration [44].

The concept of a "tool" in the context of digital forensics requires careful definition, as various authors have proposed distinct interpretations. A comprehensive characterisation emphasises the self-contained nature of forensic tools and their capacity to abstract complex operations from the end user [45]. Effective forensic tools minimise required user intervention by automating low-level technical processes. Professionals should not need to manually calculate the position of a disk sector or perform address translation operations, because such complexity must be handled transparently by the tool itself. The scope of what can be considered a valid forensic tool is deliberately broad. As examples, the most widespread are parsing scripts for file system analysis, file carving implementations that reconstruct fragmented data, and all the applications used to represent complex information in different, easily interpretable formats. It does not prescribe any specific language or development methodology; in fact, the tools could be developed by individual researchers, academic groups, and commercial entities. Furthermore, tools may incorporate or orchestrate other software components provided that such integration occurs automatically. Autopsy, a forensic platform, illustrates this principle through its architecture, which is based on plugins: each extension module, despite operating within a larger framework, can be considered independently as a forensic tool because of its discrete analytical contribution [45].

Artificial intelligence techniques, particularly those related to machine learning and deep learning, are having a growing influence on the practice of digital forensics. These computational approaches offer significant advantages in the elaborations of datasets on a large scale, providing analytical results with speed and accuracy that would be difficult to reach through manual examination alone. Consequently, AI–enhanced analysis has become an essential capability for forensic professionals managing continuously expanding volumes of digital evidence. However, the

diffusion of the available forensic tools has its own challenges. The high number of different software solutions, each with various capabilities, interfaces, and intended use cases, can overwhelm experts who are attempting to identify the most appropriate solution for a specific investigation [46]

Digital forensic tools comprise specialised software applications developed to support investigative procedures in cases where are involved the digital evidence of criminal activities. The current marketplace offers a diverse ecosystem of such instruments, including both proprietary commercial products requiring licensing fees and freely available open-source alternatives [46]

Computer Forensics Tools Computer forensics is "the practice of collecting, analysing, and reporting on digital data in a way that is legally admissible. It can be used in the detection and prevention of crime and in any dispute where evidence is stored digitally" [39].

Computer forensic tools constitute a specialised category of digital forensic software designed to ensure that evidence extracted from computer systems maintains both accuracy and reliability throughout the investigative process. These tools must guarantee that all recovered data faithfully represents the original source material. This requirement is essential for establishing evidentiary validity in the judicial branch.

The functional capabilities of computer forensic tools typically encompass several core operations. Disk imaging is the process of creating an exact, bit-by-bit copy of a storage medium, including all data on it, such as deleted files and unallocated areas. Cryptographic hashing generates unique fingerprints about data that are used to verify integrity during the acquisition and analysis phases. Finally, data recovery mechanisms are used to reconstruct deleted or damaged files from various filesystem structures [46]. The commercial and open-source landscape includes numerous established platforms such as EnCase, Forensic Toolkit (FTK), ProDiscover, Autopsy, and Stellar, among others [33] [46]. Comparative evaluations of these tools frequently assess parameters including supported filesystem types, hashing algorithm implementations, imaging capabilities, and evidence recovery effectiveness. For the purposes of this thesis, The Sleuth Kit was selected as the foundational forensic analysis component.

3.4.1 The Sleuth Kit (TSK)

The Sleuth Kit (TSK) is a foundational open-source toolkit used for digital forensic investigations. It was originally developed by Brian Carrier for UNIX-based operating systems, including Linux, macOS, FreeBSD, OpenBSD, and Solaris. The toolkit offers analytical capabilities for different file system formats, such as NTFS, FAT, UFS, EXT2, and EXT3. In its initial release, the software was designated as The @stake Sleuth Kit (TASK) and included command-line utilities derived from The Coroner's Toolkit (TCT) codebase. TSK allows forensic examiners to analyse storage media using a non-intrusive methodology that operates independently of the target system's native operating system. This approach allows the examination of deleted and hidden content inside different schemes of partitioning DOS, BSD, Macintosh, Sun, and Linux. The toolkit processes disk images created using the `dd` utility, a standard UNIX command also available for Windows platforms, storing data in non-proprietary formats [47].

The Autopsy Forensic Browser works as the graphical front-end for TSK, presenting analytical results through a web-based interface. This complementary tool is used to automate various forensic operations, such as integrity verification, keyword searching, and partition analysis. The relationship between these two components defines a layered architecture, in which TSK execute the analysis at a low level of the file system, while Autopsy provide functionality to visualise output in an intuitive way and also instruments to handle the workflow. The dual interface design, with TSK operating via command-line and Autopsy through a graphical interface, accommodates diverse user preferences and operational needs in forensic laboratories [45].

From an implementation perspective, TSK was developed primarily in C and Perl, incorporating portions of the original TCT source code. The toolkit has been validated across numerous platforms, including Linux, macOS, OpenBSD, FreeBSD, Solaris, and CYGWIN. Its command-line interface design allows for minimising memory consumption and processing overhead with

respect to graphical alternatives, making it suitable for forensic workstations with limited resources. [47].

The forensic community recognises TSK as a reliable tool for extracting and analysing evidence from digital storage devices. Comparative evaluations have demonstrated that open source tools such as TSK produce results equivalent to commercial alternatives like EnCase and FTK, although they present different levels of operational complexity. [45]. The open-source forensic ecosystem has expanded significantly with complementary tools such as Helix contributing to this domain [48]. Furthermore, both TSK and Autopsy support extensibility through plugins mechanism, allowing professionals to integrate additional analytical features based on the specific needs of the investigative scenario. [45]. Beyond its standalone applications, TSK serves as the underlying analytical engine for numerous other forensic tools in both open-source and commercial domains, demonstrating its architectural versatility and community adoption [49].

The Sleuth Kit Architecture

TSK implements a hierarchical architectural model comprising multiple abstraction layers, each addressing distinct aspects of storage media analysis.

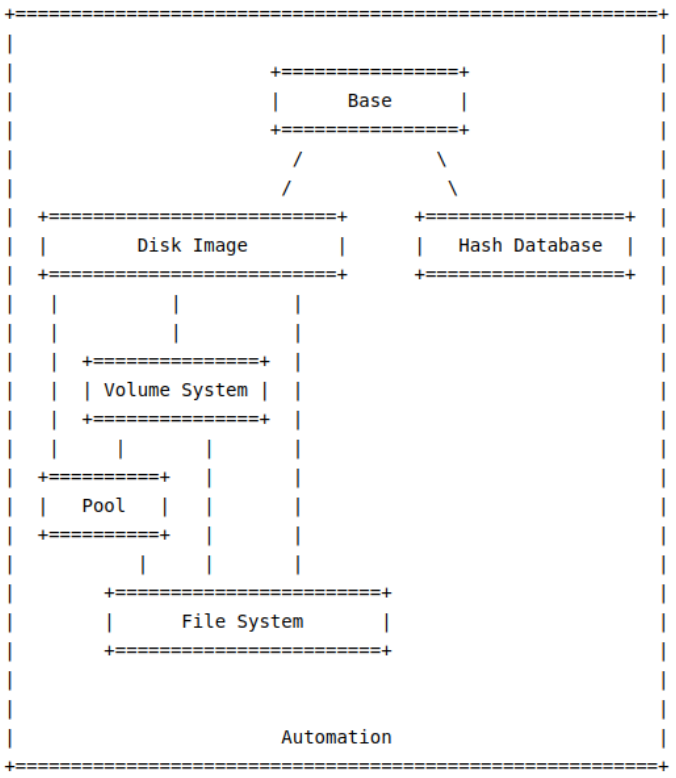

Figure 3.3. TSK Architecture (Source: The Sleuth Kit [50])

At the foundation is placed the Base Layer, which provides common programming constructs, data structures, error handling mechanisms, and utility functions that are shared by all higher layers. This layer also includes cryptographic hash computation capabilities for the MD5 and SHA-1 algorithms.

The Disk Image Layer established the next abstraction level, which is responsible for opening and processing disk images in various formats. This layer abstracts the complexities that are associated with image files split, compressed, and encrypted, presenting a unified interface to subsequent layers. All disk images must be initialised through this layer before further processing can occur.

Above this, the Volume System Layer handles volume system structures such as DOS partition tables and BSD disk labels. When analysing disk images containing volume systems, this layer identifies partition boundaries by determining the start and end sector addresses for each volume. The Pool Layer, introduced to support Apple File System (APFS), manages non-consecutive block allocations organised into logical volumes. Pools may span entire disks or reside within individual partitions, though most conventional disk images do not utilise this layer.

The File System Layer represents a critical component of the architecture that acts by providing all the necessary functionality for processing file system structures, including FAT, NTFS, and extended file systems. TSK organises file system data into five conceptual categories: File System metadata, Data Units, Metadata structures, File Name entries, and Application-level data. This categorisation enables systematic analysis of file system artefacts regardless of the underlying format. When files are deleted, their associated data units, metadata structures, and file name entries are marked as unallocated and become available for reuse. Since these components may be reallocated independently at different times, forensic examiners must exercise caution when interpreting data associated with deleted files.

Independent from the storage hierarchy, the Hash Database Layer manages hash databases such as the National Software Reference Library (NSRL) and standard hash output formats. This layer creates indexed structures from text-based hash files, enabling efficient binary search operations for rapid lookup of known good and known bad file signatures.

Finally, the Automation Layer integrates all preceding layers to facilitate automated forensic workflows and batch processing operations [50].

3.5 FAT32 File System

Among the file systems supported and analysed by TSK, a detailed description of the FAT32 architecture is provided here, as this knowledge is required for the development of the custom FAT32 parser implemented in the Secure Mode of the proposed framework. All the information presented in the following sections are based on [51].

3.5.1 Introduction

A filesystem is typically defined as the entire system for managing data stored on a storage device. It was developed in the 1980s as a simple filesystem available for floppy disks less than 500 KB in size.

FAT(File Allocation Table), which stands for File Allocation Table, refers both to the array used to manage the allocation of the `data area` and to the name of the filesystem itself. Actually, there are three types of FAT: FAT12, FAT16, and FAT32, and each of the latter versions is compatible with the previous one.

Sector: is the smallest unit of data on a storage device, used for reading and writing operations. The sector size is typically 512 bytes. Two types of sector addressing are used:

- Sector Number(relative sector number): indicates the position of a sector relative to the beginning of the volume (i.e., the partition). It starts from zero at the beginning of the partition.
- Physical Sector Number (absolute sector number): indicates the position of a sector relative to the beginning of the entire storage device (i.e., the entire disk).

This distinction is important because a volume (partition) does not always start at the beginning of the disk. Typically, there is a reserved area before the first partition (such as the partition table), meaning that the same sector will have a different number depending on whether it is addressed relative to the partition or to the entire disk.

For example, if a partition starts at physical sector 2048 of the disk, the first sector of that partition has a sector number of 0 (relative to the partition) but a physical sector number of 2048 (relative to the entire disk)

FAT Volume: A FAT filesystem is referred to as a logical volume or logical drive. Each FAT32 volume consists of three areas, and each area consists of one or more sectors:

- `Reserved area` contains the volume configuration data
- `FAT area` contains one or more copies of the File Allocation Table, which manages the allocation of clusters in the data area
- `Data area` contains the actual content of files and directories, organised in clusters.

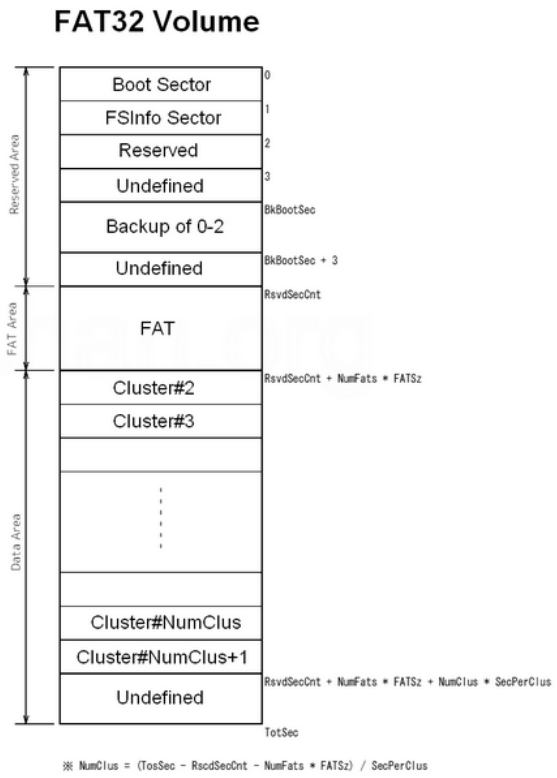

Figure 3.4. FAT32 Volume. (Source: ELM-chaN org[51])

All data structures in the FAT filesystem are stored on disk in little-endian format. If the platform's architecture for accessing the FAT filesystem is big-endian, an endian conversion is required.

Volume Parameters: The offset and size of each area are calculated from the parameters in BPB.

- `FAT Area` offset and size:

```
FatStartSector = BPB_ResvdSecCnt;
FatSectors = BPB_FATSz * BPB_NumFATs;
```

- `Data Area` offset and size:

```
DataStartSector = FatStartSector + FatSectors;
DataSectors = BPB_TotSec - DataStartSector;
```

3.5.2 Reserved Area

Boot Sector and BPB: The BPB (BIOS Parameter Block) is the most important data structure because it stores the configuration parameters of the FAT volume. The BPB is placed in the boot sector, which is the first sector of the reserved area and the first sector of the volume. The boot sector is valid only if the boot signature (`BS_Sign`) is `0xAA55`; any other value indicates that the boot sector should be considered invalid.

`BPB_TotSec32`, that is, the volume size or the total number of sectors of the FAT volume, is the number of sectors, including all three areas of the volume. If this value exceeds the actual size of the containing partition or storage device, it indicates that the FAT volume is in a serious state: corrupted or incorrectly created. Any operation on such a volume can result in catastrophic data loss, so the FAT driver should reject the volume if it detects this condition.

3.5.3 FAT Area and Data Area

The FAT defines a linked list of file extents. known as the cluster chain, which tracks which clusters belong to a specific file, since a file could be fragmented into multiple clusters that are not continuous on disk.

For example, when a file needs more than one cluster, the FAT links these clusters sequentially in a chain: the file starts at cluster 5, the FAT entry for cluster 5 points to cluster 9, which in turn points to cluster 12, until the end-of-chain marker.

In the FAT32 filesystem, there is no structural distinction between a file and a directory. They are treated in the same manner by the FAT. The only difference is that the directory is a file with a special attribute (texttttATTR_DIRECTORY) that indicates its contents are not ordinary data but a table of directory entries, specifically a list of files and subdirectories.

Typically, the top two FAT items, `FAT[0]` and `FAT[1]`, are reserved and not associated with any cluster. The third FAT item, `FAT[2]`, corresponds to the first data cluster, and the valid cluster number starts at 2. The FAT is essentially a table where each element(entry) corresponds to a cluster in the `data area` of the disk. Each entry's value indicates the state of the cluster.

FAT is usually duplicated for redundancy, because damage to any FAT sector results in serious data loss. This number is indicated by `BPB_NumFATs`, and as a consequence, the size of the FAT area is `BPB_FATSz * BPB_NumFATs`.

Data Area: It is divided into clusters, which are blocks composed of a certain number of sectors (`BPB_SecPerClus`), and the data area is managed at this level.

Determination of FAT Type

The FAT type is determined by only the counts of clusters on the volume:

```
CountofClusters = DataSectors / BPB_SecPerClus;
```

If the volume with a count of clusters is ≥ 65526, that means it is a FAT32. This is the only way to determine the FAT type.

Accessing FAT Entries

First, determine where the FAT entry is located in the FAT. It's simple in FAT32 because it's an integer array. The location of the FAT entry `FAT[N]`, the `sector number`, and `byte offset` in the sector can be obtained by following the calculation:

```
ThisFATSecNum = BPB_ResvdSecCnt + (N * 4 / BPB_BytsPerSec);
ThisFATEntOffset = (N * 4) % BPB_BytsPerSec;
```

Each FAT entry in a FAT32 volume occupies 32 bits, but only the lower 28 bits are valid: the upper 4 bits are reserved and must not be modified during normal operation. These reserved bits are initialised to zero when the volume is formatted and should remain unchanged. Therefore, when reading a FAT entry, the upper 4 bits must be masked out using a bitwise AND with `0x0FFFFFFF` to obtain the correct cluster value. Conversely, when writing a value back to a FAT entry, the upper 4 bits of the existing entry must be preserved and not overwritten.

```
/* Store a value of FAT32 entry */
ReadSector(SecBuff, ThisFATSecNum);
tmp = *(uint32*)&SecBuff[ThisFATEntOffset];
tmp = (tmp & 0xF0000000) | (NewEntryVal & 0x0FFFFFFF);
*(uint32*)&SecBuff[ThisFATEntOffset] = tmp;
WriteSector(SecBuff, ThisFATSecNum)
```

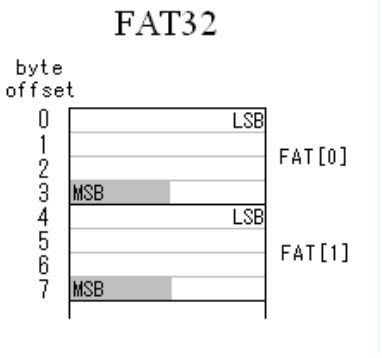

Figure 3.5. FAT32 FAT[0] and FAT[1] structure. (Source: ELM-chaN org[51])

Association of File and Cluster

Files on the FAT volume are managed by a directory, which is an array of 32-byte directory entry structures. The directory entry has the file name, file size, timestamp and the first cluster number of the file data. The cluster number is the entry point for following the file's data cluster chain. If file size = 0: the first cluster number is assigned 0, and no data cluster is allocated to the file.

As explained before, clusters 0 and 1 are reserved, and the first valid cluster is 2, which corresponds to the first cluster in the data area. As a consequence, the valid cluster numbers are from 2 to N+1, and the number of FAT entries is N+2, where N is the total number of clusters. The location of a data cluster N is:

```
FirstSectorofCluster = DataStartSector + (N - 2) * BPB_SecPerClus;
```

The cluster chain cannot be followed backwards. The FAT entry with the last link of the cluster chain has a special value, called End of Chain (`EOC`) mark, which never matches any valid cluster number. In the case of FAT32, this value falls in the range:

```
FAT32: 0x0FFFFFF8 - 0x0FFFFFFF (typically 0x0FFFFFFF)
```

There is also a bad cluster mark indicating a defective sector in the cluster; it cannot be used and has a value `0x0FFFFFF7` for FAT32.

In FAT32, the maximum number of clusters is not defined by the standards, and in theory, a valid cluster could have the value `0x0FFFFFF7`. In that case, the number could be either a valid cluster or a bad-mark cluster, creating ambiguity and confusion in the management of the disk. To avoid this problem, a practical limit is set on the number of clusters in a FAT32 volume, and that limit is 268435445 (approximately 256 M clusters).

The initial value of each allocatable FAT entry from FAT[2] is zero, indicating that the cluster is not in use and free for new allocation. If this value is nonzero, the cluster is either in use or

marked as bad. In FAT32, an additional structure called `FSInfo` is present, which stores the count of free clusters in order to avoid a full FAT scan, which would otherwise be very costly given the potentially large size of the FAT structure. The first FAT entries are assigned as follows:

- `FAT[0] = 0xFFFFFF??`: "??" has the value of `BPB_Media` . FAT[0] has only an informative purpose and no function in the file system.
- `FAT[1] = 0xFFFFFFFF`: Its bits are used to indicate flags of the state of the volume.
  - `Volume dirty flag` (bit31 in FAT32): if is **1** means a clean shutdown, therefore the volume was unmounted correctly. If **0** means dirty shutdown, it indicates a possible corruption in the logic of the file system.
  - `Hard error flag` (bit30 in FAT32): if **0** indicates there is an error of read/write that is unrecoverable during the use of the volume. This means that it is possible to inspect the disk surface to detect sector damage.

  These flags indicate the possibility of an error on the volume.

The last sector of a FAT cannot be fully used because, most of the time, FAT ends in the middle of the sector. This sector is filled with zeros during formatting and should not be modified afterwards.

`BPB_FATSz16/32` can indicate a value larger than the volume requires, meaning that unused sectors can follow each FAT. This may be a result of data area alignment or similar reasons, and these sectors are also filled with zeros during formatting.

FAT32	Meaning
0x00000000	Free
0x00000001	Reserved
0x00000002 -- 0x0FFFFFF6	In use (value is link to next)
0x0FFFFFF7	Bad cluster
0x0FFFFFF8 -- 0x0FFFFFFF	In use (end of chain)

Table 3.1. FAT32 cluster values and their meanings (Source: ELM-chaN org [51])

FSInfo Sector Structure and Backup Boot Sector

The size of FAT typically reaches several MB on a FAT32 volume. For this reason, FAT32 volumes support the FSInfo structure to avoid reading the entire FAT to find free clusters or count them. The structure below is the structure put inside the FSInfo sector indicated by `BPB_FSInfo`: The volume back boot sector is a feature of FAT32 that allows volume recovery in the event of corruption of the boot sector. The back location is indicated by `BPB_BkBootSec`, and is recommended to be 6. A copy of the FSInfo sector is also there.

FAT Directory

The directory, which is a file, contains a table of directory entries that contain metadata of the files on the volume. Each directory entry is 32 bytes long, and it corresponds to a file or directory on the volume. The maximum size of a directory is 2 MB, which means a maximum of 65536 entries.

Root directory: there is no difference between the root directory and the sub-directories, except that the root directory has no entry to indicate it and the start cluster number is indicated by `BPB_RootClus`; it does not contain dot entries (".", ".."), which always exist in the sub-directory, and it can contain a volume label (an entry with the `ATTR_VOLUME_ID` attribute). For more information, refer to the table in [51].

Among the fields of the table, the `DIR_Name` field deserves further analysis. `DIR_Name[0]` is the first byte of `DIR_Name`, and indicates the state of the directory entries. If this value is:

Field name	Offset	Size	Description
FSI_LeadSig	0	4	0x41615252. Lead signature used to validate that this is an FSInfo sector.
FSI_Reserved1	4	480	Reserved. It should always be initialised to zero.
FSI_StrucSig	484	4	0x61417272. Another signature is more localised in the sector to the fields that are used.
FSI_Free_Count	488	4	Last known free cluster count on the volume. If 0xFFFFFFFF, the value is unknown. The FAT driver must validate it.
FSI_Nxt_Free	492	4	Hint for the FAT driver: cluster number where to start looking for free clusters. If 0xFFFFFFFF, no hint is given and the driver should start at cluster 2. The FAT driver must validate it.
FSI_Reserved2	496	12	Reserved. It should always be initialised to zero.
FSI_TrailSig	508	4	0xAA550000. Trail signature used to validate that this is an FSInfo sector.

Table 3.2. FSInfo Sector Fields

- 0xE5: the entry is not in use and is free for new allocation.
- 0x00: the entry is not in use (same as 0xE5), but additionally indicates that there are no allocated entries after this one and all subsequent entries are also set to 0x00.
- Any other value: the entry is in use and represents an active file or directory.
- 0x05: a special case. If a filename genuinely starts with the character 0xE5, it is replaced with 0x05 to avoid being misinterpreted as a deleted entry.

It is an 11-byte string and is divided into two parts: 8 bytes for the body and 3 bytes for the extension. For example, a filename such as FILENAME.TXT is stored as FILENAME (8 bytes) followed by TXT (3 bytes), for a total of 11 bytes. For more information about how the file name format is handled, refer to the FAT directory section in [51].

Every file name is unique in the specific directory. The DIR_Attr field indicates the entry's attribute; e.g., ATTR_SYSTEM and ATTR_DIRECTORY are used to distinguish a system file from a directory.

Directory Operations

In the [51] is present a section where are described all the directory operations, that describe how to create a file, a sub-direcotory, deleting file (to remove a file is needed to set 0xE5 to the DIR_Name[0] to free the entry. If the file has a cluster chain, also the chain needs to be removed from the FAT) and deleting subdirectory. This section could be used for testing purposes for the framework.

Timestamp

There are also fields for time and date in the directory entry. The format of the time and date is described as follows:

Field name	Bit fields
DIR_WrtDate DIR_CrtDate DIR_LstAccDate	Bit 15-9: Count of years from 1980 in range of from 0 to 127 (1980-2107). Bit 8-5: Month of year in range of from 1 to 12. Bit 4-0: Day of month in range of from 1 to 31.
DIR_WrtTime DIR_CrtTime	Bit 15-11: Hours in range of from 0 to 23. Bit 10-5: Minutes in range from 0 to 59. Bit 4-0: 2 second count in range of form 0 to 29 (0-58 seconds).

Figure 3.6. Time and Data Format. (Source: ELM-chaN org [51])

Long File Name

LFN (Long File Name) information is recorded as a directory entry with a special attribute. The attribute assigned to an LFN entry is `ATTR_LONG_NAME`, defined as a combination of existing attribute bits: `ATTR_LONG_NAME = ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUM E_ID` A mask value is also defined as: `ATTR_LONG_NAME_MASK = ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID | ATTR_DIRECTORY | ATTR_ARCHIVE` For more information about the directory entry structure for LFN, refer to the Long File Name section in [51].

An LFN entry is always associated with a corresponding SFN (Short File Name) entry, as LFN entries never exist independently. The LFN entry contains only name information. When an LFN is assigned to a file, it becomes the primary name, while the SFN serves as an alternative. Systems that do not support LFN do not recognise LFN entries, but can still access the files using the SFN. If the LFN is longer than 13 characters, it is divided into multiple LFN entries. The

Location	First byte	Name field	Attribute	Content
DIR[N-3]	0x43	ary.pdf	--VSHR	LFN 3rd part (lfn[26..38])
DIR[N-2]	0x02	d System Summ	--VSHR	LFN 2nd part (lfn[13..25])
DIR[N-1]	0x01	MultiMediaCar	--VSHR	LFN 1st part (lfn[0..12])
DIR[N]	'M'	MULTIM~1.PDF	A-----	Associated SFN entry

Table 3.3. LFN Directory Entry Example (Source: ELM-chaN org [51])

maximum name length for LFN is 255, so that an LFN occupies up to 20 LFNs. entries. LFN resided before the associated SFN entry. LFN use as a character code Unicode in UTF-16 while SFN is ANSI/OEM code. Furthermore, a checksum is used to ensure relevance between LFN and SFN. Each LFN entry has a checksum of associated SFN in `LDIR_Chksum`; the checksum is generated as:

```
uint8_t lfn_checksum(const uint8_t *sfn) {
    uint8_t sum = 0;
    for (int i = 11; i; i--)
        sum = ((sum & 1) << 7) + (sum >> 1) + *sfn++;
    return sum;
}
```

If any checksum in the LFN entries does not match, the LFN is invalid.

Chapter 4

Trusted Computing for Digital Forensics

The integration of TEE with the digital forensics methodologies represents an emerging research area that face on critical challenges related to the integrity of the evidence and to the preservation of the chain of custody. Although this thesis proposes an innovative framework that combines Intel SGX and TSK to achieve secure analysis, the fundamental concept of leveraging hardware-based isolation for digital forensics is not entirely new. Several researchers have studied TEE technologies to improve the protection of evidence, the attestation mechanism, and the flow of secure analysis.

This chapter examines existing approaches that integrate the TEE with digital forensics, analysing both academic research and practical implementation. The study encompasses solutions that use different TEE architecture and their application to forensic problems, including the prevention of evidence tampering, the digital forensics on mobile devices in a secure manner, and the investigation on cloud environment and the maintenance of the chain of custody.

4.1 TEE-BI

The growing use of encrypted communication and anonymous services has, as a consequence, led to new regulations allowing law enforcement to conduct remote forensic investigations, in which police secretly obtain remote access to suspect computers to seize all unencrypted evidence. TEE-BI is a framework that is designed in order to demonstrate that the evidential value of the evidence obtained should only increase by "employing integrity verification techniques offered by secure hardware" and "exfiltrating the decryption key of encrypted communication only in order to decrypt communication obtained by lawful interception" [52]. TEE-BI is a solution for introspection based on TEE ARM TrustZone (TZ). It is built on Android hardware and demonstrates the ability to extract SSL encryption keys from an Android application running in user space, undetectably.

4.1.1 Remote Forensic Investigations

Encrypted communication and anonymisation services like Tor render it always more complicated for law enforcement to prevent the "going dark" phenomenon, where going dark is the expression used in the investigative environment, and it means that law enforcement is not able to access communication or data, also with a legal mandate, because of technologies such as end-to-end cryptography. One possible way to avoid this is to use remote forensic investigation. The idea is to use cyber techniques to access and install spyware on the target system: in this way, police forces can bypass anonymisation and extract data before encryption.

However, there are several disadvantages to current practices. First of all, installing spyware on a device without the device owner's consent is difficult. One possible way is to use Social Engineering, tricking the user into installing the application or revealing the password to access the system. Second, there is the problem of the evidential value of evidence obtained through remote forensics, because the target person can always say that the computer has been manipulated, and this is difficult to refute, since it is hard to provide proof. Third, it is difficult for law enforcement to intercept encrypted communication, because they must intercept the message before it is encrypted. The problem is that they should catch the item at the exact moment it is written or received. Moreover, under the new normative, like in Germany, only the interception of the content effectively sent to the network is allowed. Anyway, because of these technical uncertainties, the government spyware tends to collect more data than necessary. This could constitute a legal violation. TEE-BI is focused on the second and third problems. This raises the question of how to design remote forensic investigation tools that comply with general legal principles.

A model of collaboration between the State and device manufacturers is proposed, in which government software is integrated directly into the security architecture of mobile devices to enable targeted interceptions. In the article [52], they argue that remote forensic software should be embedded into the trusted computing base of end devices; in this way, it is easier to explain that the software is running as specified. Secondly, a variant solution of remote forensic surveillance is proposed. The idea is to extract only the minimal information as the cryptographic key to access the data network and intercept the communication. For this, TEE-BI is used, which uses ARM TZ to attest to the integrity of the software and can be used to extract the encryption key.

4.1.2 Design

TEE-BI is based on primitive architecture provided by ARMv8-A, which is the same architecture that manufacturers such as Qualcomm, Samsung, and Huawei use for their system design. Therefore, the hardware requisites for a system like TEE-BI are present in a huge number of devices worldwide.

Since TEEs are the foundation of disk encryption and the management of access to sensitive areas, we can use them to perform many actions in a trusted environment. As we said before, TEE-BI is a backed system that is used to extract just the minimal information from a target device, the encryption keys and use them to remove the cryptographic protection at the network level 4.1. This process is called exfiltration.

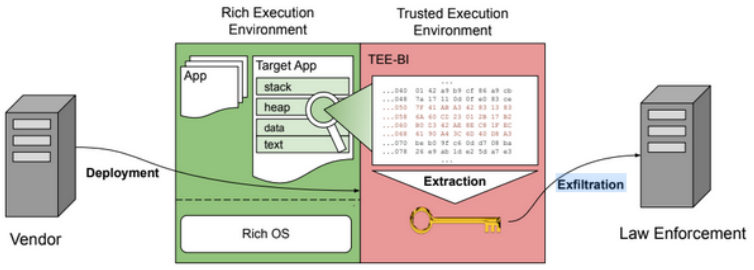

Figure 4.1. TEE-BI Architecture (Source: TEE-BI Article [53])

The three steps that make up TEE-BI are illustrated below.

1. Deployment:

The objective of the deployment process is to load TEE-BI in a trustworthy way within the TEE, using the most direct path possible and guaranteeing integrity and confidentiality. To provide EE-BI securely to the target device, two schemas can be used. The idea in the background is that

each mobile device connects regularly to update the server or other backend services provided by the device manufacturer, and this infrastructure can be exploited to deliver TEE-BI to the device's destination.

1. Direct access to the network of TEE: the TEE has direct access ot the networking interface of the device, allowing the download of the component without passing through the OS.

2. Using the stack network of the RichOS: it is a less complex and heavy diffused solution, which consists of using the stack network existing in the RichOS. For example, GlobalPlatform TEE provides detailed specifications on how to integrate and implement access to the network from TEE.

2. Extraction:

The TEE context data extraction process consists of two phases: identification of the target application and identification and extraction of relevant information.

Both phases rely on TEE introspection techniques. Specifically, TEE-BI leverages its unrestricted access to main memory and REE translation tables to reconstruct the virtual address space of the process running in the Rich OS and userland applications.

To locate the application of interest, TEE-BI traverses the REE's task list, identifying the identifiers associated with the target process. Once identified, all memory regions belonging to it are determined, allowing the complete reconstruction of its virtual memory space and the extraction of the desired data.

The cryptographic material is identified using a set of heuristics designed to recognise and extract keys from the BoringSSL library, the main cryptographic library used in Android systems.

3. Exfiltration:

Similar to the implementation phase, the exfiltration process requires a channel with reliable communication to a remote entity. This communication could be established through one of the modes described previously: the temporal control of the network interface from the TEE or by using the network stack of the REE. Independent of the schema adopted, the outgoing connection from the TEE to the remote server should be cryptographically protected to the extent necessary to guarantee the integrity and confidentiality of the extracted material during transmission. After completing the exfiltration of the cryptographic material, law enforcement could proceed to decrypt the network traffic already captured.

Data extraction

The mechanism for extracting cryptographic material from active TLS connections involves implementing a callback function that extracts the Master Secret and Client Random from a process using the BoringSSL library. It is assumed that the internal structure of BoringSSL data is known (or can be preliminarily analysed).

The necessary information is included in the SSL context's data structure. Identifying this structure in memory and searching for a digital fingerprint based on specific key length, the chyper identifier, and field alignment makes it possible to follow the offset and memory references to recover the desired value systematically.

Once the correct structure is identified, information extraction occurs directly through a controlled memory scan.

Limitations

The TEE-BI prototype was developed as a proof-of-concept to demonstrate the feasibility of TEE-based introspection and cryptographic extraction. In the prototype, extraction is initiated immediately after BoringSSL establishes a secure connection during the process.

In a real-world implementation, TEE-BI would instead employ a polling-based mechanism to perform periodic checks. Because TLS keys reside in memory only temporarily, the extraction window is limited. While it is not possible to guarantee that the key has not already been removed from memory, more frequent scans can increase the probability of success.

Finally, a secure timer is needed to trigger and ensure the extraction process runs without the Rich OS interfering with the process.

4.1.3 Assessment

Based on the results reported in the article [52], a table of TEE-BI performance metrics on the HikKey development board was compiled. On average, mapping a page takes approximately 6.95 s (48,713 s / 7008). As the article states, comparing this to the overall time spent on the page table walk, it can be concluded that the page mapping phase significantly impacts TEE-BI performance.

In particular, adjusting the memory mapping within TEE has a significant impact on performance. Currently, TEE-BI does not implement advanced caching mechanisms, thus leaving significant room for improvement in terms of optimisation.

From an architectural compatibility standpoint, the use of ARM processors, widely used globally in consumer mobile devices, makes the approach easily applicable. The solution can also be adopted in the embedded field, although it requires further adaptations.

From a legal point of view, the method presented offers advantages over traditional remote forensic telecommunications surveillance techniques, in particular in terms of authenticity, integrity, reliability, and verifiability, enhancing the evidentiary value of evidence in forensic environments.

4.1.4 Conclusions:

In conclusion, the article [52] proposes an innovative approach to remote forensic telecommunications surveillance that addresses two main critical issues with current practices: on the one hand, it increases the evidentiary value of remote investigations, and on the other, it enables legitimate surveillance of encrypted communications. To achieve these goals, remote forensic software is integrated into the Trusted Computing Base (TCB) and leverages ARM TrustZone features, such as the secure hardware-based root of trust. This approach is minimally invasive for the target device and represents one of the few solutions compliant with the legal principles of proportionality in force in Europe and the United States.

4.2 Securing System Logs With SGX

System logs are often one of the first targets of an attacker after a system compromise. The article [53] presents SGX-Log, a novel logging system that is to be implemented to ensure the integrity and confidentiality of log data. The main idea is to redesign the logging mechanism by leveraging Intel SGX, which, as we see, provides a secure enclave with sealing and unsealing functions. These features protect code and data both in memory and on disk, preventing unauthorised modifications, even by highly privileged code. Experimental evaluation demonstrates that SGX-Log imposes no observable overhead on programs that generate logging requests and has only a minimal impact on log daemon performance.

4.2.1 Introduction

Protecting system logs has always been one of the most critical security tasks, as they record detailed information about the state and execution of software. For this reason, when a system is compromised, logs are one of the first targets of an experienced attacker, as they contain traces of the intrusion. One possible solution is to use write-only disks, but this is an expensive and impractical option. Even isolating logs from the machine providing the service, for example, by sending log data to a remote system, does not guarantee consistency, availability, and partition tolerance simultaneously. Another line of defence involves advanced cryptographic solutions. Among the solutions proposed, there are the protocols based on forward integrity by Bellare and Yee; another one comes from Schneier and Kelsey, who introduced a mechanism based on forward-secure MACs and a one-way hash chain. However, such cryptographic approaches cannot fully guarantee log secrecy and may be vulnerable to power-failure attacks, which can compromise the logging system's consistency or security.

Using trusted hardware for secure logging has become a particularly attractive approach, as it protects both the data and the code that manages it. As we saw in the previous chapter, Intel SGX allows the user to allocate private memory regions (enclaves) that are isolated from the operating system. This protects system logs from OS compromise. Furthermore, sealing and unsealing capabilities ensure data remains secure even when stored on disk. Another important feature is remote attestation, which allows a remote party to verify the identity and integrity of the SGX platform. Finally, SGX offers flexibility in configuring the enclave cache size, enabling resources to be tailored to the application's needs.

SGX-Log adopts a client–server architecture. The client generates and sends log requests, while the logging server performs secure logging services. Because the size of the Trusted Computing Base (TCB) directly impacts the system's attack surface, SGX-Log adopts a minimal TCB approach, running only essential logging services within the SGX enclave. To ensure integrity and efficiency, the system introduces a block-level hash key chain protocol that protects logs in a structured, verifiable manner. However, not all logs can be stored directly within the enclave due to the limited size of the Enclave Page Cache (EPC). For this reason, SGX-Log provides the ability to protect logs even outside the enclave through sealing.

Syslog Standard

System logging is the process of recording events that happen within an operating system or application. Modern operating systems generally integrate a default syslog component.

In Linux/Unix environments, syslog is a standard defined by RFC 5424 that is used for message logging and allows the separation of the software that generates log messages, the system that stores them, and the components dedicated to analysis and reporting.

The syslog standard adopts a client–server architecture where the clients that generate the messages are called log sources, and the system that collects, processes, and stores the logs is the log server. Communication between client and server occurs via UDP or TCP/IP. There are three main modules within the log server: the Parser that divides and interprets the information contained in the events, the Processor that processes the configuration file (according to a defined grammar) and applies user-specified filters or rules and the Logger, which records logs to persistent storage, generally using a log rotation policy.

Problem: Adopting a widely used standard like syslog also simplifies the work of an attacker who may have in-depth knowledge of the logging system's structure and operation. Therefore, it is necessary to protect the log server to make tampering with the logs more difficult and maintain their integrity and reliability.

4.2.2 Design

The goal of SGX-Log is to ensure the integrity of system logs using Intel SGX. Unauthorised users must not be able to modify sensitive logs stored by the log server. Furthermore, logs must remain

confidential regardless of whether they are stored inside or outside an enclave. It is also critical to preserve the integrity of the logging code responsible for generating, reading, and verifying logs against tampering, as this code handles sensitive data.

SGX-Log protects logs by leveraging SGX and a hash key chain. To this end, all computations using the initial key in the chain and handling log messages must be protected by trusted hardware. This results in a logical partition of the system into two components: a trusted portion that uses sensitive data and an untrusted portion that does not directly access it. The architecture of the SGX-enabled log server reflects this division.

In the trusted component, the objective is to protect sensitive log messages, the log configuration, and the grammar rule; this is done by the enclave. Internally, the enclave also puts the initial key of the hash chain, and then generates the entire chain needed to verify the message. The trusted code performs all secure logging operations, operating exclusively on protected data. Its primary role is to securely record and verify system logs; therefore, it must implement the various substeps of the logging process and leverage SGX's trusted services to ensure security properties.

In the untrusted component, all logs are sealed and stored outside the enclave perimeter. The part of the log server that does not need access to the raw logs remains outside the isolation boundary. Consequently, the logging system remains secure even if this code is compromised. All code communicating with the operating system, the file I/O handler, and the network I/O handler is considered untrusted.

Registered Interface

During logging operations, system library functions such as writing sealed logs to disk or reading logs from network sockets are required to ensure the correctness of trusted code operations. However, SGX does not allow direct system call invocations from within the enclave. To address this issue, registered interfaces, known as edge routines, enable controlled communication between trusted and untrusted components.

Protocols

Figure 4.2 depicts the two-dimensional hash key chain. Log messages are divided into multiple blocks, each containing a fixed number of messages (bSize). Each block is identified by a unique identifier called bID. Similarly, each message within a block is identified by an mID. The two dimensions, bID and mID, therefore uniquely identify each message.

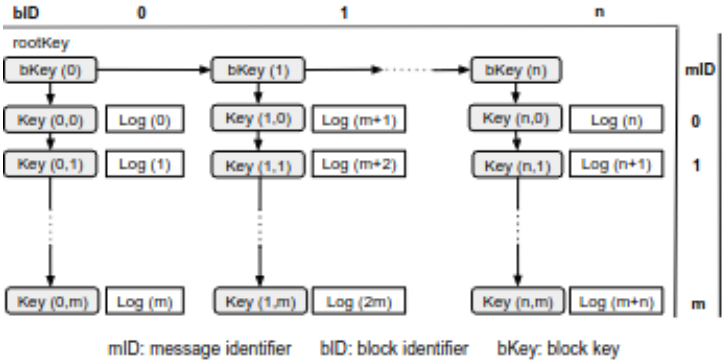

Figure 4.2. Block hash key chain (Source: SGX-Log Article [53])

SGX-Log implements secure logging and integrity verification protocols based on a forward-integrity mechanism using a two-dimensional hash key chain. In the log generation protocol, a new block of messages is initialised within the enclave: the block receives a unique identifier (via

a monotonic counter), the key for the next block is calculated, and the data is sealed to prevent replay attacks. A message-level hash chain is then constructed for each block, using the block key and the individual message identifier. The integrity verification protocol checks the generated blocks sequentially: check if sealed messages are unsealed, then verify if no rollback has happened, and then the block key is regenerated from the root key, and the message MACs are recalculated. Comparing the calculated MACs with the original ones allows for tampering.

4.2.3 SGX-Log Security

SGX-Log leverages the security guarantees provided by Intel SGX to protect log confidentiality and integrity using a block-level hash key chain. Protection is based on the memory encryption provided by SGX: log data, hash keys, and configuration parameters are kept within the enclave, preventing unauthorised access.

However, SGX-Log inherits SGX's inherent vulnerability. This is not designed to defend against side-channel attacks (e.g., cache-timing attacks), physical attacks (e.g., CPU manipulation), or microcode attacks (e.g., modifying machine code). Outside the enclave, log confidentiality is ensured through sealing/unsealing capabilities. However, sealed logs and state values can still be subject to tampering, deletion, or rollback attacks by a malicious actor.

4.2.4 Evaluation

Testing of SGX-Log Client

In the client-side test, they evaluated the impact of SGX-Log on Apache httpd server performance. 10,000 web requests were generated (20 requests per second, with 4 concurrent connections) and server throughput was compared with and without SGX-Log.

The results show no performance overhead. This is because httpd uses an asynchronous logging mechanism: from the web server's perspective, the logging operation is "fire and forget," while message transmission to the log server is handled separately by the communication protocol (UDP in standard syslog or TCP in syslog-ng).

Therefore, integrating SGX-Log does not impact web server performance.

Testing of SGX-Log Server

Log generation and log reading assessment In the SGX-Log server test, the response times for log generation and log reading were measured using datasets from six real programs (including Apache, Postfix, and Ubuntu system logs). Response time has been measured because the logging services run continuously as a daemon process, so measuring execution time is impossible.

- Log generation: Generating 100 logs with SGX-Log incurs an average overhead of 4.84%, with a median of approximately 21.5 ms.
- Log reading: Reading 100 logs sealed in a block incurs a high average of 6.29%, with a median of approximately 23.5 ms.

The time increases proportionally to the message size as the cost of copying to disk increases. Generation is more expensive than reading because it also includes parsing according to configuration rules.

SGX Trusted APIs assessment Finally, the monotonic counter (incremented once per block) does not introduce significant overhead, therefore maintaining overall system performance.

To understand SGX-Log's overhead, the security APIs used in the enclave have been analysed: trusted services such as sealing/unsealing, hashing, and MAC generation are considered the main sources of computational cost.

Sealing and Unsealing: Operations are performed at the block level. Execution time increases linearly with the number of messages in the block (bSize). Sealing is consistently slower than unsealing because it includes data encryption (with additional padding).

Hashing and MAC: In the two-dimensional hash chain mechanism, a new key is generated for each message through hashing and a MAC is then calculated to ensure its integrity. Here, too, overhead increases with message size (128–1024 bytes). The SHA-256 algorithm has been observed to be faster than AES-128 CMAC.

In general, the increase in time is proportional to the amount of data processed, but remains limited and predictable.

Enclave boundary data transfer assessment Secure logging operations require copying sealed data across the enclave boundary. The transfer cost is symmetric: copying data within the enclave costs the same as copying it out of the enclave.

The results show that transfer time increases with data size. Furthermore, dividing data into smaller chunks increases the number of enclave entry/exit operations; therefore, there is an increase in the overall overhead. Moreover, a step-like increase in performance overhead is observed when the buffer size is not a multiple of the chunk size. This highlights the importance of choosing an optimal block size: for example, to transfer data between 1 KB and 1.5 KB, a 768-byte chunk requires only two copy operations, while a 512-byte chunk may require more than two for data larger than 1024 bytes, increasing the overhead.

4.2.5 Limitations

Log Availability: When logs are stored outside the enclave, confidentiality is ensured by sealing. However, sealed data can only be unsealed on the same SGX machine, limiting availability and scalability (also due to the small size of the EPC). To improve scalability, it was proposed to partition the log into more enclaves distributed on different machines. Therefore, through remote attestation and secure key exchange, the enclave can authenticate each other and transfer the unsealed log using a cryptographic channel. The master enclave handles the partition metadata and forwards the request to the correct node.

Ring-3 Level Protection: SGX operates at the ring-3 (userspace) level. This means that kernel log retrieval occurs via system calls, which are considered untrusted. SGX-Log does not protect against attacks on the kernel log buffer or the channel between the kernel and user-space components; its goal is to protect valid logs during recording, retention, and retrieval.

Memory Limits (EPC): SGX imposes a rigid limit on the protected memory (typically 64–128 MB). Applications that exceed this limit must perform the swap using unprotected memory, which reduces performance. However, 128 MB is sufficient to run two enclaves (one for log generation and one for log reads). More efficient memory use can be achieved by increasing the size of sealed blocks.

4.2.6 Conclusions

The protection of the logging system is a crucial question, since it not only provides a detailed view of a system's current state and past, but also has significant forensic value. The article [53] presents SGX-Log, a secure logging system designed to ensure both the integrity and confidentiality of log

data. The solution builds on recent developments in Trusted Execution Environments (TEE) and uses Intel Software Guard Extensions (SGX) as a commercial hardware technology, avoiding the need for expensive proprietary solutions. SGX-Log splits the traditional logging system and performs core operations within an isolated environment (enclave), also leveraging sealing and unsealing capabilities to protect logs even outside the enclave. Implementation on a real SGX platform and subsequent experimental evaluation show that SGX-Log introduces no overhead for log-generating applications and has a very low impact (less than 7%) on logging daemons

4.3 CUSTOS

Auditing is a fundamental element of system security, since logs often represent the primary source for incident reconstruction, access control, intrusion detection, and execution integrity verification. However, attackers are aware of the value of logs and frequently employ anti-forensic techniques to delete, modify, or insert fake logs, even using automated tools (e.g. Metasploit and Last Door). Recent studies indicate that tampering with logs is increasingly common in security investigations.

Despite that, commercial OSs do not offer a robust mechanism for protecting logs; as a result, root access is often sufficient to manipulate evidence. Existing solutions, such as WORM storage, trusted remote servers, or advanced cryptographic techniques, are expensive, poorly scalable, or overly resource-intensive, making them unsuitable for the high loads of modern systems.

For this reason, CUSTOS is proposed. It is a practical, scalable, efficient, and minimally invasive solution for tamper-evident logging compared to existing auditing frameworks. Analogously, as we saw in the previous section, SGX-Log enforces log integrity by leveraging SGX enclaves to isolate critical logging operations. While CUSTOS focuses on scalability and integration with standard operating systems, SGX-Log emphasises the hardware isolation and cryptographic guarantees provided by SGX, representing two complementary approaches to secure logging.

CUSTOS is made of two fundamental elements: the first is the Tamper-Evident Logger, which introduces a tamper-proof logging layer that can be integrated into the existing audit framework. The protocol separates the cryptographic confirmation of the event from the recording operation: events are internally hashed within the enclave and signed asynchronously during the periodic audit. This approach allows for handling more than one million events per second, approximately a thousand times faster than the previous solution based on TEE. It also ensures third-party auditability without requiring log encryption, while maintaining compatibility with standard analytics tools.

The second component is a decentralised auditing system that operates nearly in real time and uses a three-way network protocol between enabled CUSTOS hosts to promptly detect any tampering. This mechanism forces the attacker into a “lose-lose” situation: either he does not alter the logs (leaving forensic evidence), or he modifies them and is detected.

The prototype, implemented on Linux Audit with Intel SGX, demonstrates low overhead (2-7% runtime, 3% network) and effectiveness in detecting tampering in real-world APT scenarios.

4.3.1 Security Requirements for Tamper-Evident Logging

The threat model considers an organisation composed of numerous hosts that is targeted by an advanced attacker (APT) capable of initially gaining unprivileged access and subsequently gaining full control of the operating system. When the system is compromised, the attacker can use anti-forensics techniques to delete, modify, or forge logs and hide their presence. It is presumed that each host has a TEE (e.g., Intel SGX) able to protect the cryptographic key, and that the cryptographic primitives used are safer against falsification and collisions. In this context, a secure logging system should guarantee the tamper detection, the verifiability from a third party, the granular audit support and log availability before the compromise, while maintaining a low overhead and the compatibility with highly loaded systems. However, traditional or TEE-based cryptographic solutions pose practical challenges, including high computational costs, rollback issues, enclave memory limitations, and scalability limitations.

4.3.2 Design

CUSTOS is composed of three main elements. The first is the Tamper-Evident Logger. It is integrated into the auditing framework of the OS with minimal modifications. When log events are generated, they are hashed within a TEE enclave, and the related integrity proofs are signed asynchronously during periodic audits. The cryptographic keys and the signature code are isolated within the enclave, while the public key is made available for external verification. The second component is Centralised Auditing. In this model, a central server (Auditor) sends an audit request to the hosts, which respond by providing logs and related cryptographic evidence. The server verifies integrity and, if an anomaly is detected, generates a security alert. The third element is Decentralised Auditing, designed to avoid the single point of failure of the centralised model. In this scheme, each host also runs an Auditor component within its own enclave and periodically initiates audits against a set of peers. The protocol also supports secure log replication and parallel reconstruction mechanisms, improving resilience and scalability.

In Figure 4.3, an overview of the CUSTOS components is presented. Each rectangle represents a host that records logs using the CUSTOS Logger, which performs integrity checks within a trusted enclave. In the A scenario, a centralised server auditor controls the log of the other host and signals the violation to the system admin. In scenario B, the audit activity is decentralised: each host runs an auditor component within its enclave and performs peer-to-peer verification. This deletes the single point of failure of the centralised model and improves scalability and resilience.

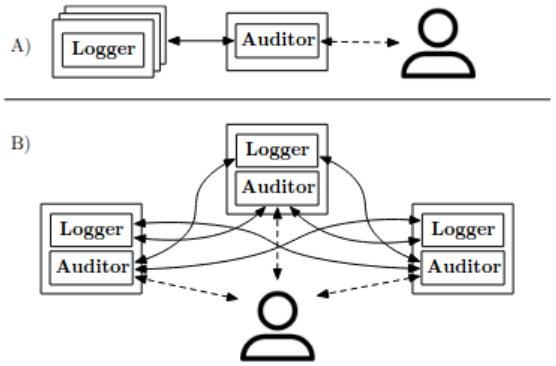

Figure 4.3. Custos Components (Source: CUSTOS Article [54])

4.3.3 Implementation and Analysis

CUSTOS was implemented in C by extending auditd from Linux Audit 2.8.2 and integrating it with an Intel SGX enclave to ensure secure logging and auditing. The system has 2,254 lines of code, of which 658 are executed in the enclave. It uses SGX features such as hardware monotonic counters, SHA-256, and ECDSA to ensure log integrity and authenticity, while communication happens via TCP with TPL serialisation.

From a security point of view, the log tamper-evident is guaranteed. Therefore, the deletion, modification or reordering of the events or block invalidates the cryptographic proof. Truncation attacks are detected during the verification, and a malicious user with root privileges cannot alter the log without being discovered, because the keys are protected inside the enclave. Attempts to terminate or restart the Logger result in verification errors (thanks to the hardware counters and sealed data), preventing rollbacks or undetected manipulations.

CUSTOS ensures third-party verifiability, making each node publish its own public key after initialisation: this allows log blocks to be verified in both online and offline audits (e.g. legal), but requires a key management service.

For granular auditing, the system allows performing checks on specific interval blocks; the block size is adjustable. Since each challenge forces a block to be committed, it is also possible to perform proof-of-retrievability on logs that have been verified.

For log availability, during decentralised auditing, verified blocks are replicated to remote nodes: an attacker would have to compromise all of them to delete them. Logs can be reconstructed using a dedicated protocol. Even malicious or colluding nodes cannot falsify audit responses or results; any concealment attempts would be detected or have a negligible probability of success.

CUSTOS also meets the criterion of minimal invasiveness, as it is fully interoperable with applications already using Linux Audit. While it runs in the auditd process space, its logic is independent of existing code and requires only 26 lines of changes to the original source files, making porting to new versions quick and easy.

4.3.4 Discussions

CUSTOS protects logs even in the case of a total compromise (root): it cannot guarantee the correctness of events recorded after the attack, but it prevents undetectable modification or deletion of previous traces (e.g. entry, reconnaissance, privilege escalation). Compared to traditional centralised logging, it does not assume a fully trusted server: even log servers can be compromised, especially in APT attacks with lateral movement. Finally, the system cannot distinguish automatically between benign crashes and normal attacks, which could lead to false positives, but forensic tools and external data can support the diagnosis.

As a conclusion, the system logs are fundamental to counter the modern threats, such as the APT(Advanced Persistent Threat), but traditional OSs can guarantee the integrity only through access control, which could easily be circumvented in the event of a compromise. CUSTOS is the first tamper-evident logging solution designed to operate practically within the real-world constraints of operating systems. The key idea is to separate event recording from their cryptographic commitments, without compromising security, and by leveraging the capabilities of Trusted Execution Environments (TEEs) already present in modern hardware. Evaluations in the article [54] show that the log commitment protocol is three orders of magnitude faster than previous secure logging solutions and introduces only 2% to 7% overhead compared to insecure logging, even under heavy loads. Furthermore, the auditing protocol detects integrity violations with less than 3% network overhead. CUSTOS therefore represents a practical and scalable solution for making operating system auditing truly tamper-resistant.

4.4 TPM-based Trusted Time Extensions (T3E)

Time is a fundamental element in secure systems, as it is used to validate security properties (such as expirations, key validity, or replay prevention). Therefore, ensuring the integrity of time information is essential.

In the context of Intel Software Guard Extensions (SGX), accessing trusted time from within an enclave is problematic, as the operating system, potentially untrusted, controls the system clock.

This article [55] analyses the critical issues about trusted time in SGX and proposes TPM-based Trusted Time Extensions (T3E), which is an innovative solution that leverages available hardware. In particular, T3E leverages the capabilities of the Trusted Platform Module (TPM) to provide trusted time services to enclaves, protecting them from common attacks such as clock manipulation or rollback attacks.

4.4.1 Trusted Time in Intel SGX

Timing information is a fundamental part of computer systems, as it serves as the reference point for events in the real world. In a security environment, time is essential to implement

secure validation mechanisms, such as the verification of a cryptography key or the timing out of sensitive sessions (e.g. the checking of the validity of a digital certificate)

In the context of Intel SGX, even if a Trusted Runtime System(TRTS) for the trusted application exists, there is an important limitation, because it is not expected to provide direct support to services on the OS side. As we saw in the previous chapter, the enclave can access OS functionality only through an OCALL, which passes through an untrusted domain.

In the past, Intel provided an API in the TRTS called `sgx_get_trusted_time`, intended as a secure alternative to the normal system services used to obtain the time. Anyway, this function was removed in 2020 with the release of Intel SGX SDK 2.8 for Linux, leaving the enclave without a secure mechanism to access the temporal information.

To address this problem, T3E is proposed. It is a solution that combines SGX with the widely available Trusted Platform Module (TPM) hardware. T3E guarantees time reliability from a secure provisioning point, providing a trusted time service that can be used by enclaves.

Trusted Time via PSE and CSME in Early SGX Implementations

Early versions of the Intel SGX SDK included a trusted service called the Platform Service Enclave (PSE). The PSE provided functionality not directly supported by the SGX instruction set.

The PSE operated as a separate enclave, to which the application enclave could connect via local attestation to establish a secure channel. It served as a gateway to the Intel Converged Security and Management Engine (CSME), which offered several trusted services, including a trusted time service and a monotonic counter service.

These services were made accessible to the application enclave through the PSE. Specifically, the SGX SDK for Linux provided the `sgx_get_trusted_time` API, which allowed the application enclave to obtain a trusted time reference from the CSME.

However, as mentioned previously, this feature was removed.

4.4.2 Design

T3E is an open-source secure time service for the Intel SGX platform. It relies on the TPM as a trusted time source and builds a path of trust between the TPM and the enclave. The solution considers the fact that TPMs are not designed to run user software, and that the current version of SGX does not provide a native trusted time mechanism adequate for applications that require it.

The TPM provides a reliable time source independent of the machine's hardware clock. It keeps track of two separate counters: time and clock. The time value represents the number of milliseconds elapsed since the TPM was turned on. The clock, on the other hand, counts the milliseconds elapsed since the TPM's epoch, i.e., since the TPM was first powered on and initialised. While the time value is temporary (ephemeral), the clock is periodically saved in non-volatile memory (NVRAM). The TPM ensures that this clock is monotonically increasing, meaning it cannot be reset.

The monotonic counter is primarily used by the TPM to provide reliable timestamps in attestation and authorisation processes.

From the figure, it is possible to see that the T3E component resides within the secure enclave, along with the client application code that uses the time management service. The T3E component obtains time information from the TPM, which is transmitted via an untrusted channel between the enclave and the TPM. The authenticity of the time data is verified using the TPM's chain of trust, which allows third parties to identify an authentic TPM device and attest to and verify the data generated by the TPM using public-key cryptography. A TPM has a manufacturer-provided Endorsement Key (EK) certified as a trusted root in the TPM's PKI. Since the EK cannot directly sign data, the TPM generates an Attestation Key (AK) derived from the EK, which is used to sign trusted information.

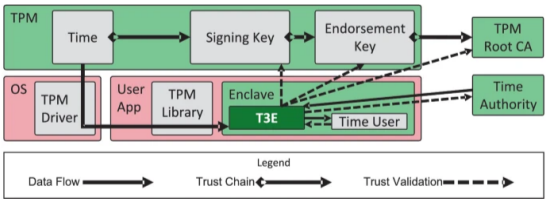

Figure 4.4. Trust model for the T3E system (Source: T3E Article [55])

T3E verifies the authenticity of time data from the TPM and stores it inside the protected enclave.

From the figure, it is evident that the TPM driver is located in the operating system (untrusted), but this is not a problem, and an encrypted channel between the TPM and T3E is not required, as trust is established during the provisioning phase via TPM attestation. Finally, T3E initialisation requires an external time authority (e.g., a trusted NTPSec server) for periodic synchronisation.

4.4.3 Evaluation and attack scenario

For a time service to be considered reliable, it must be able to guarantee the integrity of the time information or detect any attempt to manipulate the returned time within reasonable limits.

The first requirement is that the time source must be authentic. Although a system may have different time sources, such as OS time, software timer, TSC, HPET, PTP, and TPM, most do not guarantee the integrity of the time information. T3E uses two sources: an initial real source during provisioning (e.g., authenticated NTPSec server or manual provisioning) and the time provided by the TPM, which could be cryptographically attested. The enclave verifies the TPM's authenticity through its chain of trust, preventing spoofing and man-in-the-middle (MITM) attacks that aim to impersonate the TPM device or falsify time data.

Secondly, the temporal information should not be reused. The time returned from the service should be monotonically increasing, avoiding the reuse of previous values. To ensure this, T3E uses the `ClockTick` function, which includes a random nonce generated within the enclave in each request to the TPM. The nonce guarantees the freshness of the response and avoids replay attacks by using previous timestamps.

Third, time cannot be sped up. A reliable time source must guarantee a stable and coherent progression consistent with the real clock. To speed up time, an attacker should forge TPM messages or manipulate the time information returned by the TPM. However, the data is protected by cryptographic signatures, which prevent tampering. A direct attack on the TPM chip (e.g., fault injection or side-channel) falls outside the T3E threat model, as TPMs are specifically designed to resist such techniques. T3E also compares consecutive TPM timestamps and can initiate a resynchronisation if an unusual time jump is detected. However, in cloud scenarios where the hardware is managed by the service provider, the TPM owner could use its permissions to advance the clock slightly. If the advance is small, T3E might not detect it, but the attacker's advantage would be limited and neutralised during subsequent periodic resynchronisation with the trusted external time source.

Another key requirement is that the time cannot be paused or slowed down. From the moment that the time service must communicate with a trusted external source, this communication can be intercepted or delayed by the operating system, allowing an attacker to block or delay messages. T3E mitigates delay attacks by imposing a maximum use count after receiving a timestamp from the TPM. This forces the attacker to periodically allow new time measurements to arrive; otherwise, T3E stops providing time, locking the application in the enclave and preventing a silent and prolonged slowdown. The delay does not accumulate over multiple ticks, since T3E requests a new timestamp only after receiving the previous one. However, T3E does not protect against denial-of-service attacks.

4.4.4 Performance Evaluation of T3E

T3E was implemented in C++ using the Intel SGX SDK v2.17 and the TPM2 Software Stack v3.2. Critical attestation operations (challenge generation and signature verification) were moved inside the enclave to ensure their integrity and confidentiality. The experiments are executed on three different machines, and for the evaluation, the RDTS instruction, except for one machine, where an ocall is used to execute the instruction outside the enclave.

From the microbenchmark in the article, where three signature schemes are evaluated, it is possible to see that ECSA is about 30% faster than RSA. Also present is the performance of fTPM solutions, which allows more frequent ticks (up to every 30 ms), improving granularity compared to the previous sgx_get_trusted_time API (1 second).

In the macrobenchmark, a Timestamping Authority (TSA) with OpenSSL was implemented within the enclave, demonstrating its feasibility in a realistic use case. System throughput depends on the tick interval and the maximum use count, which must be properly calibrated to balance security and performance.

Finally, it offers the impact of attacks on throughput. The analysis shows that small delays reduce throughput (e.g., -25% with 10 ms on short-tick systems), while delays of 1 second can reduce it by up to 3–22%. More frequent ticks improve both timing accuracy and resistance to delay attacks.

4.4.5 Conclusions

In conclusion, T3E efficiently addresses the problem of trusted time in SGX enclaves, filling the gap left by the exclusion of sgx_get_trusted_time. Based on TPM 2.0, it provides an authenticated time service that is resistant to replay and manipulation, without requiring hardware modifications or deprecated features. Furthermore, it offers greater granularity and lower latency than previous solutions, making it a practical, deployable approach for systems that require reliable time in untrusted environments.

4.5 Prov-Trust

Data provenance describes the ownership chain of data, including who created, modified, and accessed it. In security investigations, it is essential to identify malicious actions, data breaches, and access policy violations. For these reasons, there are needed systems that must guarantee both the integrity and secure verifiability of provenance information, as well as its confidentiality and protection. In the cloud, the problem is more complex because data management is delegated to a provider that is not completely trusted, and provenance metadata can reveal sensitive information about the user. For this reason, several secure provenance systems have been proposed that protect data integrity, access, and privacy, and Prov-Trust is one of those.

Prov-Trust is a data provenance architecture based on Intel SGX and blockchain, designed to ensure security and privacy in the generation and preservation of metadata. It leverages the distributed and immutable nature of blockchain, cryptographic techniques, and trusted hardware to ensure the integrity and verifiability of records. The SGX enclave executes in a secure way the provenance kernel, handling collection, storage, and querying of the data. The system does not depend on a trusted third party: a smart contract monitors document changes, enforces access policies, and records downloads and transfers, even allowing voting among authorised users on certain events. Provenance data are stored in smart contracts or blockchain transaction logs, reducing storage costs. Finally, Prov-Trust ensures privacy-preserving auditing thanks to logs signed by the SGX enclave, transaction logging after each voting session, and the use of advanced encryption.

4.5.1 Security Requirements

Prov-Trust must meet several fundamental security and functionality requirements. First of all, data confidentiality: it should ensure the secrecy of provenance data and related metadata during collection, storage, and transmission. Another requirement is integrity and verifiability: it ensures that provenance records are not altered during storage, processing, or transport, as integrity is the key property for reliable provenance. The third requirement is accountability, which means providing complete traceability of remote data operations, allowing the owner visibility into collection, access, and processing. Auditability also should be satisfied, as it allows authorised authorities to initiate investigations and obtain consistent evidence in the event of non-compliant activity. Finally, privacy preservation (unlinkability) prevents correlations between smart contracts created by the same owner and between blockchain transactions related to the same data, protecting the identity and user activities.

Blockchain, introduced in 2008 with Bitcoin, is a distributed ledger that allows the transfer of value without a central authority. It is based on digital signatures, peer-to-peer networking, and cryptographic mechanisms (such as proof-of-work), recording transactions in blocks that are shared among all nodes in the network. Miners validate transactions, while nodes participate in their dissemination and verification.

4.5.2 Blockchains

There are two main types of blockchains: permissionless (public), like Bitcoin, open to anyone, and permissioned (private), which maintain decentralisation and transaction validation but have a central entity that manages read/write rights and supports smart contracts written in programmable languages. Permissioned blockchains are particularly suited to enterprise applications, where users do not necessarily need to trust each other. Prov-Trust uses Hyperledger, a permissioned blockchain and collaborative project designed for business applications that offers modular consensus, high scalability, and the ability to handle thousands of transactions per second, supported by a large consortium of leading companies.

4.5.3 Architecture

Prov-Trust is based on three main layers that support different operational phases: SYS INIT, STORAGE, and AUDIT.

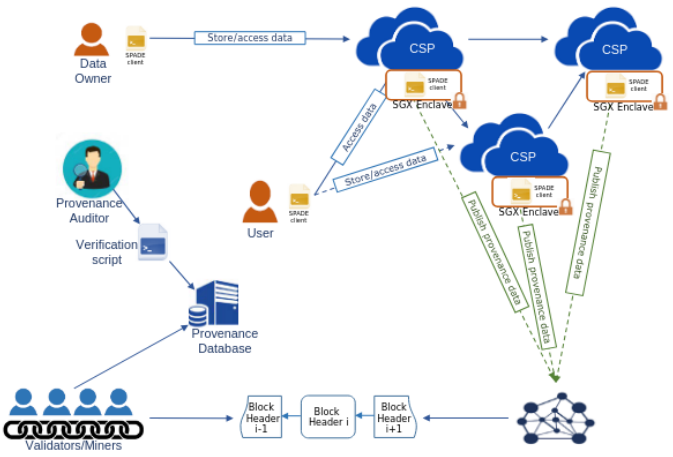

Figure 4.5. Prov-Trust Architecture (Source: Prov-Trust Article [56])

1. SYS INIT (Initialisation) The system is configured, and each entity receives a public/private key pair. Intel SGX enclaves are installed on host devices and provided with the relevant cryptographic keys to ensure trusted execution.

2. STORAGE (Storing Provenance) It includes two protocols:

Create: The Data Owner (DO) creates a smart contract (prov-doc) for each file to be tracked. This contains Access Control Lists (ACLs) on both the data and event logs, and its creation is recorded in a blockchain transaction. Optionally, a voting smart contract (*vote-doc*) can be created to validate sensitive operations such as *download* and *transfer*.

Evt: When a Data User (DU) performs an action, a signed log is generated by the SGX enclave. For downloads/transfers, the *vote-doc* is activated, and the voting result is recorded on the blockchain.

3. AUDIT (Verification) The audit can be performed by the DO or an authorised auditor. Although some parts of the logs and transactions are encrypted, the system allows for the consistent reconstruction of the provenance graph for a specific file, ensuring integrity and traceability.

4.5.4 Evaluation and Conclusions

Prov-Trust is robust against log tampering and provenance data alteration thanks to the combined use of an immutable blockchain and signatures generated by Intel SGX enclaves. An external attacker cannot modify or read sensitive data, because it is published on the blockchain only as random identifiers or encrypted information using Attribute-Based Encryption schemes. An internal attacker also cannot forge records without owning the enclave keys or granting unauthorised access, while critical operations such as downloads and transfers may require a verifiable voting mechanism. The system guarantees auditability and traceability thanks to transactions that are signed and verified publicly. However, security depends strongly on the integrity of SGX and the assumption that the majority of blockchain nodes are honest: a 51% attack could compromise the availability or recording of transactions. Furthermore, Prov-Trust protects log integrity but cannot prevent an authorised user from recording formally valid but semantically malicious actions.

Prov-Trust is a distributed data-provenance architecture designed to ensure security, integrity, and privacy in cloud systems. It is based on the innovative use of Intel SGX to reliably collect provenance data, with storage on the blockchain via transactions and smart contracts, ensuring immutability and auditability. The system also integrates a secure and authorised auditing function, allowing the data owner or an auditor to verify operations performed on data stored in the cloud. Security and privacy analysis demonstrates compliance with defined requirements.

4.6 Memory Forensics in SGX environments

A fundamental aspect to consider in the intersection between Intel SGX and digital forensics regards the challenges that SGX technology poses to traditional forensic investigations. More particularly, in the article [57], the authors note that the isolation provided by SGX is a "double-edged sword": on the one hand, it protects authentic code from untrusted environments, while on the other, it prevents security tools from examining enclave memory and conducting forensic investigations. In particular, the memory of an enclave in release mode is completely opaque to the operating system and any conventional acquisition tools, since Enclave Page Cache (EPC) pages are encrypted by hardware and return a constant value (e.g. 0xFF) when read from outside.

The main contribution of the authors consists of a methodical study of the artefacts recoverable from an SGX machine despite enclave isolation. The authors demonstrate that, although the enclave contents are not directly accessible, a forensic analyst can still extract significant information from the kernel structures (ISGX and DCAP drivers, which maintain metadata about allocated

pages and ELRANGEs), the user-space memory layout (which allows distinguishing API-like frameworks like Intel SGX SDK from container-like frameworks like Graphene/Gramine), and the enclave interfaces (the ECALL and OCALL functions, which, being defined in the untrusted portion of the host process, are recoverable through static analysis). The work was validated on a dataset of 45 SGX applications, including commercial applications (R3 Conclave) and two malware enclaves (SGX-ROP and SnakeGX), achieving zero false positives and zero false negatives in identifying interfaces.

A particularly relevant point for our work is the observation of the author regarding enclaves used for malicious purposes: since compiling in release mode requires a key issued by Intel (thus exposing the identity of the developer), malware enclaves tend to be deployed in debug mode, making them inspectable via the EDBGRD opcode. This reinforces the rationale behind our framework: using SGX enclaves in release mode to protect the forensic process ensures not only computational integrity (via remote attestation), but also that the contents of the enclave, including evidence hashes, chain of custody, and cryptographic keys, are inaccessible even to an attacker with root privileges on the system.

In the article [57], the work complements the one seen in this thesis: the authors study how to conduct forensic investigations on machines hosting SGX enclaves, analysing traces left by the host process and kernel from the outside. This framework, in contrast, takes the opposite approach: it uses SGX internally to protect the forensic workflow itself, performing hashing, chain of custody, and data sealing within the enclave. In other words, while Toffalini and others consider SGX an obstacle to forensic analysis, our work proposes using the same "wall" to protect digital evidence during the preservation and analysis phases.

4.7 Proposed Framework: Secure Digital Forensics with Intel SGX and TSK

The works presented in the previous sections address specific, but isolated, aspects of the intersection between TEE and digital forensics:

- **TEE-BI**: exploits ARM TrustZone to conduct digital investigations on remote Android devices, by extracting cryptographic SSL keys directly from the memory of the target device, in order to decrypt encrypted communications in a legally compliant way.
- **SGX-Log**: rebuilds the system logging of an OS integrating Intel SGX. The logs are protected through a hash chain and the sealing mechanism.
- **CUSTOS**: addresses the same problem as SGX-Log but focuses on scalability. It also includes an auditing system to detect OS compromise.
- **T3E**: resolves the problem of the lack of a trusted time service inside SGX enclaves, following the removal of the `sgx_get_trusted_time` API. It uses the TPM as a trusted time source, protecting the enclave from time manipulation attacks.
- **Prov-Trust**: combines Intel SGX and blockchain to guarantee data provenance in cloud environments.
- **Memory Forensics in SGX**: studies the opposite problem with respect to the other works. Instead of using SGX to protect data, it analyses how to conduct forensic investigations on machines hosting SGX enclaves, identifying which artefacts can be recovered despite hardware isolation.

However, none of these solutions addresses the problem of protecting the entire forensic analysis workflow (from evidence hashing and metadata extraction to chain of custody management and the production of forensic output for judicial use) inside a trusted boundary imposed by the hardware.

This thesis proposes a framework that draws inspiration from these existing works to build its own solution. In particular, the hash chain mechanism and the sealing approach introduced by SGX-Log were taken as a reference for protecting the integrity and confidentiality of the evidence log, while the concept of remote attestation, used in TEE-based systems, to attest the integrity of the software running within the trusted environment, was adopted and extended to allow external parties to verify the integrity and authenticity of the forensic output. Building on these foundations, the proposed framework integrates Intel SGX with TSK. Additionally, an ECDSA digital signature mechanism was integrated to ensure that the exported forensic report is cryptographically bound to the enclave that produced it. Building on these foundations, the proposed framework integrates Intel SGX with TSK. A description of the architecture, implementation, and testing of the framework is provided in the following chapters.

Chapter 5

Design and Implementation

Forensic-SGX was developed and tested on a virtual machine hosted on the Microsoft Azure cloud platform. The VM uses Ubuntu 22.04 LTS and is configured with the Standard DC2s v3 size, which makes available two Intel SGX-enabled vCPUs and 16 GiB of memory. The DCsv3 series was chosen for its native support for Intel SGX, a requirement for running enclaves in hardware mode. This configuration enabled us to work in an environment aligned with the project's objectives, ensuring the availability of SGX's isolation features.

5.1 Objectives and Motivation

The framework was developed to integrate a digital forensic tool with Intel SGX, with the specific goal of protecting the analysis and preservation of evidence. As a digital forensic tool, TSK has been selected as a specialised tool for analysing disk images. Due to the memory constraints of Intel SGX, which in this specific server configuration amounts to 8 GB, the disk images used for testing were limited to this size. Furthermore, it was decided not to use the enclave to store the disk image itself (this latter is not used as evidence), but to use it to store only the log, containing the metadata of each file found on the disk, each associated with a SHA-256 hash computed entirely within the trusted environment, in order to protect the data from tampering. As discussed in the following sections, the integration of TSK directly into the enclave has significant challenges because of the numerous external libraries and the limitations of TEEs, which do not support system calls or direct filesystem access. For that reason, two solutions have been developed: in the first, the extraction of the file and its metadata are performed within the untrusted part using TSK, while all security activities are executed in the enclave, such as the hash engine, sealing and unsealing operations, ECDSA key generation, and construction of the signature and CSV files. In the second mode, a similar TSK solution is developed, but, as the reason stated before, a custom FAT32 analyser is used instead, just for comparison and as a first solution for future research, because it is limited to only one filesystem with respect to TSK, which is able to analyse different ones. The difference with the other mode is that the disk is now temporary inside the enclave, and all analysis and data extraction are executed entirely within the enclave. Another limitation is that only disk images lower than the HeapMaxSize can be analysed. To ensure security, the disk image is analysed to not exceed the memory enclave, with a size smaller than 7.50 GB. That is because a part of the heap is used by the framework to store everything that is needed(e.g. enclave metadata).

The final scope of this framework is to allow an investigator to obtain a csv signed report that can be used as evidence in an investigation. In the following section, it will be explained how the realisation of the signed CSV is divided into two phases: the first one, which is the analysis phase, is the one described before with the two modalities, and it produces a sealed file containing all the log information about the files analysed, each of them associated with a SHA-256 hash. After this part, a possible investigator could decide to obtain the CSV-signed file in extract mode or wait until it is needed. During this phase, all necessary controls are executed before producing the file: unsealing and hash chain verification, followed by the signature procedure and the generation of

the DCAP quote. The typical flow for a possible investigation is shown in the figure below 5.1. It is worth noting that the acquisition phase, which typically involves tools such as FTK Imager to create a bit-by-bit copy of the original disk and verify its integrity via MD5/SHA-1 hash, is outside the scope of this work. The framework assumes that a forensic disk image has already been acquired and takes it as input. After the analysis and export phase, the signed CSV file is submitted to an external verifier or anyone who needs the evidence, who can verify the integrity and authenticity of the evidence log without requiring access to the original disk image or the enclave. The transmission of the CSV file and the corresponding .sig file to the external verifier is outside the scope of this work. The framework assumes that the two files are delivered to the verifier through a secure and trusted channel. The blocks in yellow are not implemented in the framework and are outside of this work.

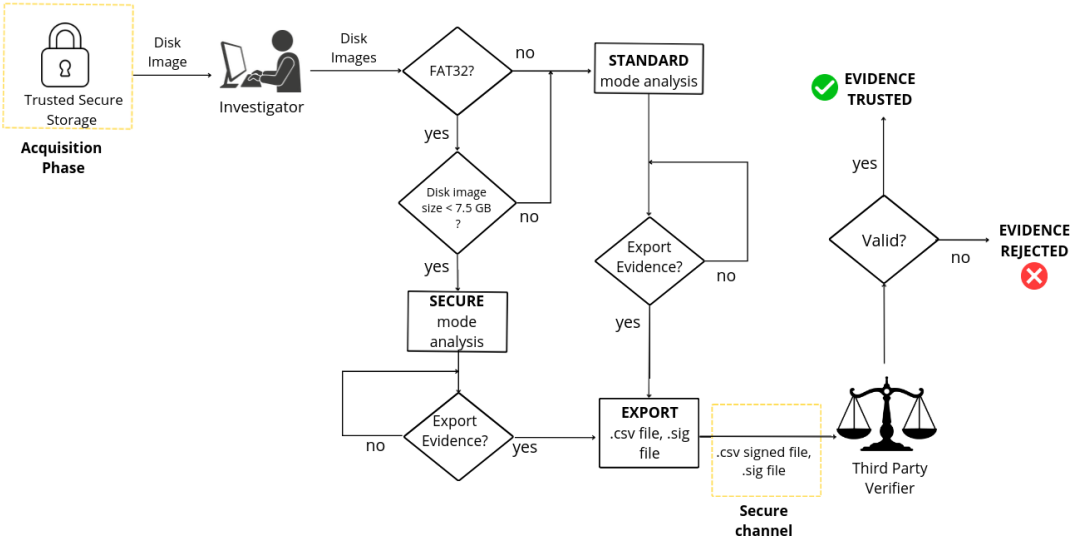

Figure 5.1. Forensic-SGX investigation workflow

5.2 Architectural overview

This section describes the high-level architecture of the framework developed, which is organised following the Intel SGX implementation, where a typical Intel SGX application is divided into two domains: the untrusted domain, that is, the Host Application, and the trusted domain, the Enclave. The two domains can exchange information by using ECALLs and OCALLs. ECALLs are typically used by the application to communicate with the enclave; instead, OCALLs are used by the enclave to communicate with the untrusted component.

Figure 5.2 illustrates the general architecture of an Intel SGX application, with the division between the trusted part (Enclave) and the untrusted part (App). The Forensic-SGX framework is developed following the same structure. As shown in the figure, the first operation performed by the App is to create the enclave, which is loaded into the EPC, a hardware-protected memory region. Every time the App needs to call a trusted function, the call passes through the *call gate* before reaching the enclave. The call gate is the only entry and exit point of the enclave, and corresponds to the boundary defined in the `.edl` file. Inside the enclave, all data are processed normally, but cannot be read from outside. This is represented by the red cross in the figure, which indicates that privileged system code, such as the OS, VMM, and BIOS, is not allowed to access EPC memory, even though it holds the highest system privileges. When an ecall terminates and returns to the untrusted side, the data remains in EPC and does not leave the trusted environment unless explicitly specified in the `.edl` file. The call gate also handles ocalls, which follow the same mechanism in the opposite direction: when the enclave needs to call an untrusted function, the call passes through the call gate before reaching the App.

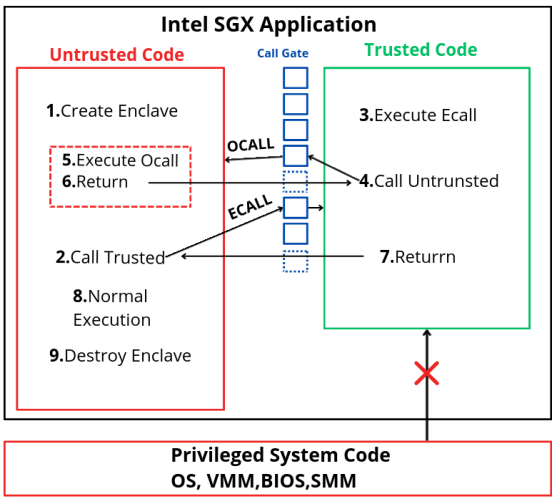

Figure 5.2. Intel SGX Application

Figure 5.3 illustrates the high-level architecture of the realised framework, referred to as Forensic-SGX. The architecture shows the three operating modes: Standard Mode, Secure Mode, and Export, all invoked through the command line interface as described in Table 5.1. The general execution flow follows the same structure shown in the image 5.2: depending on the mode selected, the untrusted App initialises the enclave and begins execution through a series of calls to the trusted environment. The red-dotted components represent parts considered insecure: they could be subject to compromise, and any compromise is not detected. The yellow components indicate parts that could also be compromised, but for which any modification is detected. Finally, the green components represent the trusted parts of the system, which are considered trusted.

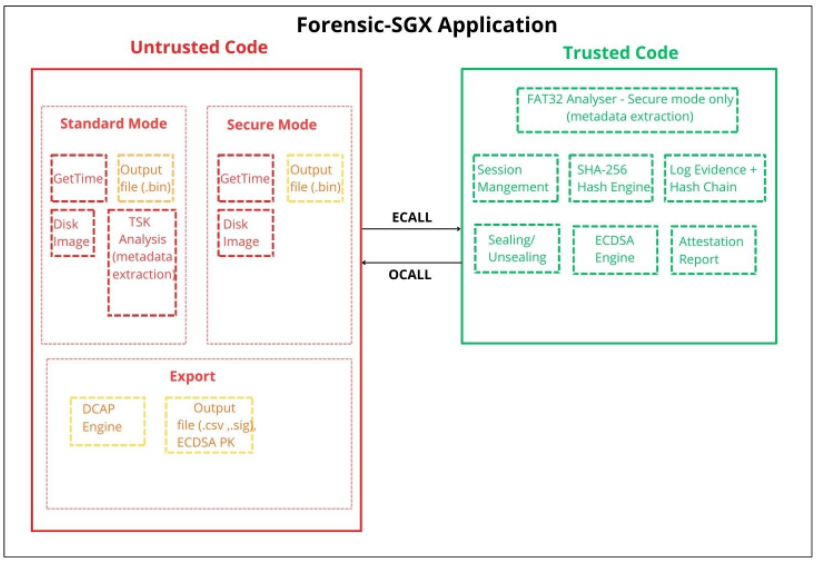

Figure 5.3. Forensic-SGX High-Level Architecture

5.2.1 Untrusted component: App

All the parts related to untrusted activity are implemented inside App.cpp. The untrusted part can be considered as the orchestrator of the entire framework. This is because all three operating

modes are implemented on the App side, and every interaction with the enclave is initiated from the untrusted environment through the corresponding ECALLs. However, not all components on the App side have the same level of risk. The following describes the specific untrusted components and the reasons why they are considered insecure. Getting time is an operation considered untrusted because the timestamp is computed by the operating system, which could lie outside the trusted environment. Similarly, the disk image and the TSK library are not considered safe, as the files or the metadata extracted from them could be modified before reaching the enclave. The management of disk images stored in the App is also a concern in Secure mode, where the images could be altered before being transferred into the enclave. Regarding the yellow components, these represent all the outputs coming from the enclave that, once they reach the untrusted App, could be altered. However, this is not a concern, since any modification can be detected through MAC verification during unsealing or ECDSA signature verification during export. Finally, for the DCAP Engine, the App is only responsible for initialising the DCAP infrastructure and triggering the quote generation process. Any compromise of this component would be detected by the external verifier during the attestation verification phase.

Command-line	*Description*
`./app analyse <disk_image> <analyst_id> <case_id>`	Analyses the disk image in standard mode (TSK) and seals the evidence to file.
`./app analyse-secure <disk_image> <analyst_id> <case_id>`	Analyses the disk image entirely inside the enclave (secure mode, FAT32 only).
`./app export <sealed_file> [output.csv]`	Unseals the session, verifies the hash chain, start DCAP quote generation and exports the ECDSA-signed CSV.

Table 5.1. Command-line supported by the framework

Inside the App is also present the implementation of a command that is called cmd_verify-quote, which simulates the verification steps that should be followed by an external verifier.

5.2.2 Trusted Component: Enclave

The Enclave is implemented in the `Enclave.cpp` file and it is the component where all sensitive operations and data are securely executed and stored. Sealing is essential to preserve the current state of the forensic session, guaranteeing that the sealed file can be unsealed only by the same enclave instance on the same platform. The SHA-256 engine is computed entirely inside the enclave to prevent any external interference or compromise, and the same applies to the hash chain, which guarantees the integrity and ordering of the entire evidence log. The evidence log itself is produced inside the enclave, and the CSV export is produced directly from this trusted data. The ECDSA engine is responsible for key pair generation and CSV signing, with the private key that is never exported outside the trusted environment. Finally, the attestation report is generated inside the enclave and constitutes the cryptographic proof used during the remote attestation phase, allowing an external verifier to confirm that the operations were performed inside a genuine SGX enclave.

The FAT32 Analyser, instead, is specific to Secure Mode and represents the component where the entire filesystem analysis and metadata extraction are performed inside the enclave, ensuring that all data processed is considered trusted.

5.2.3 EDL Interface Design

The interface between the untrusted host application and the trusted enclave is formally defined in the `Enclave.edl` file, written in Enclave Definition Language (EDL). The EDL file governs every interaction across the SGX security boundary: it lists all functions and specifies the direction of each parameter. The Intel **Edger8r** tool processes this file during the build phase and automatically generates the trusted and untrusted edge routines (`Enclave_t.c` and `Enclave_u.c`),

which handle data verification and marshalling at runtime. In particular, the trusted bridge verifies incoming ecall parameters, ensuring that enclave memory is not unintentionally overwritten and preventing TOCTOU (Time-of-Check to Time-of-Use) attacks, while the trusted proxy copies output parameters from the enclave to untrusted memory at the beginning of each ocall. This mechanism is fundamental to the SGX security model: the enclave never directly accesses untrusted memory, and all data transferred must be explicitly declared in the EDL file. If the conditions specified in the EDL file are not met, both components report an error 5.4.

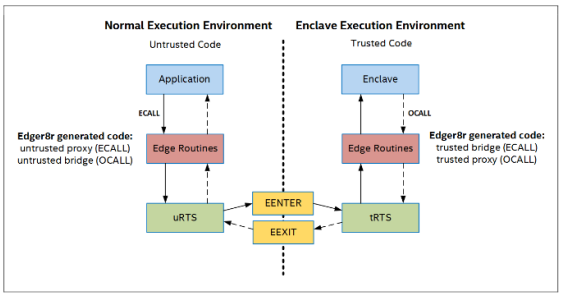

Figure 5.4. Enclave interface with trusted edge routines (Source: Intel SGX [58])

EDL Pointers and Attribute specifiers

Each parameter in an EDL function declaration carries one or more direction and type annotations that instruct **Edger8r** how to handle data passing across the security boundary. The framework uses the following annotations. When an application executes an ecall with a parameter `[in]`, the edge routine copies the contents of untrusted memory to a trusted memory area and passes that copy to the enclave. For an `[out]` parameter, the edge routine instead allocates a buffer in trusted memory, clears it, and passes it to the trusted function. After the function finishes execution, the trusted bridge copies the buffer contents from trusted to untrusted memory 5.5. Other attributes can be used as `user_checks` and `isptr`. For more details, refer to [58]. Ocalls handle parameters in a similar way: for [in] parameters, the contents are copied from trusted to untrusted memory, while for [out] parameters, the copy occurs in the opposite direction.

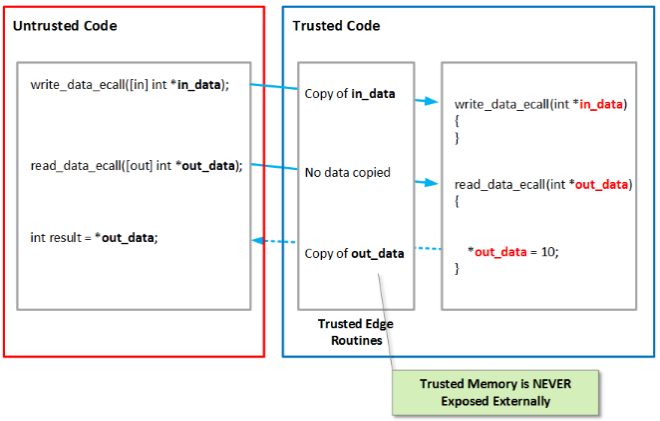

Figure 5.5. ECalls function with "in" and "out" attributes (Source: Intel SGX [58])

Intel SGX provide also attribute modifiers. The modifiers used in this framework are `size`, which indicates the buffer size in bytes used for copy depending on the direction attribute (`[in]`/`[out]`) and attribute `string`, which indicates the parameter is a NULL-terminated C string. Furthermore, in the case of the `[in]` pointer in ecalls or `[out]` pointer in ocalls, all objects passed to the enclave via are valid exclusively for the duration of the corresponding ecall/ocall.

Trusted Functions: ECALLs

The trusted block of the `Enclave.edl` file declares all the functions that the host application can invoke on the Enclave, called ECALLs. Table 5.2 lists all the ECALLs invoked in the framework.

ECALL	*Description*
`ecall_init_forensic_session`	Initialise a new forensic session in the enclave. Must be called before any analysis operation.
`ecall_close_forensic_session`	Closes the current forensic session and releases all associated internal resources. Should be called before destroying the enclave.
`ecall_init_hash`	Initialises a new SHA-256 hash context inside the enclave. Must be called before processing any file data block.
`ecall_update_hash`	Updates the SHA-256 computation with a new data block. Is called every time TSK library reads file blocks in Standard Mode.
`ecall_finalize_hash`	Finalizes the SHA-256 computation and stores the final digest in the enclave. The hash never gets out from the trusted environment.
`ecall_add_evidence_entry`	Adds a new entry to the log inside the enclave. Is associated also the SHA-256 hash that is stored inside the enclave.
`ecall_get_chain_hash`	Returns the current chain hash head and the total number of entries in the hash chain.
`ecall_verify_chain`	Recomputes the entire hash chain and verifies if is been tampered or not.
`ecall_get_sealed_data_size`	Computes and returns the buffer size that is required to store the sealed forensic session data. Must be called before `ecall_seal_forensic_session`.
`ecall_seal_forensic_session`	Seals the entire forensic session (in particular the state of the enclave) using SGX sealing with `MRENCLAVE` policy. The sealed file can then only be unsealed by the same enclave on the same platform.
`ecall_unseal_forensic_session`	Unseals a previously sealed file and restores its internal state. It fails if the sealed file has been tampered with.
`ecall_get_csv_export_size`	Returns the buffer size required for the CSV export. Must be called after unsealing to determine the allocation size.
`ecall_export_evidence_csv_signed`	Generates the evidence log as a CSV file. All the operation are executed entirely inside the enclave. Is also computed the SHA-256 hash of the CSV file, and the latter is signed through ECDSA. Also DCAP is implemented.
`ecall_get_ecdsa_public_key`	Exports the ECDSA public key generated inside the enclave. This key can be used by external verifiers to validate the signature.
`ecall_load_disk_image_init`	Allocates a buffer of the size of the disk image in the enclave heap to receive the disk image. This is the first step of the loading of the image through chunks.
`ecall_load_disk_image_chunk`	Transfers a 32 MB chunk of the disk image into the preallocated enclave buffer at the specified offset. This is called repeatedly until the entire image has been transferred.
`ecall_load_disk_image_finalize`	Finalizes the loading phase. That signals that the disk image has been fully transferred into trusted memory and is ready for analysis.
`ecall_analyse_disk_image`	Performs a FAT32 analysis entirely inside the enclave, operating on the disk image that is previously loaded into enclave. Extracts file metadata and computes SHA-256 hashes.
`ecall_unload_disk_image`	Securely frees the disk image from enclave memory after analysis is complete.
`ecall_create_attestation_report`	Generates an SGX report for DCAP remote attestation.

Table 5.2. ECALLs

Untrusted Functions: OCALLs

The untrusted block of the `Enclave.edl` file declares the functions that the Enclave can invoke on the host application, the OCALLs. The framework developed deliberately minimises the number of OCALLs, exposing only two functions as defined in the Table 5.3,

OCALL	*Description*
`ocall_print_string`	The enclave cannot directly call `printf` because it does not have access to the standard output (that is, an OS resource). When a message needs to be displayed from the enclave, it is necessary to invoke this call, which copies the string out of the enclave and passes it to the Untrusted application. This ocall must be called only in debugging mode. In a real environment, it should not be used.
`ocall_get_current_time`	since the enclave does not have direct access to the system clock, this function is used by the enclave to obtain information about the current UTC timestamp. It is returned to the enclave as an ISO 8601 formatted string.

Table 5.3. OCALLs

Minimising the surface area of OCALLs reflects a core design principle: each OCALL represents a potential threat through which the Enclave could be affected by untrusted code. By limiting OCALLs to non-sensitive operations, the framework minimises the attack surface for interactions between the enclave and the external environment.

External EDL files

In addition to the functions defined by the application, `Enclave.edl` imports two external EDL files provided by the Intel SGX SDK and DCAP packages:

- `sgx_tstdc.edl`: imports trusted wrappers for the C Standard Library (sgx_tstdc), providing a secure subset of the C Standard Library functions available within the Enclave, including memory operations and string handling.
- `sgx_dcap_tvl.edl`: imports the Trusted Verification Library for DCAP (sgx_dcap_tvl), enabling quote verification capabilities directly in the Enclave. This import supports the remote attestation infrastructure used during the export phase to validate the integrity of the attestation chain.

Both imports follow standard Intel SGX SDK conventions and do not introduce additional security issues, as they are distributed and maintained by Intel as part of the official SDK and DCAP packages.

5.2.4 Enclave Configuration File

This section describes the XML configuration file that defines all the parameters set to establish the correct context for starting the enclave. The general XML configuration that is used for this specific framework is as follows:

```
<EnclaveConfiguration>
    <ProdID>0</ProdID>
    <ISVSVN>0</ISVSVN>
    <StackMaxSize>0x100000</StackMaxSize>
    <HeapMaxSize>0x200000000</HeapMaxSize>
    <TCSNum>1</TCSNum>
    <TCSPolicy>1</TCSPolicy>
```

```
    <DisableDebug>0</DisableDebug>
    <MiscSelect>0</MiscSelect>
    <MiscMask>0xFFFFFFFF</MiscMask>
</EnclaveConfiguration>
```

The first two `ProdId`, the product ID assigned by the ISV (Independent Software Vendor), and `ISVSVN`, that is, the ISV Security Version Number, are both set to 0 because this is a research project. These parameters are relevant in a production context. The `StackMaxSize` is used to impose the amount of stack that is usable by a thread when it is created and initialised. In this framework, it has the value `0x10000` that corresponds to 1 MB. It is typically relevant in scenarios with multithreads or functions with a huge array allocated on the stack or deep recursion. The `HeapMaxSize` instead is the total amount of heap a thread can use. There is no specific value because, based on the situation, a different weight is used.

`TCSNum` defines the number of Thread Control Structures (TCS), which corresponds to the maximum number of concurrent threads that can execute inside the enclave. In this project, it is set to 1, which is sufficient because the analysis is inherently sequential: files are processed one at a time, hashed individually, and added to the evidence log in order. texttttTCSPolicy is set to `1`, meaning that a TCS is bound to an untrusted thread, so the same thread that enters the enclave will be the one to exit. `DisableDebug` is set to `0`, meaning that the enclave runs in **debug mode**, which allows a debugger to inspect the enclave's memory. This value should be set to 0 just in the research case. In production, the setting should be set to 1 to prevent any external inspection of the protected memory. `MiscSelect` and `MiscMask` control which additional information is included in the enclave's measurement during attestation. The values are the standard used in most enclave deployments.

5.3 Standard Mode Analysis

5.3.1 Tsk_analyser: the Reference Code

Before the development of Forensic-SGX, a forensic analysis tool called `tsk_analyser` was developed as a preliminary prototype. It was built starting from an example available in the official TSK repository, which was subsequently modified to perform a complete forensic analysis of a disk image. The program accepts a disk image as input and analyses its entire content. Specifically, the original code was adapted to extract file metadata and record the start and end times of the analysis. With respect to the original version, MD5 hashing was removed and replaced with SHA-256 to improve the cryptographic strength of the digest computation. This prototype served as the foundation for the subsequent design of the Standard Mode of Forensic-SGX, and its internal structure directly influenced the architecture of the TSK integration within the framework.

Figure 5.6 provides an overview of the forensic analysis execution flow using TSK. Specifically, it is possible to observe how the start and end times of the analysis are recorded at the beginning and end of the process, respectively, using the `clock_gettime()` function. These timestamps are included in the CSV file, providing temporal evidence of the analysis. After opening the log file, the disk image is opened by calling `TskImgInfo::open()`, which also determines the filesystem type and verifies whether it is compatible with TSK. Once the disk image is successfully opened, the volume system analysis process, indicated as `procVs`, begins.

The `procVs()` function is responsible for processing the disk as a volume system, identifying the partitions and volumes present, and determining how the disk is organised before proceeding with filesystem analysis. It receives as input parameters the image information structure, containing the data required for the analysis, and the byte offset from which the analysis begins, which in this case is zero, corresponding to the beginning of the image. The reason for this layer is that a real disk can contain multiple partitions (for example, a system partition and a data partition), and `procVs()` ensures that all of them are analysed, not just the first. As illustrated in Figure 5.7, two distinct scenarios can occur. In the first case, if the disk contains a partition table, the volume system is opened using `TskVsInfo::open()`. This function iterates over all allocated partitions via `vsPartWalk()`, and for each partition found, it invokes `vsAct()`, which

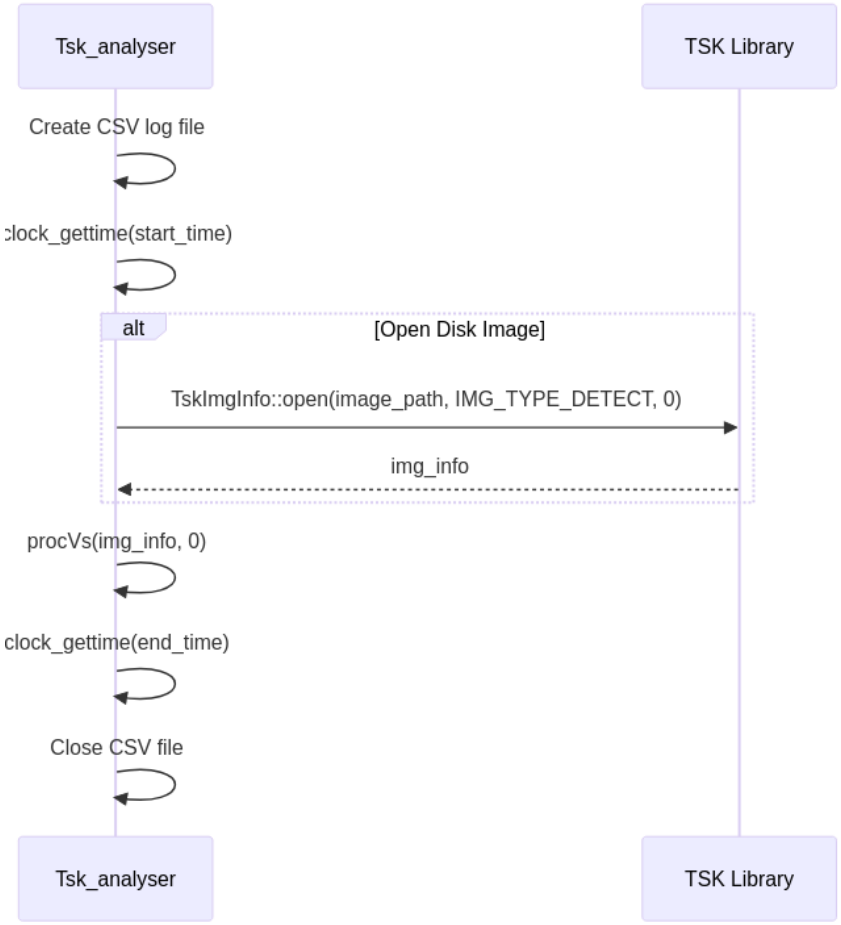

Figure 5.6. Tsk_analyser: Top-level execution flow

in turn calls `procFs()`. In the second case, if no partition table is present, the entire image is treated directly as a filesystem and `procFs()` is invoked directly at offset zero.

The `procFs()` function opens and analyses the file system contained in a single partition, starting from the offset received as input. It attempts to open the file system at the specified offset, using the auto-detection mechanism; if the operation fails (for example, if the partition is unrecognisable), it returns an error and continues with the next partition. If the file system has been successfully opened, a recursive operation of the entire directory tree is initiated, starting from the root inode, using `TskFsInfo::dirWalk(rootINum, RECURSE, dirAct, NULL)`. For each entry found during the walkthrough, whether it is an allocated file or a deleted file, the `dirAct()` callback is invoked, which in turn calls `procFile()` to process that individual file.

As shown in Figure 5.9, the function `procFile()` executes a series of steps for processing a single file. Before proceeding with the analysis in `procFile()`, a series of preliminary checks is performed, including verifying that the file has valid metadata and names and that it is a regular file. System files, directories, and other types are then discarded. If the checks are successful, the metadata is extracted from the structure. Next, the SHA-256 context is initialised via `SHA256_Init(&sha256_ctx)` and the reading of the file contents is started via `fs_file->walk(NOID, fileAct, &hash_ctx)`. TSK reads the file block by block and for each block invokes the `fileAct()` callback, which incrementally updates the digest via `SHA256_Update(&sha256_ctx, buf, size)`. Once all blocks have been read, `SHA256_Final()` finalises the digest, producing the SHA-256 hash of the file's contents. An example of the first lines of the CSV generated is shown in 5.8.

Finally, the computed hash, along with the extracted metadata and a `CLOCK_REALTIME` timestamp, are written directly to the CSV file via `fprintf()`.

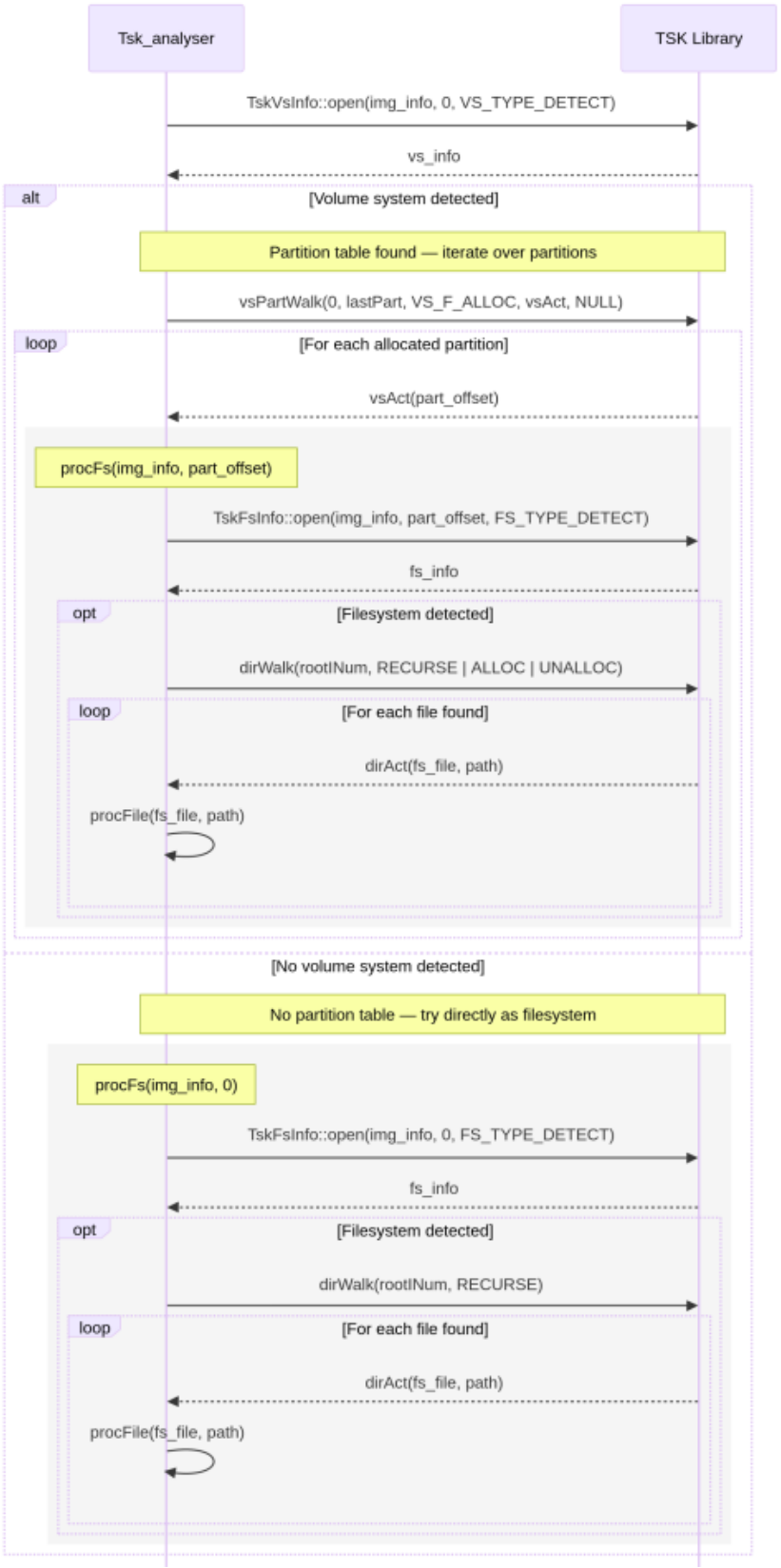

Figure 5.7. Tsk_analyser: Analysis Flow of procVs()

Analysis Start: 2026-03-06T23:10:57.513Z
Analysis End: 2026-03-06T23:11:09.404Z
Mode: TSK_ANALYZER (no SGX)

Timestamp	Path	Inode	Size	SHA256	ChainHash	MTime	ATime	CTime	CRTime	Deleted
2026-03-06T23:10:5	MYDISK (Volume	3	0	e3b0c44298fc1c14	N/A	1772827000	1772755200	0	1772827000	0
2026-03-06T23:10:5	documents/work/repc	391	80454	5213a4b97051ed28	N/A	1772827048	1772755200	0	1772827048	0
2026-03-06T23:10:5	documents/work/repc	394	57301	f7d54ae3718fd0a7c	N/A	1772827048	1772755200	0	1772827048	0
2026-03-06T23:10:5	documents/work/repc	397	68599	8cb3e907e9cc2fc81	N/A	1772827048	1772755200	0	1772827048	0
2026-03-06T23:10:5	documents/work/repc	400	78941	381fb6ddc16545c72	N/A	1772827048	1772755200	0	1772827048	0

Figure 5.8. Example of CSV Output generated by the TSK Analyser

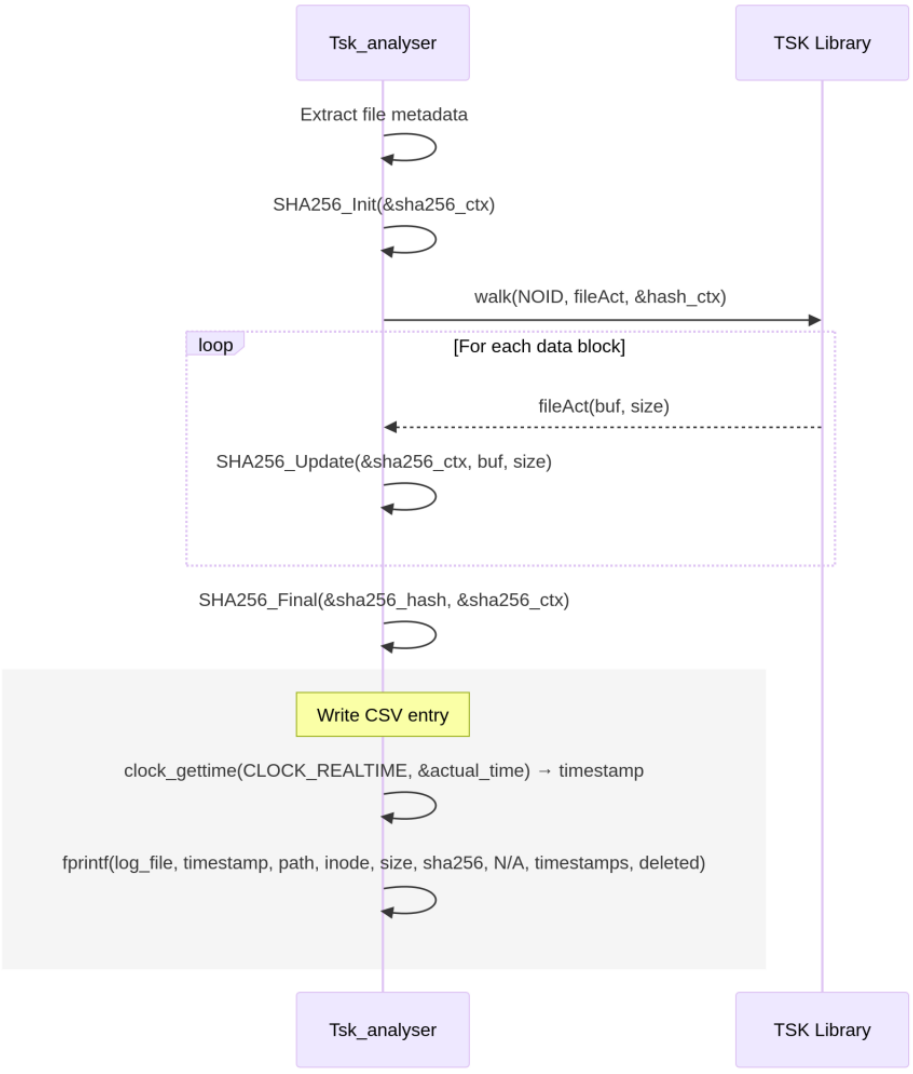

Figure 5.9. Tsk_analyser: procFile() Flow

5.3.2 From Tsk_analyser to Forensic-SGX framework

Observing `tsk_analyser`, it is evident that not all the operations performed are considered trusted, since everything is executed in an untrusted domain. This limitation motivated the development of a new framework capable of providing security guarantees during the analysis phase. In the Forensic-SGX application, the TSK analysis process is essentially the same as in the prototype. Up to and including the `procVs()` function described in Figure 5.7, the analysis flow is unchanged and has been directly incorporated into the new framework. The main difference lies in the `procFile()` function. In Forensic-SGX, the SHA-256 computation is performed inside the enclave, a hash chain is maintained to guarantee the integrity and ordering of the evidence,

and the evidence log is generated and stored entirely within the trusted environment.

5.3.3 Forensic-SGX framework

Figure 5.10 illustrates the flowchart of the Standard Mode, showing the communication between the enclave and the untrusted component. Since the execution takes place in an SGX environment, the first operation performed is the initialisation of the enclave via `sgx_create_enclave()`. Once the enclave is created, the forensic session is initialised through `ecall_init_forensic_session()`. During this call, all necessary checks are performed, including validating the input parameters provided via the command line. If no session is already active, the ECDSA key pair generation is initiated. It is important that this generation occurs within the enclave and prior to the sealing phase. The private key never leaves the trusted environment and will be used during the `export` phase to sign the evidence log. During this phase, the hash chain is also initialised. It was initially implemented as an additional defence-in-depth measure; however, after analysing the design, it was found to be redundant, since the framework already provides integrity guarantees via AES-GCM sealing (as documented by the `sgx_seal_data()` function) and the ECDSA signature. Before concluding the forensic session initialisation, the enclave invokes `ocall_get_current_time()` to request the current timestamp from the App, which corresponds to the time at which the analysis begins. In a forensic context, recording both the start and end timestamps is important in order to establish a temporal record of when the analysis was conducted. After the forensic session has been initialised, the actual disk analysis starts using TSK. The only difference with respect to the `Tsk_analyser` lies in the `procFile()` function, which, in this case, exhibits different behaviour.

In the `procFile()` function of the Standard Mode, illustrated in Figure 5.11, for each file identified by TSK, the metadata extraction is performed in the untrusted App. Subsequently, `ecall_init_hash()` is invoked to initialise a new SHA-256 context within the enclave. After that, `fs_file->walk(NOID, fileAct, NULL)` begins reading the file block by block: for each block read, `ecall_update_hash()` is invoked, passing the block to the enclave, which updates the running digest accordingly. Once all blocks have been read, `ecall_finalize_hash()` finalises the digest, which is stored in EPC memory and never leaves the enclave at this stage. Once the hash has been computed, `ecall_add_evidence_entry()` is invoked to add the evidence entry, including the file metadata, the associated SHA (256 hash, and the hash chain update, to the evidence log, which will be used during the export phase to produce the signed CSV file. At the end of the forensic analysis, the app starts the closing operations. First, `ecall_close_forensic_session()` is invoked to close the forensic session. During the execution of the function, the enclave requests the closing timestamp via `ocall_get_current_time()`, which is then inserted into the CSV file.

Next, the sealing phase begins: first, the buffer size required for the sealed blob is requested via `ecall_get_sealed_data_size()`. Secondly, `ecall_seal_forensic_session()` is called, which executes the sealing process. Through `sgx_seal_data()` with the `MRENCLAVE` policy, the entire session state is encrypted and authenticated inside the enclave (including the evidence log and the ECDSA-P256 key). The resulting sealed blob is then sent outside the enclave boundary via an OCALL, and written to disk by the untrusted application as `sealed_<analyst_id>_<case_id .bin`. The `MRENCLAVE` policy binds the sealed data to the exact measurement of the enclave that produced it. That means only an instance of the same enclave binary, running on the same platform, can unseal it. Any modification to the enclave code would alter its measurement, making the sealed data permanently inaccessible.

5.4 Secure Mode Analysis

In Secure Mode, since the entire disk image must be transferred into the enclave, an additional step is required after enclave initialisation. The disk image is first read from storage into RAM as a contiguous buffer using `fread()`, so that it can subsequently be transferred into EPC memory during the ecall phase. After this, the forensic session is initialised through `ecall_init_forensic_session()`, as in Standard Mode. The overall flow of Secure Mode closely

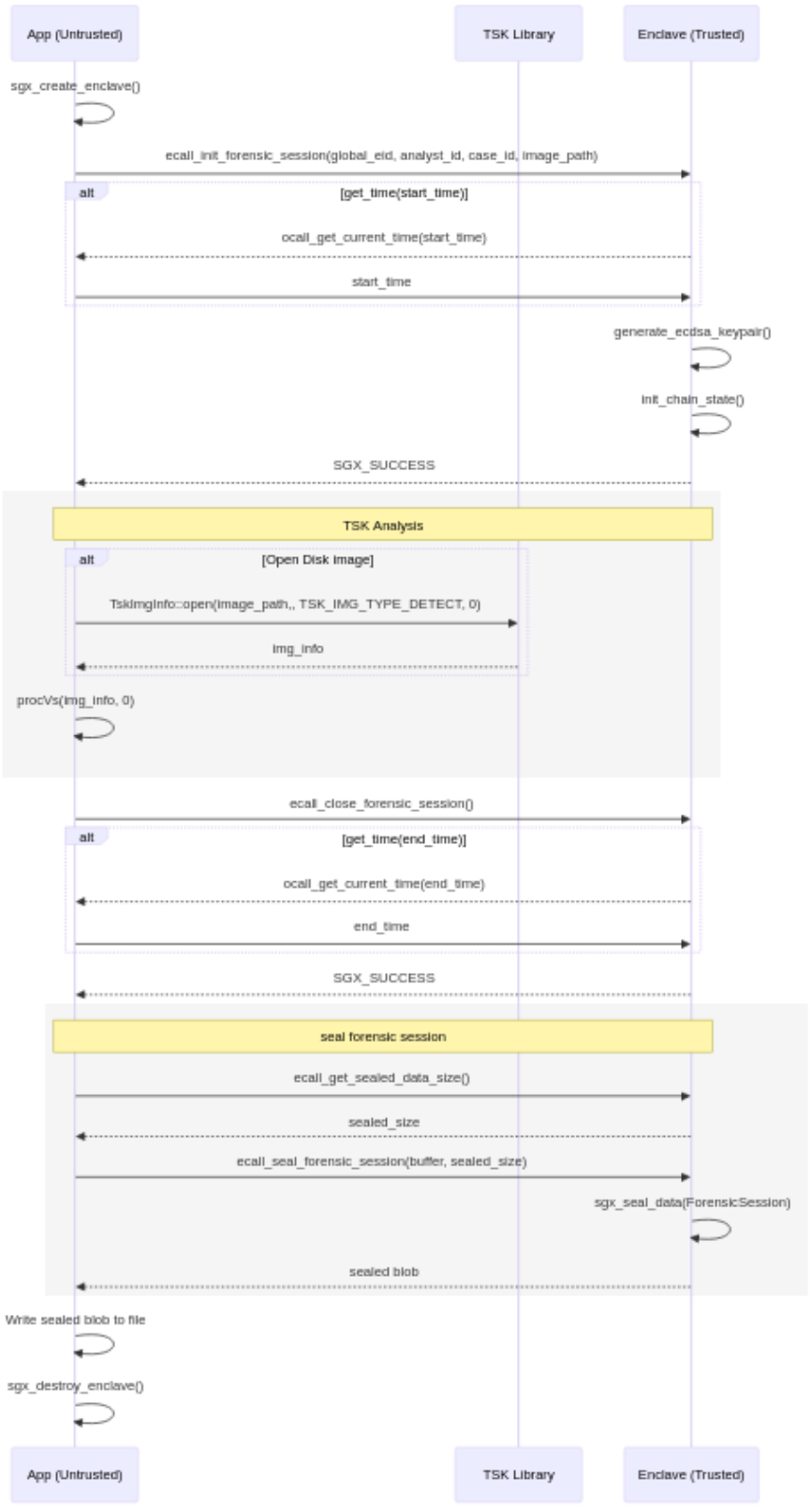

Figure 5.10. Forensic-SGX: Standard mode Analysis Flow

follows the structure illustrated in Figure 5.10, with two main differences. The first is the preliminary disk reading and loading in a buffer phase described above. The second is the replacement of the TSK analysis block with a different approach, which will be discussed in the

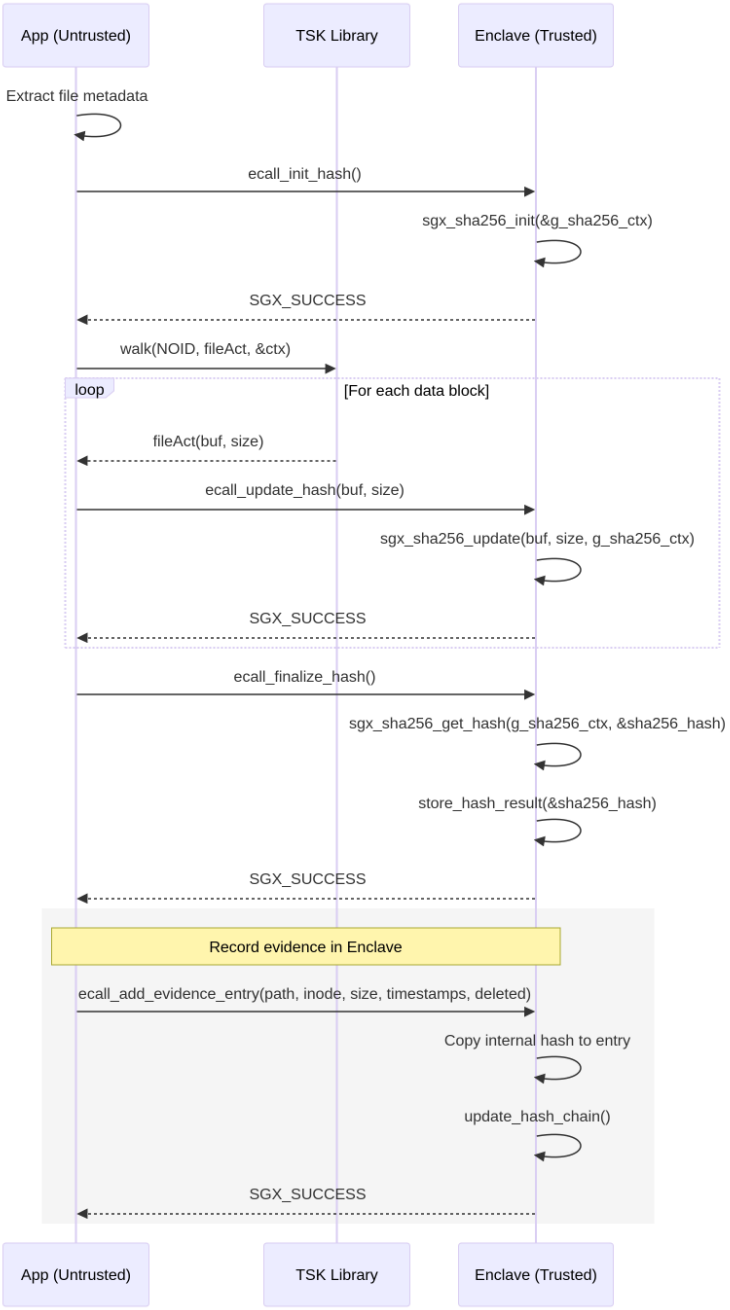

Figure 5.11. Analysis flow of ProcFile() in Forensic-SGX framework

following section. Therefore, all the components described subsequently substitute the TSK Analysis block in Figure 5.10. The first step is the loading of the disk image into the enclave, as shown in Figure 5.12. The image is delivered in chunks of 32 MB: for each chunk is invoked `ecall_load_disk_image_chunk()`, which internally calls `fs_write_chunk()` to copy the portion of data into the enclave buffer at the correct offset. When the loading phase is complete, the image buffer allocated in the untrusted App is securely wiped and freed, and then begins the analysis of the disk image via `ecall_analyse_disk_image()`, which performs the entire filesystem analysis entirely within the trusted environment. When the analysis is complete, the disk image is securely unloaded through `ecall_unload_disk_image()`, and the forensic session is subsequently closed via `ecall_close_forensic_session()`. As in Standard Mode, the session concludes with the sealing phase.

Before the present implementation, the process of loading the disk image was executed with a single ecall. However, after several tests, it was observed that in this approach, there is an excessive memory usage in the SGX enclave heap. The reason for this anomalous behaviour lies in the mechanism by which the SGX runtime handles ecall parameters and is related to how the `ecall_load_disk_image()` function was defined in the `.edl` file:

```
[in, size=image_size] const uint8_t* image_data,
uint64_t image_size);
```

That is because, as explained in section 5.2.3, when an application executes an ecall with a parameter `[in]`, the SGX runtime copies the contents of untrusted memory. More specifically, the SGX runtime allocates a temporary buffer on the enclave heap, of size equal to that specified by the `size=...` attribute, into which the input data is copied before the actual execution of the function. This buffer exists only for the duration of the ecall and is deallocated upon its termination. For this reason, in the case of a 3 GB disk image, as soon as `ecall_load_disk_image(image_buffer, size_disk_image)` is invoked, the runtime allocates a temporary buffer of 3 GB in EPC memory. Furthermore, since the data must persist beyond the duration of the ecall, the function contains a `memcpy` that copies the contents of this temporary buffer into a second buffer dynamically allocated on the enclave heap. During the execution of this ecall, two 3 GB buffers therefore coexist, requiring a total of 6 GB of EPC memory. By splitting the loading phase into 32 MB chunks instead, the temporary buffer allocated by the SGX runtime for each ecall amounts to only 32 MB.

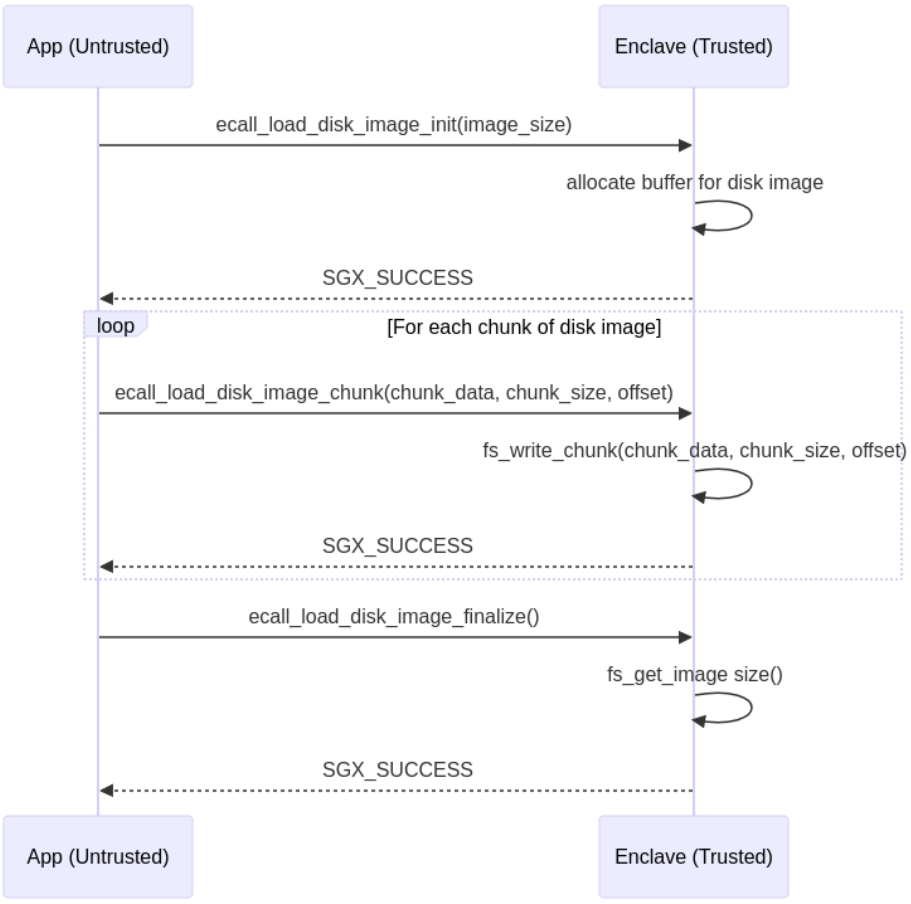

Figure 5.12. Secure Mode: Loading disk Image into Enclave

In Secure Mode, the disk analysis is executed entirely within the enclave; therefore, TSK cannot be used. The enclave is constrained in memory, and TSK is a large library with thousands of functions, data structures, and external dependencies. As documented in the Intel SGX Developer

Guide [20], several C/C++ features are not supported or only partially supported within the enclave environment. In particular, calls to shared objects, OS API calls, and system-provided C/C++ standard libraries are not supported. A trusted version of the standard libraries is instead provided with the Intel SGX SDK. TSK to execute all the operations needs the use of system calls such as `open()` and `read()` to access and read the filesystem directly, which are not available inside the SGX enclave environment, since the enclave has no direct access to operating system services.

For these reasons, a custom FAT32 parser was implemented to replicate the filesystem analysis functionality provided by TSK within the TEE. The custom parser supports only FAT32 filesystems, as this work represents a preliminary study, and is intended for comparison purposes and as a reference for future extensions. The disk analysis phase invoked by `ecall_analyse_disk_image()` is illustrated in Figure 5.13 and represents the main steps performed by the custom parser to analyse a FAT32 filesystem. After `ecall_analyse_disk_image()` is called, begins a initial validation phase.

As the first operation, the parser should verify that the disk image is not empty and that no other active session exists. It then determines if the image contains a valid FAT32 filesystem using `fs_find_fat32()`. If the FAT32 is found, it returns the byte offset of the FAT32 partition within the disk image. Subsequently, `fs_analyse()` is called with as parameters the FAT32 offset and the `file_callback_function()`. Within this function, the FAT32 parser is first initialised via `fat32_init()`. Here, the filesystem's validity is verified, and the positions of its internal data structures are computed. Specifically, it verifies the Boot Sector signature, confirms that the filesystem type is FAT32 and not FAT12 or FAT16, and calculates the volume parameters (including the position of the FAT region, the start of the Data Area, and the cluster size) necessary for the read operations. It is important to note that the validation process is divided into two distinct phases. The first one, `fs_find_fat32()`, operates at the disk level, in order to find if a FAT32 partition exists and by identifying its offset in the disk image, and secondly, `fat32_init()`, operates at the filesystem level, following a deeper validation of the Boot Sector and the internal FAT32 structures.

The nexp phase regard the filesystem processing that starts with `fat32_process_dir()`, which receives as parameters `BPB_RootClus` (that indicate the first cluster of the root directory, and typically is the number two), the current path string (initialised as `"/"`, and extended at each recursive call with the name of the current subdirectory) and the `file_callback_function()`, which is invoked each time a file is found.

A directory cluster does not contain file data, but instead, it contains a table of 32-byte directory entries. Each entry describes a filesystem object (a file or a subdirectory) and stores all the information about it, such as its name, attributes, timestamps, and the number of the first cluster where its data is located. In the case of a regular file, this cluster contains the actual file data; in the case of a subdirectory, it contains another table of directory entries. For example, the root directory cluster may contain entries such as `DOCUMENTS/`, `IMAGES/`, and `README.TXT`: the first two are subdirectories that point to other clusters containing other entries, while the third is a regular file whose data is stored in a separate cluster.

```
[cluster 2] /root
            |
            +-- DOCUMENTS/   [ ..., first_cluster=5]
            |
            +-- IMAGES/      [ ...,  first_cluster=8]
            |
            +-- README.TXT   [ ..., first_cluster=12]


[cluster 5] /root/DOCUMENTS
            |
            +-- REPORT.DOCX  [ ..., first_cluster=20]
            |
            +-- NOTES.TXT    [ ..., first_cluster=25]
```

```
[cluster 8] /root/IMAGES
            |
            +-- PHOTO.JPG    [ ..., first_cluster=30]
            [cluster 12]

[cluster 12]
            |
            +-- (file data for README.TXT)
...
```

It is this tree structure that makes `fat32_process_dir()` recursive: when an entry with attribute `ATTR_DIRECTORY` is encountered, the function calls itself on the cluster indicated by that entry, descending one level deeper into the directory tree. When a regular file is encountered instead, its SHA-256 hash is computed through `compute_sha256_chunked()`, and the `file_found_callback()` is invoked, which is responsible for adding the evidence entry (including the file metadata and the associated hash) to the evidence log via `ecall_add_evidence_entry()`. In particular, when `compute_sha256_chunked()` is invoked, it receives as parameters the first cluster of the file, the file size in bytes, and a pointer to the output buffer where the resulting hash will be written. After initialising the cryptographic context with `sgx_sha256_init()`, a loop processes the file's FAT chain. At each iteration, `fat32_get_next_cluster(cluster)` reads the FAT directly from `g_disk_image`, that is, the trusted copy of the disk image stored in EPC memory, and determines the next cluster in the chain. After that, `read_bytes(cluster_offset)` is used to read the cluster contents from the same trusted memory region, and `sgx_sha256_update(chunk)` updates the actual digest. Once all clusters have been visited, `sgx_sha256_get_hash()` finalises the computation and returns the 32-byte SHA-256 digest.

5.5 Export Phase

The export phase can be divided into four main phases:

Phase1: Unsealing The first phase involves the unsealing process. The sealed binary file, placed as a parameter to the `./app export <sealed_file.bin>` command, is read from disk and its content is sent to the enclave using `ecall_unseal_forensic_session()`. Inside the enclave, `sgx_unseal_data()` is executed: it decrypts the sealed blob and restores the forensic session to the exact state it was in at the time of sealing. If the sealed file has been tampered with, `sgx_unseal_data()` returns `SGX_ERROR_MAC_MISMATCH` and the export process is immediately aborted. The unsealing operation, when using the `MRENCLAVE` policy, can only be executed by the same enclave that created the sealed data. This policy binds the sealed data to the exact enclave measurement (MRENCLAVE), ensuring that only an enclave with identical code can successfully unseal it. In addition, the sealing key is derived from secrets stored inside the processor, meaning that the operation is also tied to the hardware platform. As a result, if the sealed file is transferred to a different machine, the unsealing operation will fail.

Phase 2: Hash Verification When the unsealing is completed, the function `ecall_verify_chain()` is invoked to verify the integrity of the evidence collected during the analysis phase. The function does not compute the hashes of the single files, but recompute the hash chain on the set of entries already present in the forensic log inside the enclave, according to the formula:

$$\texttt{hash\_chain[i]} = \texttt{SHA256(hash\_chain[i-1] | entry[i])} \tag{5.1}$$

The final result is compared with the hash chain value stored in the session at the time of sealing. If the two values match exactly, this confirms that the evidence has not been tampered with and the verification is successful. Any discrepancy indicates a possible alteration of the forensic log. As explained in the Standard Mode section 5.3, this additional verification is redundant,

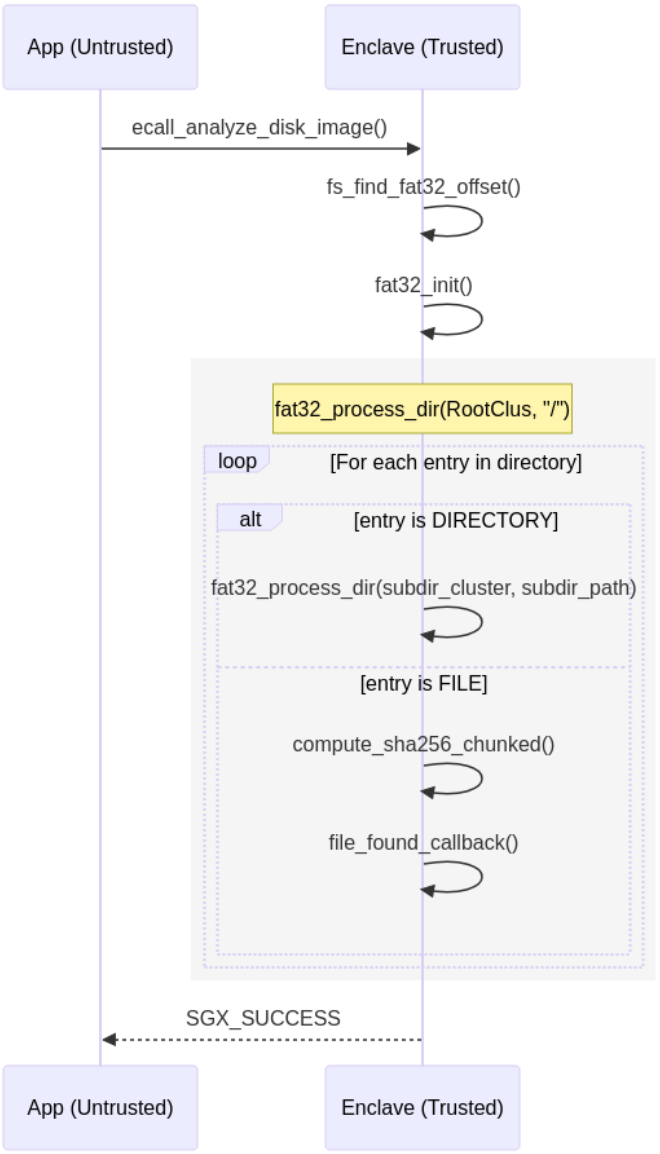

Figure 5.13. Secure Mode: Custom Parser FAT32 Disk Analysis

since the sealing operation and the ecdsa verfication are sufficient to verify whether the evidence log has been altered.

Phase 3: Signed CSV Generation and Exportation After verifying the hash chain integrity, begins the generation and export phase of the forensic report. The first operation consists of the construction of the CSV. All evidence entries in the forensic log are serialised to CSV format directly in the enclave, producing a trusted buffer containing the complete report of the analysis. In this context, serialising means transforming the data structure into rows of text. When the CSV buffer is built, its hash256 is computed through `sgx_sha256_msg()`, obtaining a digest of 32 bytes. This hash is signed with the ECDSA private key(generated during the initialisation of the enclave) via `sgx_ecdsa_sign()`, producing the pair of values `sig_r` and `sig_s` that constitute the digital signature of the CSV. The ECDSA private key must never leave the enclave.

Phase 4: DCAP Quote Generation The last phase consists of generating the DCAP attestation quote, which is necessary to prove that the CSV is generated inside an authentic SGX enclave and not by a malicious program. The first step is to invoke the function `sgx_qe_get_target_info(&qe_target_info)`, which has the scope to retrieve information about

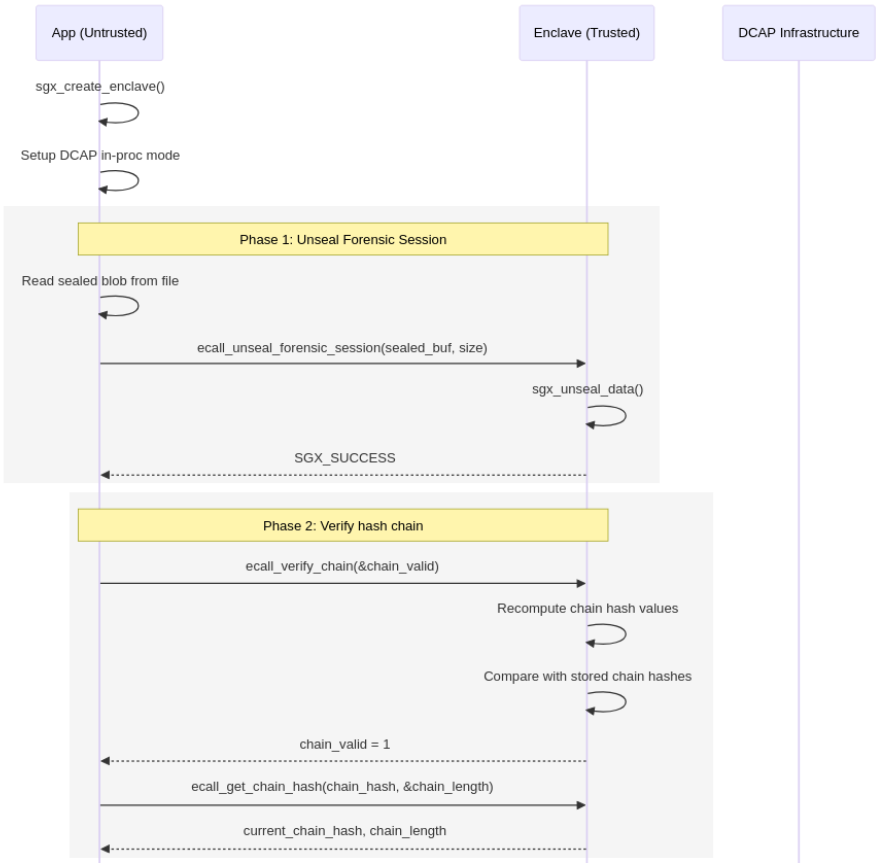

Figure 5.14. Export phase 1 of 2

the Quoting Enclave. This operation is executed by QE that is an SGX software enclave provided by Intel as part of the DCAP infrastructure.

The typical structure of `target_info` :

```
typedef struct _target_info_t
{
    sgx_measurement_t mr_enclave; //MRENCLAVE
    sgx_attributes_t attributes;
    uint8_t reserved1[2];
    sgx_config_svn_t config_svn;
    sgx_misc_select_t misc_select;
    uint8_t reserved2[8];
    sgx_config_id_t config_id;
    uint8_t reserved3[384];
} sgx\_target\_info\_t;
sgx\_target\_info\_t target_info;
```

This information is essential to correctly address the report. After the information is returned on the app side, it is called the `ecall_create_attestation_report&qe_target_info, &report` function. Inside the ecall, the `SHA256(PUBKEY_X || PUBKEY_Y` is computed. In particular, the concatenation of the keys is performed, and the result is placed on `sgx_sha256_msg` to obtain the SHA256. The result of the computation is stored in the first 32 bytes of 64 of **report_data** that has as a structure:

```
typedef struct _sgx_report_data_t {
    uint8_t d[64]; // 64 bytes totali
} sgx_report_data_t;
```

```
sgx_report_data_t report_data;
```

As the last operation of the ecall, is invoked `sgx_create_report(&qe_target_info, &report_data, &report)`. This is the function that generates the `report`. The data structure that contains the report information is as follows:

```
typedef struct _report_t
{
sgx_report_body_t body;
sgx_key_id_t key_id;
sgx_mac_t mac;
} sgx_report_t;
sgx_report_t report;
```

The body contains all the information of the enclave that creates the report, such as `MRENCLAVE`, `MRIGNER`, `ISVSVN`, `ATTRIBUTES`,... and the `report_data`. The structure is as follows:

```
typedef struct _report_body_t
{
    sgx_cpu_svn_t cpu_svn;
    sgx_misc_select_t misc_select;
    uint8_t reserved1[12];
    sgx_isvext_prod_id_t isv_ext_prod_id;
    sgx_attributes_t attributes;
    sgx_measurement_t \textbf{mr_enclave};
    uint8_t reserved2[32];
    sgx_measurement_t \textbf{mr_signer};
    uint8_t reserved3[32];
    sgx_config_id_t config_id;
    sgx_prod_id_t isv_prod_id;
    sgx_isv_svn_t isv_svn;
    sgx_config_svn_t config_svn;
    uint8_t reserved4[42];
    sgx_isvfamily_id_t isv_family_id;
    sgx_report_data_t \textbf{report_data};
} sgx_report_body_t;
sgx_report_body_t body;
```

The others paramters are the `key_id`, that is the "value for key wear-out protection". It is the root key that is used to generate other keys. More specifically, it is used as input to derive the effective key to compute the MAC for the report. This is used to avoid reusing the same keys. The last parameter is the `mac` that is the cmac (AES-CMAC) value of the report data using `report key`. The Mac is needed to guarantee that the report is generated by a trusted enclave SGX, and the data on the report has not been modified. An illustration of this part is shown in 5.15. This part is important for the framework, because here the ECDSA PK is bound to the trusted enclave, and could be used to verify the ECDSA signature.

After obtaining the report from the enclave, the app side calls `sgx_qe_get_quote_size(&quote_size)` to obtain the size of the quote in order to allocate a buffer for the quote that will be generated. Once the buffer is allocated, `sgx_qe_get_quote(&report, quote_size,)` is called. The Quoting Enclave (QE) receives the report, verifies that it originates from an authentic enclave using the certificate chain retrieved from the Trusted Hardware Identity Management (THIM) service (THIM corresponds to the PCCS for Azure, but is handled directly by Microsoft), and then produces the DCAP quote. The DCAP quote has the following structure:

```
typedef struct _sgx_quote3_t {
    sgx_quote_header_t header;
    sgx_report_body_t report_body;
    uint32_t signature_data_len; // The length of the signature_data. Varies
        depending on the type of sign_type.
```

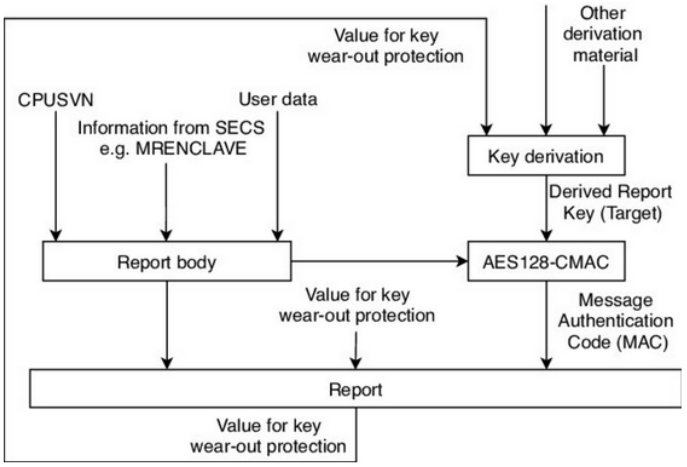

Figure 5.15. Structure of a Report generated using EREPORT instruction (Source: Remote Attestation in Intel SGX [59])

```
    uint8_t signature_data[]; ///< 436: Contains the variable length
        containing the quote signature and support data for the signature.

} sgx_quote3_t;
```

Since the DCAP quote is a binary structure containing non-printable bytes, it cannot be safely embedded directly into a text-based signature file. Therefore, the quote is encoded using Base64 before being written to the .sig file. During verification, the Base64 string is decoded to reconstruct the original binary quote buffer, which is then passed to the SGX quote verification library.

More specifically, the steps executed in phase 4 are shown in more detail in Figure 5.16. The part implemented in the framework is the one on the right, including the QE and Intel SGX application, but behind it are all the steps required to retrieve the keys necessary to perform remote attestation. This is an example of the Intel SGX DCAP remote attestation flow, and this part is the same for Azure and for a local deployment. When the QE receives the request for its identity information via `sgx_get_target_info()`, which is needed to know where to address the report, the QE sends to the Quote Provider Library (QPL) its identifier (`pck_cert_id`), in order to receive the PCK (Provisioning Certification Key) certificate for the requesting platform. The PCK is the signing key available to the Provisioning Certification Enclave for signing certificate-like QE report structures. The QPL responds with the TCB, which describes the hardware and firmware version of the platform, and the QE certification data, defined as the data required to verify the QE Report Signature [60]. The QE then generates its Attestation Key (AK), an ECDSA key pair, where `pk(AK)` is the public key that will be used by the QE to sign the quote, which contains the measurements and identity of the application enclave. Finally, the QE sends its `pk(AK)`, QE identity, and TCB to the PCE, with the purpose of certifying that this is the legitimate public key of the QE. The PCE generates the PCK and creates a certificate that binds `pk(AK)` to the platform, serving as cryptographic proof that the AK belongs to a legitimate QE running on authentic Intel hardware. For further details on how to configure the Intel QPL to use the Azure Trusted Hardware Identity Management (THIM) service, refer to [61]. It is worth noting that Azure THIM and Intel PCS provide different TCB baseline versions. While Intel PCS updates the TCB baseline immediately upon the release of a new version, Azure THIM adopts a more gradual approach, allowing customers to update their platforms at their own pace without experiencing attestation failures. As a consequence, the TCB baseline retrieved from Azure THIM may refer to a slightly older version compared to Intel's latest release. In the context of this project, this difference has no practical impact on the correctness of the attestation process, but should be taken into account in a production deployment where the most up-to-date TCB baseline is required for compliance or security reasons.

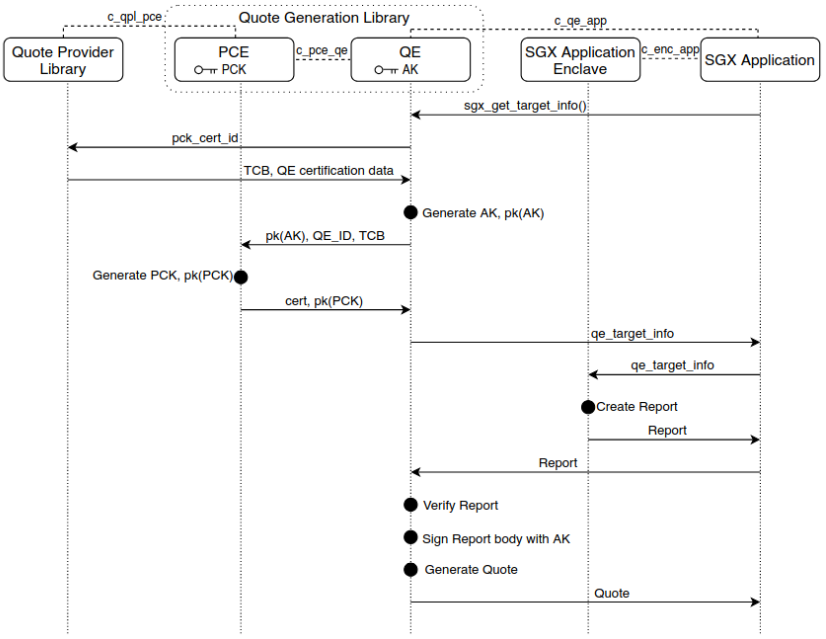

Figure 5.16. Intel DCAP Infrastructure: quote generation flow (Source: [60])

File Generation: .csv and .sig

After generating the DCAP quote, the untrusted component writes the CSV file to disk after receiving the buffer from the enclave and then creates the `.sig` file that contains all the elements necessary for verification: the `SHA256` of the CSV, the ECDSA signature (`r`, `s`), the public key (`pk_x`, `pk_y`), and the DCAP quote in base64 format. Both the `.csv` and `.sig` files must be stored in a secure location before transmission and transmitted via a secure channel when needed by an investigation, because the CSV file is not directly signed but is instead accompanied by a separate `.sig` file, and for that reason, it is essential that the two files are always kept together. Losing the `.sig` file would make it impossible to verify the integrity and authenticity of the forensic report, as all cryptographic evidences are contained exclusively within it. For this reason, both files should be treated as a single forensic unit and stored or transmitted together in a secure manner. An output examples of the two files produced is shown in the Figure 5.18 and 5.19. The CTime are zeros because FAT32 does not support change time(CTime) as a timestamp, and as consequence is set to zero:

5.6 Verifier Phase

This phase is used to verify that everything has been executed correctly and that no value essential to the verification process has been compromised. In this framework, it is implemented as a command-line tool, acting as a verifier simulator.

Step 1: Parsing and Decoding of the Signature File As a first step, the verifier parses the `.sig` file in order to read and extract all the relevant fields: `CSV_SHA256`, `ECDSA_R`, `ECDSA_S`, `PUBKEY_X`, `PUBKEY_Y`, and `DCAP_QUOTE_B64`. It also verifies that both the DCAP quote and the public key are present, since without either of them, the verification cannot proceed.

As a second step, it is necessary to decode the DCAP quote from its base64 representation into raw bytes, since it is necessary to access the `sgx_quote3_t` structure and extract the data contained in the `report_body`. This structure includes the `MRENCLAVE` and `MRSIGNER` fields, which are the parameters that identify the enclave that generated the quote. Inside there is also the `report_data` field, which contains the `SHA256(pk_x || pk_y)` hash that was added by the enclave

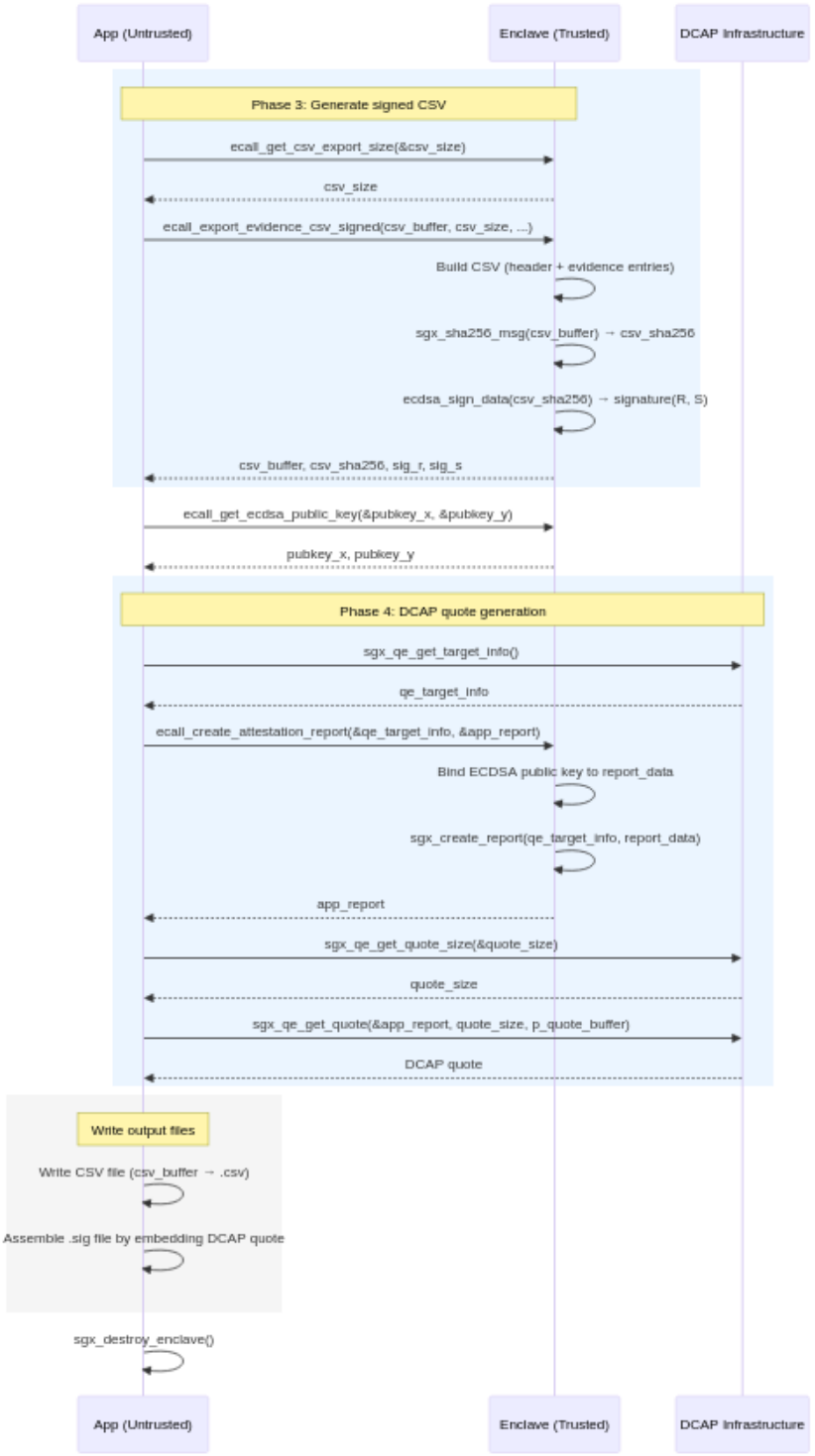

Figure 5.17. Export phase 2 of 2

during the `ecall_create_attestation_report()`. These values are essential for the subsequent verification steps.

Step 2: Quote Verification In the next step, the verification of the DCAP quote is performed through the Intel SGX ECDSA Quote Verification Library (QVL) that supports two verification modes: a trusted mode that relies on a Quote Verification Enclave (QvE), where the verification result is cryptographically protected and attestable, and for this mode is necessary

Analysis Report:
Analyst: Analyst
Case ID: 001
Image: disk/disk_3_500.img
Analysis Start: 2026-03-06T23:29:29.376Z
Analysis End: 2026-03-06T23:30:03.749Z

Timestamp	Path	Inode	Size	SHA256	ChainHash	MTime	ATime	CTime	CRTime	Deleted
2026-03-06T23:30:0	/documents/WORK/R	69	219598	4ab5f60ce2409890c	05c06e992c0d3f9b4	1772826994	1772755200	0	1772826994	0
2026-03-06T23:30:0	/documents/WORK/R	123	203160	7c634af5931653c27	3c80f417134dbeec7	1772826994	1772755200	0	1772826994	0
2026-03-06T23:30:0	/documents/WORK/R	173	229264	be9569ef7efeea04c	c3375f9dd1c9f2760	1772826994	1772755200	0	1772826994	0
2026-03-06T23:30:0	/documents/WORK/R	229	223247	2b40e31bd216d451	4b78bc1d3f11fa402	1772826994	1772755200	0	1772826994	0
2026-03-06T23:30:0	/documents/WORK/R	284	212770	1218b37635ed9949	0a4f53636e87976c3	1772826994	1772755200	0	1772826994	0

Figure 5.18. CSV File Example Output

```
# Forensic-SGX
#
#
CSV_FILE=forensic_log_20260306_233007.csv
CSV_SIZE=116704
CSV_SHA256=ec528d1d04825a9a6072668e8d850b19bbe6f40b8088b6b4416ed17f7f372c29
ECDSA_R=f67b2ffd9ff88a26e708f705182c0450102aecfc24bb4f3623600bca50a217b7
ECDSA_S=3336065a81bcfa075934e784881f71d4fdd7be0fb8d29c1ef3322bc0e023d689
PUBKEY_X=8c58742301fecf348b42f95198aa342d09df6df5a07fffe01975576809fc6e7b
PUBKEY_Y=1ef47fa1dddd0907223f9d9b05ae8c720848c896ebb064ab135d42a1f63722d4
DCAP_QUOTE_SIZE=4730
DCAP_QUOTE_B64= ...
```

Figure 5.19. Signature File Example Output

to have an Intel SGX environment, and an untrusted mode, where the verification is performed directly by the library in userspace without invoking any enclave.

Since this implementation acts as a verification simulator to demonstrate the correctness of the framework, the additional security guarantees provided by the QvE are not strictly necessary. In a production deployment, where the integrity of the verification process itself must also be guaranteed, the trusted QvE mode should be considered.

The first operation executed is a call to `tee_get_supplemental_data_version_and_size()`, which allows the verifier to retrieve the latest version and size of the supplemental data associated with the quote. This information is used for two purposes: first, to determine the correct buffer size to allocate for the supplemental data; second, to decide which version of the supplemental data structure to use based on the latest version returned by the API.

As a second step, the quote verification is performed through the `tee_verify_quote()` API, which is called as follows:

```
tee_verify_quote(
    decoded_quote,
    (uint32_t)decoded_len,
    NULL, /* collateral downloaded automatically /
    current_time, / timestamp for certificate expiration check /
    &collateral_expiration_status,
    &quote_verification_result,
    NULL, / no qve_report_info: untrusted QVL mode */
    &supp_data
);
```

The second `NULL` parameter, corresponding to `qve_report_info`, indicates that no QvE is involved, confirming that the untrusted QVL mode is used.

The `current_time` parameter is set using `time(NULL)`, which retrieves the current system time. In a production environment, a trusted time source should be used, since the system clock could potentially be manipulated.

Internally, the API downloads all the necessary collateral (PCK certificate chain, TCB info, and Certificate Revocation Lists) automatically from the configured endpoint, which in this case is the Azure THIM service. It then performs the following checks: it verifies that the quote was

signed by a key that belongs to Intel's root of trust, verifies the ECDSA signature on the quote body, checks the TCB level of the platform against the requirements specified in the TCB info, and uses the current time to verify that all certificates and CRLs are not expired. The results are finally written to `quote_verification_result`, which indicates the overall verification output, and to
textttcollateral_expiration_status, which specifies if the collateral is still valid.

Step 3: ECDSA PK Binding Check The final check conducted by the verifier consists of verifying that the ECDSA public key contained in the `.sig` file was genuinely generated inside the attested SGX enclave. The public key coordinates are read from the `.sig` file and converted from hexadecimal to bytes. Subsequently, `SHA-256(PUBKEY_X || PUBKEY_Y)` is computed, that is the same operation executed by the enclave during the quote generation phase, as illustrated in Figure 5.17. The resulting hash is then compared with the `report_data` field extracted by the quote body. If the two values match, that means that the public key was generated inside the specific attested enclave and has not been substituted or tampered with. If they do not match, the `.sig` file must be considered compromised, and the evidence cannot be trusted.

This check is essential, since without it an attacker could generate their own ECDSA key pair using a tool such as OpenSSL,and produce a forged CSV file, and sign it with his own private key. He could then replace the public key, the signature, and the CSV hash in the `.sig` file, while keeping the original DCAP quote intact. The binding check prevents this attack: by comparing `SHA-256(PUBKEY_X || PUBKEY_Y)` with the `report_data` field that is embedded in the quote, the verifier can detect if the public key in the `.sig` file does not correspond to the one generated inside the enclave, and if the two values not correspond, that means there is a compromise. This is possible because the quote is signed by the QE, and therefore the `report_data` field cannot be modified without invalidating the quote signature itself (a check that is performed in the previous verification step)

Chapter 6

Testing and Limitations

In this section, all testing activities executed on the framework are described to evaluate the correctness, robustness, security, and performance of the realised framework. The objective is not only to verify that everything works correctly, but most importantly, to demonstrate that the security guarantees offered by the integration with Intel SGX resist against realistic attack scenarios, despite some significant limitations that still exist. All tests were executed on the same virtual machine used for the framework development: an Azure Standard DC2s v3 VM running Ubuntu 22.04 LTS, with two Intel SGX-enabled vCPUs and 16 GiB of memory. The framework was compiled and tested in hardware mode (`SGX_MODE=HW`), which is the only configuration where Intel SGX security features are effectively guaranteed by hardware. The tests were also executed with `DisableDebug=0`, which means that debug mode is enabled. This choice is due to the possibility of inspecting the internal state of the enclave during development, although it constitutes a critical vulnerability. In a real deployment, it is important to set `DisableDebug=1` and compile with `SGX_DEBUG=0`, which prevents debugger access to the enclave memory contents [20].

6.1 Input Validation and Robustness Tests

This category of tests verifies that the framework handles anomalous, malformed, or potentially malicious input at both the host application and enclave levels. The goal is to ensure that no unexpected input can cause crashes, internal state corruption, or undefined behaviour. First, the framework is designed so that if any required fields are left empty (such as `analysis_id`, `case_id`, `path`), an explicit error is returned and the process terminates before starting the analysis. Second, to avoid buffer overflow issues, was implemented a function called `safe_strcpy()`, which silently truncates inputs that exceed a maximum length defined in the code, thus preventing any out-of-bounds writes to the buffer.

For test log management, a static array with a maximum of 50,000 entries was chosen. In the case that the analysed disk contains more than 50,000 files, the error `SGX_ERROR_OUT_OF_MEMORY` is returned, but the entries recorded up to that point are not lost, and the process can conclude normally, producing a valid report with the collected entries.

TSK analysis was tested with various types of Disk images, including corrupted or invalid images generated with random data. In all cases, the framework has handled the errors in a controlled manner, without producing crashes. Finally, it was verified that inputs with special or potentially malicious characters were handled correctly, preventing crashes and the extraction or disclosure of sensitive data. In particular, shell injection, path traversal, null byte injection, and newline injection scenarios were tested in the `analyst_id` and `case_id` parameters. It is important to note that parameters passed to the CLI are treated by the framework as simple C strings and not as executable commands: even if a parameter contained a sequence such as `"$(...)"` or `"; rm -rf /"`, this would be passed to the program as literal text and never interpreted as an instruction. In an enclave, such strings are copied into memory using the `[in, string]` mechanism that is defined in the EDL, which performs a secure copy into trusted memory without any execution of

the contents. Furthermore, since the ECDSA private key is generated and resides exclusively in the EPC, no input parameter can access or extract it, as there is no ECALL that exposes it to the untrusted domain.

6.2 Sealing/Unsealing Integrity Tests

This section explains the results of the tests applied to verify the integrity and confidentiality guarantees provided by the Intel SGX sealing mechanism. As mentioned in previous sections, sealing is the mechanism through which the enclave encrypts and authenticates the entire state of the forensic session before writing it to disk. Since the sealed blob resides in the untrusted domain, it is essential to detect any tampering attempt at the moment of unsealing. Within the framework, unsealing is executed using the `sgx_unseal_data()` function, which verifies the integrity of the blob and allows the session to be restored only if the authentication MAC is valid.

To verify that everything has been implemented correctly, several tests were performed, both with intact and tampered sealed blobs. It was found that every time, even a single byte is modified, whether it is the first, the last, or any other position within the blob, tampering is detected during unsealing. The result is the return of the error `SGX_ERROR_MAC_MISMATCH`, which indicates that the authentication MAC does not match the contents of the blob. It can therefore be concluded that **tamper detection** is implemented correctly.

Subsequently, was tested a fundamental property of the sealing policy adopted by the framework: with the `MRENCLAVE` policy, the sealed blob can only be unsealed by the same enclave instance that produced it, identified by its `MRENCLAVE` cryptographic hash. Because C/C++ is nondeterministic by default, even a recompilation of the same source code without any modifications produces a different binary and therefore a different `MRENCLAVE` value. Tests verified that attempting to unseal with a recompiled binary produces `SGX_ERROR_MAC_MISMATCH`, confirming that the policy is being applied correctly. This property ensures that sealed data cannot be read by a modified or replaced version of the enclave.

Another test that could be verified is that a sealed blob cannot be unsealed on a machine other than the one on which it was produced. This property is guaranteed by the fact that the Intel SGX sealing key is derived from specific cryptographic keys of the hardware, using the `EGETKEY`, and for that reason, is not transferable between different platforms. However, this test was not performed in practice, as the framework was developed and tested on a single Azure virtual machine. Without a second VM with SGX enabled, to which to transfer the sealed blob, it was not possible to verify this behaviour. However, the property is guaranteed by the design of the Intel SGX sealing mechanism and is documented in the official Intel documentation [20].

6.2.1 Rollback Attack Vulnerability

Finally, a known vulnerability has been documented in the framework in its current configuration: the **rollback attack**. Since the Intel SGX sealing mechanism does not include monotonic counters, it is possible to replace a current sealed blob with a previous version of the same blob without the framework detecting the change. An attacker with access to the filesystem could preserve a copy of an older sealed blob and replace the current blob with it; the `export` command would proceed correctly with the data from the previous session, without returning any errors or warnings. This attack was successful during testing, confirming the vulnerability. A possible mitigation would have been to use SGX monotonic counters via `sgx_increment_monotonic_counter()`, which binds the sealed blob to a non-decreasing value. However, as documented in the official Intel note [62], this API has been removed from the Intel SGX SDK for Linux starting with version 2.9, along with the entire Platform Service Enclave (PSE). The alternative mitigation recommended by Intel for Linux environments is to use the Trusted Platform Module (TPM) as an external source of trusted counters.

6.3 Hash Chain Integrity Tests

In this category, the results of tests carried out have been analysed and reported, verifying that the hash chain mechanism implemented within the enclave guarantees the integrity and ordering of the evidence collected during analysis. Each entry is linked to the previous one according to the formula:

$$\texttt{chain\_hash}_i = \text{SHA-256}(\texttt{chain\_hash}_{i-1} \,\|\, \texttt{file\_hash}_i \,\|\, i) \tag{6.1}$$

This structure guarantees that any changes, deletions, insertions, or reordering of entries in the exported CSV are detected during verification. Within the framework, and specifically through the `export` command, a control is performed that recalculates the entire hash chain within the enclave, comparing the calculated value with the stored values. If the results match, it means the chain has been successfully verified on all registered entries, and the process can proceed with the rest of the execution. To verify that everything was implemented correctly, a hash chain validation was performed directly on the exported CSV file using a Python script. The script reads the `SHA256` and `ChainHash` columns for each row, recalculates the expected value of the hash chain according to the defined formula, and compares it with the value in the CSV. It has been found that when a SHA-256 value of an entry is changed, the change is detected by the hash chain verification, since any change at one point in the chain alters the value of all subsequent entries. This ensures **SHA-256 tampering detection**. The same detection occurs when deleting an entry, exchanging two lines (whether adjacent or not), or duplicating or inserting a new entry. This confirms that the hash chain correctly detects any alteration in the number or sorting of entries in the exported CSV.

6.4 ECDSA Signature Tests

Regarding this section, the tests performed aimed to verify the guarantees of authenticity and integrity provided by the ECDSA-P256 digital signature mechanism implemented in the framework. As described previously, at the end of each forensic session, the enclave generates a CSV containing the evidence log, calculates its SHA-256 digest, and signs it with the ECDSA-P256 private key that was generated internally during session initialisation. The private key never leaves the enclave, while the public key is exported to the `.sig` file along with the signature components (R, S) and the DCAP quote.

First, a test was performed to verify the ECDSA signature: after executing the `export` command, was run a script that, by reading the `.sig` file, reconstructs the ECDSA-P256 public key from the X and Y coordinates, recalculates the SHA-256 digest of the CSV and mathematically verifies the signature. The test produced the expected result, confirming that the signature generated by the enclave is valid and externally verifiable without requiring access to the enclave itself. Another easily verifiable test was the CSV Hash Verification, where it was enough to calculate the SHA-256 digest of the CSV using `sha256sum` and verify that it matches the `CSV_SHA256` value stored in the `.sig` file. As counter-evidence, the CSV was modified after it was signed. As expected, the result has highlighted that the SHA-256 digest of the modified CSV no longer corresponds to the value `CSV_SHA256` present in the `.sig` file, and the signature verification consequently fails. This confirms that any alteration of the CSV content, even minimal, is detectable, thus ensuring the ECDSA is correctly implemented.

6.4.1 ECDSA Signature Forgery and the Role of DCAP

During testing, a significant vulnerability was found in the framework when DCAP attestation is not used. In practice, it is demonstrated that if an attacker is able to generate his own ECDSA-P256 key pair, he can sign the CSV file with his own private key and replace the `ECDSA_R`, `ECDSA_S`, `PUBKEY_X` and `PUBKEY_Y` fields in the `.sig` file with the values corresponding to the attacker's key. The checks adopted previously are limited to checking the mathematical correctness of the signature with respect to the public key present in the file: testing such a case, the script would return that the signature is valid, since it is mathematically correct, but it was produced by an

external key and not by the enclave. This confirms that using the ECDSA signature alone is not sufficient to guarantee that the CSV was produced by an authentic SGX enclave. Mitigation for this vulnerability is provided by DCAP remote attestation: the DCAP quote in the `.sig` file cryptographically binds the ECDSA public key to the enclave identity via the `report_data = SHA-256(PUBKEY_X || PUBKEY_Y)` field. A verifier validating the DCAP quote can then ensure that the public key in the `.sig` file was generated within an authentic SGX enclave with a specific `MRENCLAVE`, making the forgery attack detectable. For this reason, DCAP quota verification should be considered a mandatory step in the forensic validation process of the report produced by the framework.

6.5 DCAP Remote Attestation Testing

This category of tests verifies the guarantees provided by the DCAP remote attestation mechanism integrated into the framework. The DCAP quote cryptographically binds the ECDSA public key of the session to the enclave identity through the `report_data = SHA-256(PUBKEY_X || PUBKEY_Y)` field, which is signed by the Quoting Enclave (QE3) with Intel attestation keys. To verify that quote generation was successful, the steps for quote verification provided by Intel have been used, with some modifications. After running the `export` command, the DCAP quote is generated and embedded in the `.sig` file. To perform the verification, the QVL (Quote Verification Library) is invoked, which operates in untrusted mode and checks the validity of the quote, the Intel certificate chain, and the state of the platform's TCB. The test produced the expected result, confirming the success of the verification quote.

As a possible result of the quote verification, there are: `TEE_QV_RESULT_OK` that indicates that the quote is valid and the collateral is up to date; `TEE_QV_RESULT_OUT_OF_DATE` or `TEE_QV_RESULT_CONFIG_NEEDED`, which indicates that the quote is still valid, but the platform TCB may not be at the latest version. Results such as `TEE_QV_RESULT_INVALID_SIGNATURE` or `TEE_QV_RESULT_REVOKED` indicate instead a failure that interrupts the verification since the quote cannot be trusted. Furthermore, supplemental data is reviewed to obtain a list of Security Advisory IDs that affect the platform, with the scope to identify known vulnerabilities that may impact the TCB. During testing, the verification quote phase produced the following diagnostic output:

```
Note: Quote is valid but platform TCB may be out-of-date.
Supplemental data Major Version: 3
Supplemental data Minor Version: 3
Advisory ID: INTEL-SA-00615, INTEL-SA-00657
```

The result confirms that the DCAP quote was successfully verified and that the enclave is running on genuine Intel SGX hardware. The result `OUT_OF_DATE` status, with the reported advisory IDs, is a consequence of the THIM service that adopts a more conservative approach to TCB baseline updates compared to the Intel PCS. As documented by [61], THIM intentionally delays the adoption of new TCB baselines in order to allow customers to meet the updated requirements at their own pace, avoiding disruptions due to forced updates. As a result, the platform TCB may appear outdated with respect to Intel's latest baseline, even though the quote remains valid and the attestation is successful. This behaviour does not represent a security issue for the purposes of this framework, as the quote signature and the public key binding remain fully verified.

The binding between the public key and the quote has also been investigated. After the extractions of the `PUBKEY_X` and `PUBKEY_Y` coordinates from the `.sig` file, it is possible to calculate `SHA-256(PUBKEY_X || PUBKEY_Y)`, and compare the result with the first 32 bytes of the `report_data` field in the quote. It was confirmed that the two values match, proving that the public key used to sign the CSV is indeed the one generated within the enclave and attested by the SGX hardware.

During testing, it was also verified that a DCAP quote generated in a different session cannot be used to attest to the public key of another session. A custom hybrid `.sig` was built by combining the public key and ECDSA signature components of session A with the DCAP quote of session B, produced by recompiling the enclave. The test confirms that it was detected a

binding mismatch: the `report_data` field in B's quote contains `SHA-256(PUBKEY_B)`, which does not match `SHA-256(PUBKEY_A)` calculated from the public key in the `.sig` file. Furthermore, there is also confirmation that the binding between public key and quote is robust: even a valid quote produced by authentic SGX hardware is unusable if it does not match the public key of the session it is intended to attest.

6.6 Vulnerabilities and Limitations

During the testing activities, was adopted an approach that was not limited to verifying the standard operating conditions but also included the analysis of anomalous situations. Some of these are due to conscious implementation choices related to the development and prototyping context, while others are architectural limitations of Intel SGX. As discussed previously, known vulnerabilities in this framework include **Rollback Attacks**, discussed in Section 6.2.1, and **ECDSA Forgery**, discussed in Section 6.4.1, which is not actually a critical vulnerability when using the DCAP framework. Analysing vulnerabilities from a forensic point of view, one of the most relevant concerns is **Time Manipulation Attacks**. As the timestamps in the CSV file are derived entirely from the operating system via OCALL, if the OS is compromised, it could modify these values before they reach the enclave. An attacker with sufficient privileges on the operating system, or via an `LD_PRELOAD` hook, that is a Linux technique that allows a custom shared library to be loaded before all others, enabling the interception and replacement of standard library functions without modifying the application. In the context of SGX, an attacker with root privileges could exploit this mechanism to intercept OCALLs or manipulate data in transit between the enclave and the untrusted application, and could return arbitrary timestamps to the enclave, forging the forensic timeline without any possibility of being detected by the framework. This limitation is shared with most SGX frameworks on Linux, where `sgx_get_trusted_time()` has been removed from the SDK since version 2.9. One possible mitigation is to use T3E or an authenticated NTPSec service as the trusted time source.

Other vulnerabilities, as mentioned, involve the use of simulation mode and debug mode, both of which must be avoided in a production context to ensure the framework's security guarantees. In simulation mode, the sealing keys are not tied to the physical CPU, while in debug mode, the enclave memory is inspectable via a debugger, making it possible to extract the ECDSA private key.

Evidence Manipulation: Similar to the Time Manipulation Attack, evidence metadata is also subject to possible alteration before reaching the enclave. In standard mode, file metadata is extracted from TSK in the untrusted domain and transmitted to the enclave via ECALL. Since the enclave does not have direct access to the disk image, it cannot verify the correctness of this metadata, because a compromised operating system could alter the path, timestamp, or the deletion state, but also add file entries or omit existing files from the directory tree walk, before the data reaches the trusted boundary. In all these cases, the enclave would embed false data into the hash chain and CSV without being able to detect it. This limitation is partially avoided in `analyse-secure` mode, where the entire image is loaded into the enclave, and parsing occurs entirely in the trusted domain. It is partially avoided because the disk image resides in the SSD, which is placed in the untrusted part, and could also be affected by alteration before reaching the enclave.

A limitation also concerns the sealing policy adopted. As discussed previously, the `MRENCLAVE` policy guarantees that the sealed blob can only be unsealed by the same enclave instance that produced it. While this property provides an important security guarantee, it also represents a possible limitation in a forensic context: any framework update, like a simple bug fix, produces a binary with a different `MRENCLAVE` value, making all sessions sealed with the previous version permanently inaccessible. In an investigative scenario, where investigations can continue for months or years, and software updates are inevitable, this limitation could result in the loss of access to forensic evidence that has already been collected. A possible mitigation is to adopt the `MRSIGNER` policy, that is, the hash of the pk that is used to sign the enclave (`enclave_private_key.pem typically`), allowing updated versions of the framework to unseal sessions produced by previous versions. If the same key is always used between different versions of the enclave, the MRSIGNER

remains unchanged. However, this choice involves a reduction of security, because any enclave signed by the same developer could access the sealed data.

Limitation on FAT32 file system: Both the standard mode and the secure mode are able to detect deleted files, but there is a limitation of FAT32 that requires the implementation of other techniques to recover certain deleted files. When a deleted file is overwritten by another file, it is not possible for either TSK or the custom FAT32 parser to recover the file in question.

During testing, a FAT32 disk image was created in which 50 files were deleted from a directory, and then new files were created in the same directory. As a conclusion of the final testing session, it was found that only 44 of the deleted files were detected by the two modes, while 6 of them were overwritten.

In order to detect the remaining files, a possible solution is to use a file carving tool, which does not rely on filesystem metadata but instead reconstructs files directly from the raw disk data, ignoring the filesystem structure. However, with this technique, it is not possible to recover the original filename, the path, or the timestamps, which reduces the forensic value of the recovered files. This is not a limitation of the framework, but rather an issue inherent to the FAT32 file system itself.

6.7 Performance Tests

For performance tests, several scenarios were analysed. The tests presented in the following sections were performed by repeating both the analysis and the export phases of each disk image for 10 iterations; therefore, the values reported in the results represent the average of the observed execution times across all iterations. Regarding the choice of file system, FAT32 was adopted for all tests, as it represents the only file system currently supported by the Secure Mode for analysis.

6.7.1 Impact of File Count

The first phase of testing evaluates the performance difference of a 7168 MB (7 GB) disk image containing 500, 5,000, and 50,000 files, respectively. The total size of the files stored in the image is approximately 4200–4300 MB. For comparison purposes, all available modes of the framework were evaluated: the Standard Mode and the Secure Mode, as well as a baseline comparison using TSK outside of Intel SGX. For the two framework modes, the execution times of both the analysis and export phases were considered, with their combined total taken as the overall execution time of the analysis and extraction of the evidence. As discussed in the previous chapters, the analysis phase includes initialising the forensic session, analysing the disk image, and sealing the session, while the export phase involves unsealing the session, verifying the integrity of the hash chain, DCAP quote generation and exporting the signed CSV file along with the related `.sig` file containing the ECDSA signature and the DCAP attestation quote. The purpose of these tests is to observe how the number of entries within a disk image affects the total processing time across the three modes analysed, and to verify whether this factor represents a significant variable in the overall execution time.

Before discussing the results obtained, it is important to make a clarification regarding the Secure Mode. By running the following command:

```
sudo dmesg | grep -i epc
```

it is possible to verify the amount of EPC memory physically available on the system. In the case of the Azure VM used for testing, the output confirms a total EPC section of 8GB (`EPC section 0x4c0000000-0x6bfffffff`). However, it is crucial to understand that this value does not represent the amount of memory available to the user's enclave. In fact, a portion of the SGX system itself, or some portion used by the SGX driver, or by enclave metadata, should be taken into consideration before setting the `HeapMaxSize`. More specifically, approximately 30 MB of the EPC should be used to maintain SGX metadata [63]. As a result, the memory effectively

available to an enclave is lower than the nominal 8GB reported by the system. This explains why setting `HeapMaxSize` to `0x200000000` (8GB) immediately takes to EPC paging already during the enclave initialisation phase, since SGX must allocate and cryptographically measure all declared heap pages at startup, regardless of how much memory will actually be used. When the EPC capacity is exceeded, the Intel SGX processor enables the EPC paging mechanism: the extra pages are encrypted and written to conventional RAM, to then be reloaded and decrypted whenever the enclave needs them. This process introduces the worst performance, even though it can guarantee data confidentiality thanks to hardware AES cryptography. As shown by Table 6.1, it is possible to observe how both the enclave initialisation and the disk image loading phases are heavily affected by EPC paging when `HeapMaxSize` is set to `0x200000000` (8GB). In contrast, reducing the `HeapMaxSize` to `0x1D0000000` (7.25GB) is sufficient to remain below the maximum EPC reserved to the user, avoiding paging and therefore reducing the total execution time. The value of 7.25GB was chosen for efficiency, as during testing, it was found that 256MB is sufficient for all other portions of memory reserved. This suggests that, for the analysis phase, where the disk image is stored inside the EPC enclave, it is sufficient to allocate, as `HeapMaxSize`, the disk image size plus 256 MB.

Operation	*8 GB HeapMaxSize*	*7.25 GB HeapMaxSize*
Total time initialise enclave	49.602323 s	23.598042 s
Total time reading disk image	4.601444 s	3.596986 s
Total time forensic session	44.413768 s	8.286810 s
- Total time loading disk	38.761984 s	2.843013 s
- Total time disk analysis	3.985925 s	4.018020 s
- Total time unloading disk	0.617512 s	0.626612 s
Total time sealing session	0.113881 s	0.119663 s
Total time execution	**98.731461 s**	**35.601547 s**

Table 6.1. Secure Mode - Performance comparison on a 7GB disk image with different HeapMaxSize

Starting with the Secure Mode, the results in the figure 6.1 show a consistent behaviour across the three entry configurations, with negligible differences in execution time between the 500, 5,000, and 50,000 file cases. This demonstrates that, unlike the Standard Mode as will be explained later, the number of entries does not influence in a significant way the total execution time in Secure Mode. This is due to the difference in the architecture between the two modes: while the Standard Mode relies on the TSK library, which traverses the filesystem entry by entry and invokes a separate ecall for each data block of each file, the Secure Mode employs a custom internal FAT32 parser that operates entirely on the disk image already loaded into EPC memory. In this mode, the hashing process is executed directly inside the enclave through the `compute_sha256_chunked` function, which traverses the FAT cluster chain of each file and computes the SHA256 hash one cluster at a time, reading data directly from the trusted copy of the disk image without any ecall/ocall call during the hash computation itself. As a result, the total execution time in Secure Mode is not affected by the number of entries and remains constant across the three entry configurations.

The Standard Mode, as illustrated from the graph 6.2, shows a total execution time ranging from 73.79 to 78.86 seconds and a significantly lower export time compared to the Secure Mode (approximately 1.68–2.11 seconds versus approximately 23.87–23.96 seconds). Compared to the previous case, this scenario shows a little but observable difference in the analysis and export performance that tends to increase with the number of entries. In particular, the time required to complete the entire analysis phase increases by approximately 0.62 seconds when moving from 500 to 5,000 files (from 72.11 to 72.73 seconds), and by approximately 4 seconds when moving from 5,000 to 50,000 files (from 72.73 to 76.75 seconds). Even if the analysis time does not grow in a significant way, suggesting good scalability of the framework, which maintains acceptable performance with both small and large numbers of entries, it should be assumed that, in the case of disk images containing millions of files, this difference may become more evident and could contribute to slowing down the overall analysis process. As illustrated in the Table 6.2, which reports the times for the individual operations, the progressive increase is directly attributable to

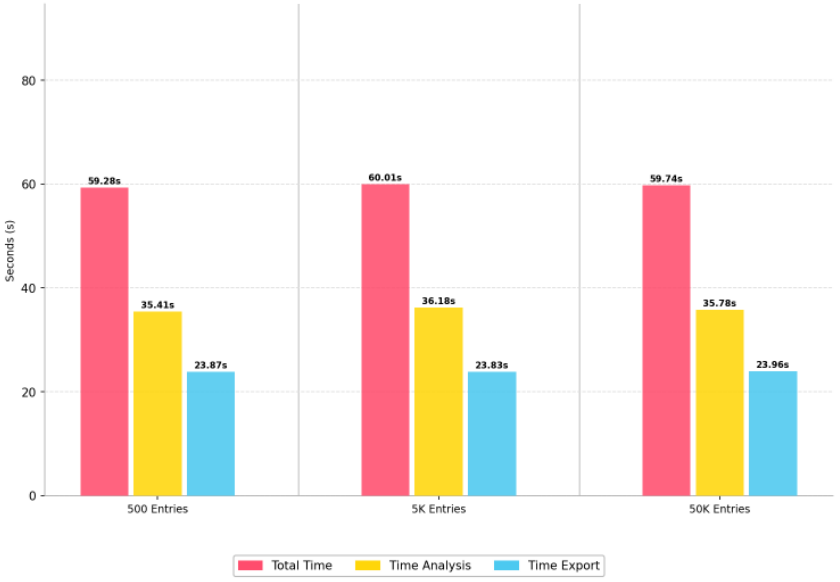

Figure 6.1. Secure Mode: Performance benchmark of analysis and export execution on datasets of increasing size.

the cumulative time of the **hash process**, intended as the entire process of hashing and which includes the three sequential operations performed for each file: SHA-256 context initialization via `ecall_init_hash()`, incremental digest updating via `ecall_update_hash()`, invoked for each data block read from the filesystem, and finalization via `ecall_finalize_hash()`. Each data block that is read from the filesystem triggers the ecall to communicate to the enclave, during which `sgx_sha256_update()` updates the cryptographic context in trusted memory. As the number of files increases, the total number of blocks to be processed increases proportionally, and consequently, also the number of `sgx_sha256_update()` invocations performed within the enclave. The comparison of the hash process is shown in the Table 6.2.

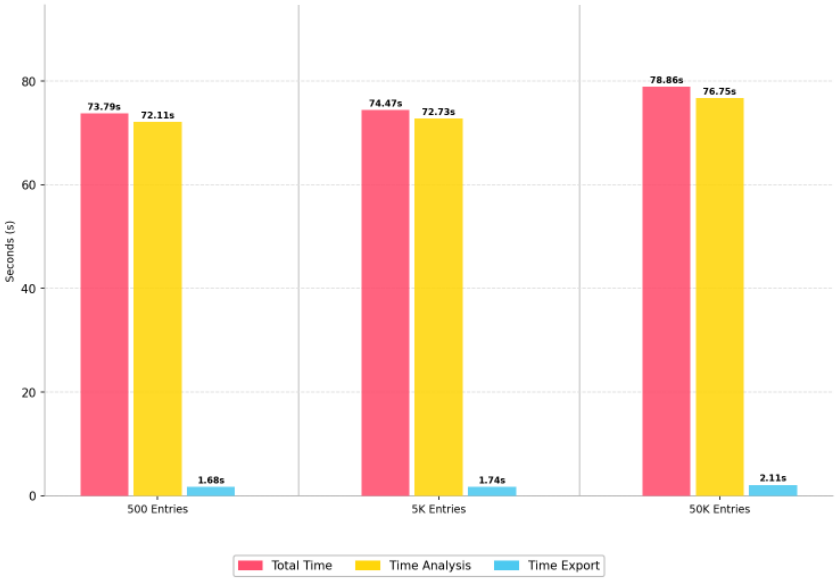

Figure 6.2. Standard Mode: Performance benchmark of analysis and export execution on datasets of increasing size.

Finally, with pure TSK, the results in the graph 6.3 show a clear advantage in terms of performance, maintaining a total execution time between 5.70 and 6.83 seconds regardless of

Operation	*500 Entries*	*5K Entries*	*50K Entries*
`ecall_init_forensic_session`	0.106759 s	0.106703 s	0.106904 s
`ecall_init_hash`	0.003463 s	0.033182 s	0.325627 s
`ecall_update_hash`	59.851294 s	60.857967 s	61.469544 s
`ecall_finalize_hash`	0.003381 s	0.032211 s	0.314139 s
`ecall_add_evidence_entry`	0.007701 s	0.067915 s	0.632347 s
`ecall_close_forensic_session`	0.000041 s	0.000046 s	0.000040 s
`ecall_get_sealed_data_size`	0.000030 s	0.000033 s	0.000028 s
`ecall_seal_forensic_session`	0.083321 s	0.109635 s	0.120500 s
`ocall_get_current_time`	0.001251 s	0.007288 s	0.047604 s
hash process	66.398890 s	67.862810 s	69.910712 s

Table 6.2. Standard Mode - Performance Comparison of 7GB Disk Images with different entries per disk

the number of entries, demonstrating excellent scalability across all tested configurations, as illustrated in Figure 6.3. Similarly to the Standard Mode, a slight increase in execution time can be observed as the number of files grows, although in this case, the impact remains negligible. That is because, also in this case, the TSK model is used. This result is primarily due to the fact that the analysis is conducted entirely outside the SGX framework, and therefore without any of the overhead introduced by Intel SGX. Consequently, there is no enclave initialisation cost, no ecall/ocall transitions, and no EPC memory constraints, allowing TSK to operate at native speed directly on the disk image.

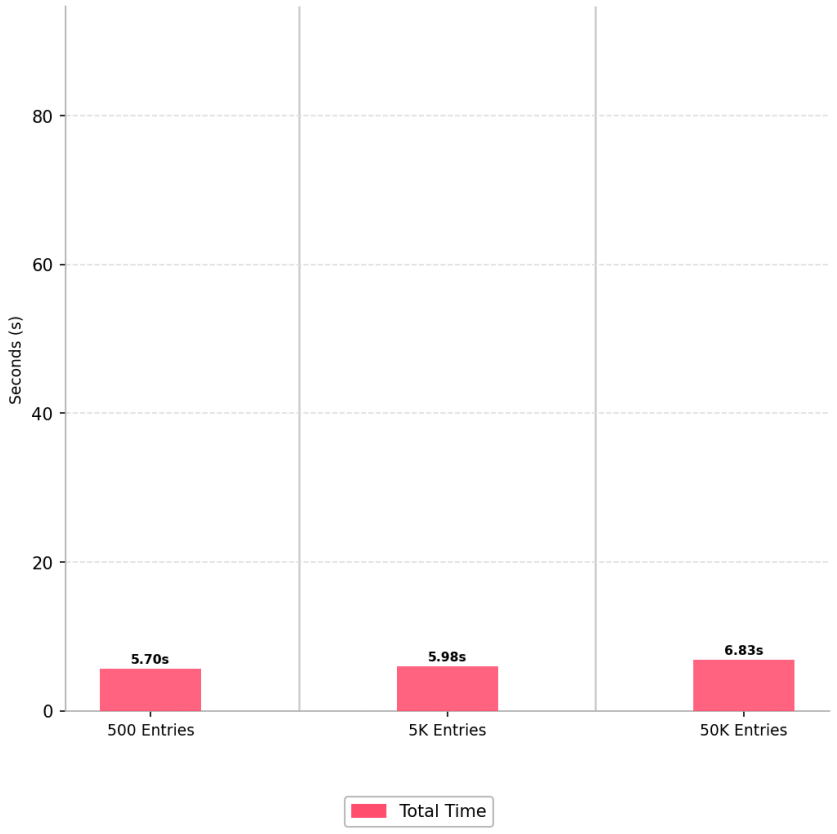

Figure 6.3. TSK Mode: Performance benchmark of TSK mode on datasets of different sizes.

6.7.2 Impact of Disk Image Size on Framework Performance

This testing section has the scope to compare the performance of the modes of the framework across three disk images with fixed entries (approximately 500 entries) of different sizes: 7, 3, and 1 GB, none of which is completely filled, representing a general scenario. The evaluation is executed for both Standard Mode and Secure Mode. For the Secure Mode, as in the previous testing section, the `HeapMaxSize` is set individually for each disk image analysis to the minimum value necessary to avoid waste of EPC memory, following the rule established earlier: the size of the disk image plus 256 MB. Therefore, the heap sizes used are approximately 7.25 GB, 3.25 GB, and 1.25 GB for the 7, 3, and 1 GB disk images, respectively. For the Standard Mode, since the disk image is not loaded into EPC memory but instead is accessed directly from the untrusted environment, the `HeapMaxSize` remains fixed at 256 MB regardless of the disk image size, as only the enclave metadata and working structures need to reside in trusted memory.

Secure Mode: As shown in Table 6.3, reducing the memory required for the analysis phase reduces the overall execution time. This is directly related to the heap size declared for the enclave: the larger the heap, the more EPC pages SGX must allocate, measure, and cryptographically verify before the enclave can begin execution. Specifically, SGX must execute an EADD and EEXTEND instruction for each 4KB page declared in the heap, each with a fixed cost per page. This behaviour is clearly confirmed by the experimental data: with a 7.25 GB heap, the initialisation time is approximately 23.60 seconds, resulting in a total execution time of 35.60 seconds; with a 3.25 GB heap, the initialisation drops to approximately 10.44 seconds, with a total of 15.41 seconds; and with a 1.25 GB heap, the initialisation is reduced to approximately 4.64 seconds, with a total execution time of only 6.04 seconds. It is also possible to note that, despite significant differences in disk image size, the actual analysis disk image time of the FAT32 parser does not vary dramatically across the three cases, ranging from 4.02 seconds for the 7 GB image to 0.25 seconds for the 1 GB image, confirming that the prevailing cost factor is the enclave initialisation rather than the disk forensic analysis itself. The same consideration applies to the export phase, as evidenced by Table 6.4, which also includes an enclave initialisation phase and is therefore subject to the same overhead depending on the declared heap size. In the table, it is evident that a large part of the time needed to complete the export is due to the enclave initialisation, which represents the 95-99% of the overall execution in each case. Furthermore, the cryptographic and security operations,such as data unsealing (0.166 seconds), hash chain verification (less than 0.001 seconds), CSV export, DCAP quote generation, and file writing operations, amount to only approximately 0.20 seconds, remaining constant across all disk image sizes. This confirms that the real bottleneck is not the complexity of the security operations, but rather SGX's management of EPC memory. Optimising the heap size, therefore, does not imply reducing security guarantees, but simply avoiding the performance degradation caused by unnecessary EPC allocation overhead.

	DiskImage 7GB	*DiskImage 3GB*	*DiskImage 1GB*
HeapMaxSize	7.25 GB	3.25 GB	1.25 GB
Total time initialize enclave	23.598042 s	10.442199 s	4.638940 s
Total time reading disk image	3.596986 s	1.445243 s	0.350011 s
Total time forensic session	8.286810 s	3.412133 s	0.941499 s
- Total time loading disk	2.843013 s	1.207617 s	0.406725 s
- Total time disk analysis	4.018020 s	1.499772 s	0.250029 s
- Total time unloading disk	0.626612 s	0.270302 s	0.089192 s
Total time sealing session	0.119663 s	0.115188 s	0.114348 s
Total time execution	**35.601547 s**	**15.414805 s**	**6.044844 s**

Table 6.3. Secure Mode - analyse Performance based on different HeapMaxSize

Standard Mode: Similar to the Secure mode case, the three disk images are evaluated, but in this case, the `HeapMaxSize` is fixed at 256 MB in all the configurations. As expected, the enclave initialisation has a very low impact on the total execution time, since the allocation and encryption of EPC pages have a minimal cost compared to the previous cases. This is confirmed by the Table 6.5, where the initialisation time remains stable and under 2 seconds. In this case,

	DiskImage 7GB	*DiskImage 3GB*	*DiskImage 1GB*
HeapMaxSize	7.25 GB	3.25 GB	1.25 GB
Total time initialize enclave	23.574338 s	10.616813 s	4.7197700 s
Total time setting DCAP	0.000002 s	0.000002 s	0.000002 s
Total time unsealing session	0.166645 s	0.164148 s	0.166745 s
Total time verifying hash chain	0.000197 s	0.000194 s	0.000193 s
Total time exporting csv	0.038569 s	0.040110 s	0.037175 s
- Total time export evidence	0.005141 s	0.005310 s	0.005146 s
- Total time DCAP quote gen	0.022544 s	0.022735 s	0.022447 s
- Total time writing .csv file	0.000244 s	0.000190 s	0.000183 s
- Total time writing .sig file	0.000098 s	0.000093 s	0.000065 s
Total time execution	**23.779801 s**	**10.771320 s**	**4.941862 s**

Table 6.4. Secure Mode - Export Performance based on different HeapMaxSize

the hash phase is the process that affects the final execution time. As shown in Table 6.6, the progressive increase in execution time across the 1, 3, and 7 GB disk images is directly attributable to the increasing number of ecalls during the hash process. That is because each time a file is processed, it is divided into blocks, and for each block, `ecall_update_hash` is called, and as the disk images grow in size, the total number of blocks to be processed increases, resulting in a higher number of enclave transitions.

	DiskImage 7GB	*DiskImage 3GB*	*DiskImage 1GB*
HeapMaxSize	256 MB	256 MB	256 MB
Total time initialize enclave	1.472349 s	1.465314 s	1.474830 s
Total time forensic session	70.497669 s	24.910032 s	4.366784 s
- Total time disk analysis	70.390869 s	24.802781 s	4.259978 s
- Total time hash process	66.398890 s	24.304644 s	3.995286 s
- Total time adding evidence	0.007701 s	0.007488 s	0.006672 s
Total time sealing session	0.140228 s	0.120629 s	0.110465 s
Total time execution	**72.11 s**	**26.49 s**	**5.98 s**

Table 6.5. Standard Mode - analyse Performance on different Disk images

Regarding the export performance, as in the case of Secure Mode, the factor that most affects the total execution time is the enclave initialisation, since all other operations, such as unsealing, hash chain verification, CSV export, and DCAP quote generation, are negligible in comparison, as reported in Table 6.7.

6.7.3 Performance Comparison: Standard Mode vs Secure Mode

By comparing the total execution times between Secure Mode and Standard Mode through the three disk images, it is possible to observe in the Figure 6.4, how the performance gap between the two modes increases as the disk image size grows, with Standard Mode being increasingly disadvantaged with respect to the other mode. As discussed in the previous testing section, this slowdown is primarily due by the large number of `ecall_update_hash` calls executed during the hashing phase: for each data block identified by TSK during the file walk, a separate ecall is issued to the enclave, and as the disk image grows in size, the total number of blocks, and consequently the number of ecall transitions, increases proportionally, introducing a cumulative overhead that becomes increasingly significant. In contrast, the Secure Mode does not exhibit a comparable degradation, since all hashing operations are performed entirely within the enclave using the custom FAT32 parser, which reads data directly from the trusted copy of the disk image already loaded into EPC memory. This eliminates the ecall/ocall overhead during hash computation, resulting in significantly lower transitions and a more contained growth in execution time as the disk image size increases.

	DiskImage 7GB	*DiskImage 3GB*	*DiskImage 1GB*
Total size of files (MB)	4216 MB	1569 MB	257 MB
`ecall_init_hash`	0.003463 s	0.003506 s	0.003316 s
`ecall_update_hash`	59.851294 s	22.223187 s	3.651507 s
`ecall_finalize_hash`	0.003381 s	0.003450 s	0.003205 s

Table 6.6. Standard Mode - Hash Process on different Disk images

	DiskImage 7GB	*DiskImage 3GB*	*DiskImage 1GB*
HeapMaxSize	256 MB	256 MB	256 MB
Total time initialize enclave	1.468479 s	1.473381 s	1.475224 s
Total time setting DCAP	0.000002 s	0.000002 s	0.000001 s
Total time unsealing session	0.162001 s	0.161016 s	0.161714 s
Total time verifying hash chain	0.000193 s	0.000194 s	0.000191 s
Total time exporting csv	0.038958 s	0.036828 s	0.068577 s
- Total time export evidence	0.005124 s	0.005301 s	0.005152 s
- Total time DCAP quote gen	0.022979 s	0.021893 s	0.022127 s
- Total time writing .csv file	0.000185 s	0.000190 s	0.000187 s
- Total time writing .sig file	0.000064 s	0.000090 s	0.000102 s
Total time execution	**1.679683s**	**1.671467 s**	**1.705756 s**

Table 6.7. Standard Mode - Export Performance on different Disk images

The issue related to the high number of ecalls becomes even more evident in Figure 6.5, which presents a comparison between fully populated disk images. In this scenario, the performance gap between Secure Mode and Standard Mode is considerably more evident. As observed in the previous testing phases, the only disk analysis time in Secure Mode does not vary significantly regardless of whether the disk image is full or not, since the prevalent cost factor of the total analysis phase is the enclave initialisation, which depends exclusively on the declared heap size and not on the actual content of the disk. In contrast, for Standard Mode, the amount of data present on the disk is a critical factor that directly impacts the overall analysis phase performance and, in particular, the disk analysis, since a fuller disk implies a larger number of blocks to be processed, and consequently a higher number of ecall transitions. This is further confirmed by Table 6.8, which reports the total number of ecalls performed by each mode. In the case of Secure Mode, the number of ecalls is relatively low, since the disk analysis occurs entirely within the enclave through the custom FAT32 parser, eliminating the need for repeated enclave transitions during data processing. In Standard Mode, however, the total number of ecalls reaches the order of millions, with the majority attributable only to `ecall_update_hash()`, while all other framework ecalls, such as `ecall_init_hash()`, `ecall_finalize_hash()`, and `ecall_add_evidence_entry()`,etc, collectively represent only a negligible fraction of the total. This is clearly illustrated in Table 6.9: in Standard Mode, the total number of ecalls reaches 14,043,232 for the 7 GB disk image, 5,904,479 for the 3 GB image, and 1,853,680 for the 1 GB image, with `ecall_update_hash()` alone with 14,041,797, 5,902,987, and 1,852,245 calls respectively, representing over 99.98% of all enclave transitions in every tested configuration.

For the latter part of performance testing, it can be concluded that the performance overhead observed in Standard Mode is primarily caused by the ecall transition latency rather than by the computational cost of the operations performed inside the enclave. This is clearly evident in Figure 6.6, which shows that when considering only the disk analysis phase and comparing TSK process in Standard Mode, in Intel SGX framework with TSK outside the Intel SGX framework (with this is intended the analysis of the disk including the hashing process), the difference in execution time is substantial, despite both performing the same underlying filesystem traversal and hashing operations.

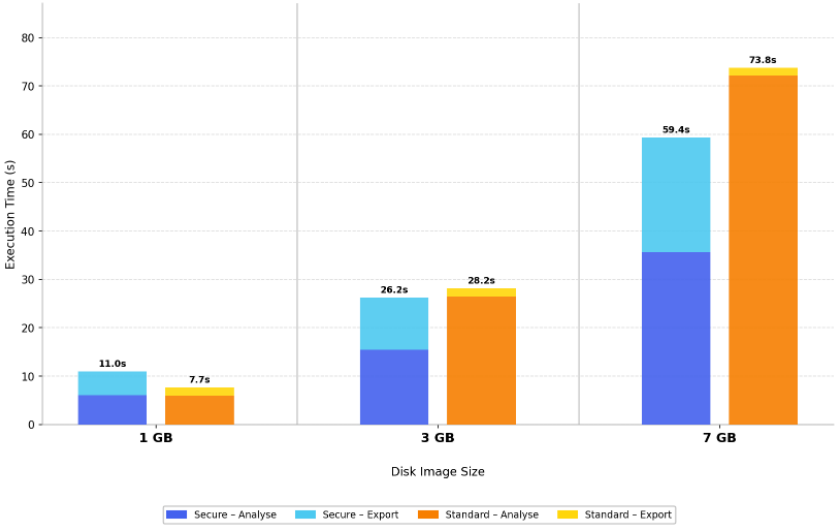

Figure 6.4. Total Execution Time Comparison with no full disk images: Standard Mode vs Secure Mode through Disk Image Sizes

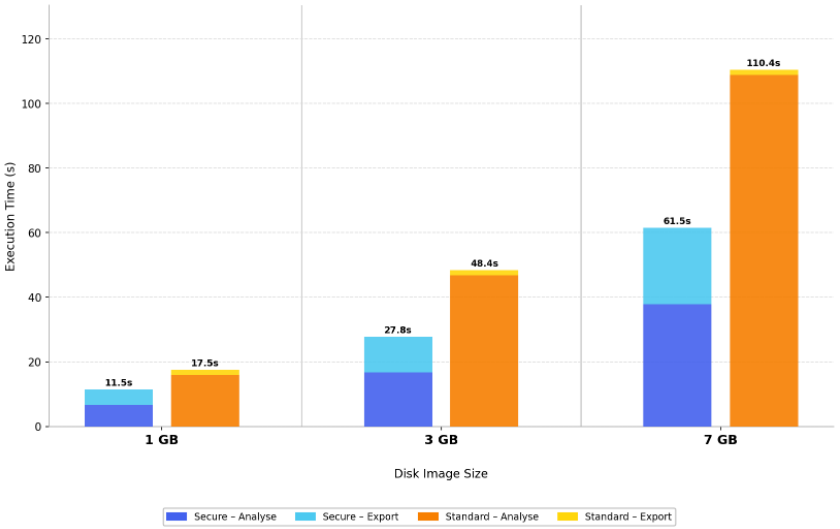

Figure 6.5. Total Execution Time Comparison with full disk images: Standard Mode vs Secure Mode through Disk Image Sizes

6.7.4 Performance Testing Final Considerations

As a conclusion, the important points that can be defined by the testing phase are:

1. The bottleneck of the Secure Mode is the enclave initialisation, and not the forensic analysis of the disk. The time grows proportionally to the HeapMaxSize declared, while the analysis of the disk image itself (with the custom parserFAT32) and the security operations remain contained.

2. The bottleneck of the Standard Mode is the ecall transitions during TSK analysis. `ecall_update_hash` represents over 99% of all the ecall transitions, reaching 14 million calls for a 7 GB image. This is demonstrated by comparing TSK inside and outside the enclave.

3. The Secure Mode is more scalable than the Standard Mode: when the size of the disk increases, the Secure Mode is not significantly affected because the ecall/ocall transitions during hashing are completely eliminated, and the difference between different disk image sizes is minimal in terms of performance.

Image Size	*Secure Mode*	*Standard Mode*
7.25 GB	235	14.043.232
3.25 GB	107	5.904.479
1.25 GB	43	1.853.680

Table 6.8. ECall count: Secure vs Standard Mode

Image Size	*Total ECall count*	`ecall_update_hash()` *count*
7.25 GB	14.043.232	14.041.797
3.25 GB	5.904.479	5.902.987
1.25 GB	1.853.680	1.852.245

Table 6.9. Standard Mode - ECall count

4. The number of files has a negligible impact on the Secure Mode, while it is slightly more evident in the Standard Mode, for the three cases tested.

5. All the security operations, such as sealing, export, DCAP and ECDSA signature, have an irrelevant cost with respect to the rest.

In terms of overall performance, the Secure Mode achieves better results and also provides greater security, but requires that the entire disk image fit in the EPC and supports only FAT32. On the other hand, the Standard Mode is more flexible, because it supports all the filesystems that TSK is able to analyse, and the disk is not loaded into enclave memory, but is worse in terms of performance.

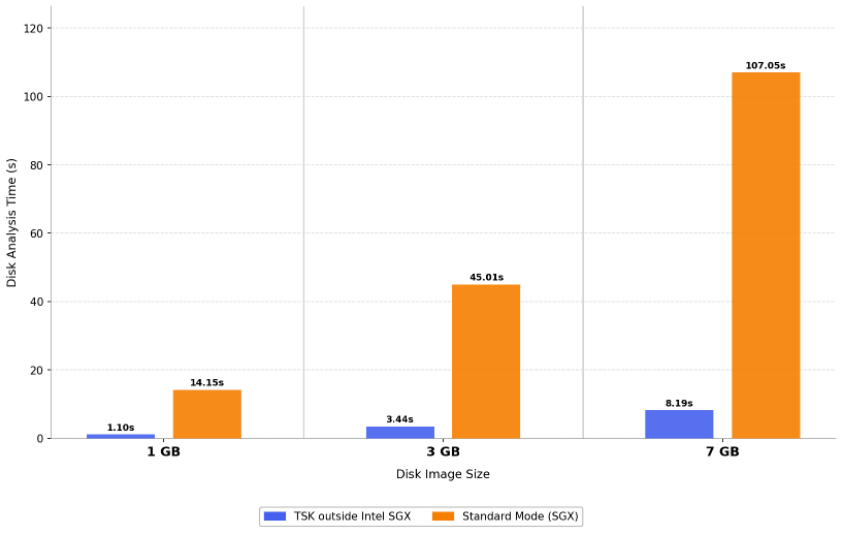

Figure 6.6. Disk Analysis Comparison: TSK in Standard Mode(Intel SGX) vs TSK (No Intel SGX)

Chapter 7

Conclusions and Future works

This work originated from a need to address the gap in digital forensics regarding the protection of evidence during analysis. As a solution, Forensic-SGX was proposed: a framework that combines the security features of Intel SGX with digital forensics, specifically by integrating The Sleuth Kit (TSK), a widely used forensic tool for analysing disk images across different filesystems, to ensure the preservation of evidence throughout an investigation. The framework offers two different modes of analysis: Standard Mode and Secure Mode. The Standard Mode represents the initial and core contribution of the framework, as it provides the actual integration of a forensic tool within an SGX application. The Secure Mode, on the other hand, is a complementary solution that replicates part of TSK's functionality through a custom in-enclave parser, currently supporting only the FAT32 filesystem. In addition, in order to guarantee the integrity and authenticity of the evidence, the framework incorporates sealing, ECDSA signature and DCAP remote attestation, making the forensic report verifiable by external parties. This is essential in an investigative context, where evidence must remain verifiable over time and by trusted entities, such as a court, to allow accurate decisions about a case.

In conclusion, it was demonstrated that the framework operates correctly and that the preservation of evidence is guaranteed from the moment it is generated inside the enclave to when it reaches a third-party verifier. This guarantee does not mean that the evidence cannot be compromised, since the process of securely distributing the evidence was not implemented; however, it is guaranteed that any compromise will be detected. Moreover, whenever a compromise occurs, the enclave's sealed session allows the evidence to be recovered by repeating the export process. Most of the identified vulnerabilities concern the analysis phase, where the disk image passes through the untrusted environment, and the reliance on untrusted time sources, which remains an open problem. From a performance perspective, the analysis time is considered acceptable. It was found that, in terms of both performance and security, the Secure Mode scales better than the Standard Mode. In particular, when testing with the largest disk image supported by the Secure Mode, the total execution time was approximately half that of the Standard Mode. Additionally, through a critical analysis of the framework design, it was concluded that the hash chain is largely redundant. It was originally implemented to guarantee the ordering of evidence entries, but this property is already covered by the sealing process, while the integrity of the exported report is independently guaranteed by the ECDSA signature. Finally, a further limitation was identified regarding the MRENCLAVE sealing policy: in a forensic context, sealed evidence could become permanently inaccessible after a framework update. This limitation is particularly relevant because, in a forensic investigation, evidence may need to be recovered at a later time, for instance, if the exported CSV or .sig file is lost.

Future Works

As future work, the most promising direction would be to implement the TSK filesystem parsing functionality entirely within the enclave. More specifically, the idea is to develop custom parsers, similar to the FAT32 parser already implemented in ForensicSGX, for all the filesystems that are

supported by TSK, such as NTFS, ext4, etc. This configuration would mean that the entire analysis will be performed in Secure Mode, reducing execution time by deleting the ecall transitions that represent the main bottleneck in Standard Mode performance, while also providing stronger security guarantees since all data analysed is within the trusted memory. However, this approach is not feasible on all devices, since the entire disk image must be loaded into EPC memory, and consequently, a substantial amount of EPC is required. That could be a problem because most Intel SGX processors currently support only 128 or 256 MB. Nevertheless, cloud platforms such as Microsoft Azure already offer confidential computing virtual machines with EPC memory in the terabyte range, making this solution increasingly feasible. As these resources become increasingly available worldwide, it is more realistic to imagine a framework version that can integrate parsers for all major filesystems and perform complete forensic analysis entirely within the enclave.

Another possible improvement concerns the acquisition of trusted time. In a forensic context, it is essential to guarantee that the timestamps associated with the analysis are reliable in order to prove exactly when the evidence was processed. A possible solution is to implement a communication channel with a Trusted Timestamping Authority (TSA), which would provide cryptographically signed timestamps.

Also, the sealing policy could be improved. Adopting the MRSIGNER policy, either as a default or as an alternative to MRENCLAVE, should prevent the loss of sealed sessions after enclave updates. In this scenario, the sealing key would be bound to the public key of the developer, rather than to the specific enclave binary. This could be an important solution, particularly in a forensic context, where investigations can last months or years, and software updates are inevitable.

On the verification side, the actual implementation relies on the QVL that operates in untrusted mode. Integrating the QvE (Quote Verification Enclave) modality would strengthen the verification process by enabling quote validation also within an Intel SGX machine, with the objective of executing a trusted verification. Both modalities should be implemented, since an external verifier may not own an SGX-integrated device.

Finally, another interesting improvement can be made on the mechanism for loading the disk image, which is executed in the untrusted component. A possible approach to detect alterations of the disk image could involve using the hash of the image computed during the acquisition phase and signed by an external trusted authority, similar to the TSA approach described for timestamps. The enclave could verify the signature of this hash using the public key of the authority, then compute the hash of the loaded image and compare it with the signed one. As a consequence, although the enclave cannot prevent the host application from modifying the data during loading, it would be able to detect any alteration.

If all the improvements described above were implemented, the framework could guarantee a trusted chain of custody from the moment the disk image is loaded into the enclave to the delivery of the signed evidence to the verifier. This does not mean that evidence cannot be compromised, but it guarantees that any compromise would be detected.

Appendix A

Installation of the Intel SGX Development Environment on Linux

A.1 Introduction

This appendix provides all the instructions for installing the Intel Software Guard Extensions (SGX) development environment on Linux. The installation was tested on Ubuntu 22.04.5 LTS, with Linux kernel 6.8.0-1044-azure. All experiments were conducted on a virtual machine deployed on Microsoft Azure.

Intel provides three primary software components for SGX development and deployment: the Intel SGX Software Development Kit (SDK), which facilitates the creation of SGX-enabled applications; the Intel SGX Platform Software (PSW) for Linux OS, which supplies the runtime modules necessary for executing SGX applications; and the Intel SGX Data Center Attestation Primitives (DCAP) for Linux, which enables remote attestation within datacenter environments.

Before proceeding with installation, verify whether the processor supports Intel SGX:

```
$ cpuid | grep -i sgx
     SGX: Software Guard Extensions supported = True
     SGX_LC: SGX launch config supported = True
```

If both flags return `true`, the processor supports Intel SGX with Flexible Launch Control (FLC), which is required for Intel SGX DCAP functionality.

Note: The Intel SGX SDK allows experiments to be performed either in simulation mode or in hardware mode. Simulation mode does not require SGX support on the host processor. However, some specific tests conducted in this work require execution in hardware mode, which in turn requires SGX-enabled hardware. Since the local processor did not support SGX, a virtual machine on Microsoft Azure was used. For further details on how to create and configure an Azure virtual machine, refer to Section Installation and Configuration of an Azure VM

A.2 Installation Instructions

To build or develop an application that uses an Intel SGX enclave, all components described in the section Intel SGX Application User must be installed. Additionally, the Intel SGX Software Development Kit (SDK) and developer packages are required, as described in this section Intel SGX Application Developer.

A.2.1 Driver Installation

Starting from Linux kernel version 5.11, the Intel SGX driver has been integrated into the mainline kernel. Therefore, no additional driver installation is required on Linux distributions running a kernel version equal to or newer than 5.11. Once properly loaded, the corresponding device files are exposed under `/dev/sgx_enclave` and `/dev/sgx_provision`.

Their presence can be verified using the following command:

```
$ ls -l /dev/sgx*
```

If the driver is correctly initialised, the above command should list both device nodes.

```
crw-rw-rw- 1 root sgx 10, 125 Feb 11 17:31 /dev/sgx_enclave
crw-rw---- 1 root sgx_prv 10, 126 Feb 11 17:31 /dev/sgx_provision

/dev/sgx:
total 0
lrwxrwxrwx 1 root root 14 Feb 11 17:31 enclave -> ../sgx_enclave
lrwxrwxrwx 1 root root 16 Feb 11 17:31 provision -> ../sgx_provision
```

It is important to ensure that the underlying platform supports Flexible Launch Control (FLC) and that this feature is correctly enabled in the system configuration.

A.2.2 Intel SGX Application User

To execute SGX applications built with the Intel SGX SDK, the Platform Software (PSW) runtime components must be installed.

To configure the Intel® SGX runtime environment, the main packages `libsgx-quote-ex` and `libsgx-dcap-ql` must be installed.

Installing these primary packages automatically pulls in several dependent components required for proper SGX operation and attestation support. The following packages are installed as dependencies:

- `libsgx-ae-le`
- `libsgx-ae-pce`
- `libsgx-ae-qe3`
- `libsgx-ae-qve`
- `libsgx-aesm-ecdsa-plugin`
- `libsgx-aesm-quote-ex-plugin`
- `libsgx-dcap_quote-verify`
- `libsgx-enclave-common`
- `libsgx-pce-logic`
- `libsgx-qe3-logic`
- `libsgx-urts`
- `sgx-aesm-service`

These components provide architectural enclave support, quoting and verification mechanisms, runtime libraries, and background services required for enclave execution and remote attestation.

To do that, we need to follow these steps:

1. Ensure that we have a connection to the internet and open a terminal
2. Setup the necessary package repository:

```
$ sudo tee /etc/apt/sources.list.d/intel-sgx.list > /dev/null <<EOF
deb [signed-by=/etc/apt/keyrings/intel-sgx-keyring.asc arch=amd64]
    https://download.01.org/intel-sgx/sgx_repo/ubuntu jammy main
EOF
```

3. Download the public key of the package repository and add it to the list of trusted keys that are used by apt to authenticate packages:

```
$ curl -fsSLO
    https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-deb.key
$ sudo mv intel-sgx-deb.key /etc/apt/keyrings/intel-sgx-keyring.asc
```

4. Update apt and install the following packages:

```
$ sudo apt-get update
$ sudo apt-get install libsgx-quote-ex libsgx-dcap-ql
```

5. **(Optional)** To debug with sgx-gdb, install the debug symbol packages:

```
$ sudo apt-get install \
    libsgx-aesm-ecdsa-plugin-dbgsym \
    libsgx-aesm-launch-plugin-dbgsym \
    libsgx-aesm-pce-plugin-dbgsym \
    libsgx-aesm-quote-ex-plugin-dbgsym \
    libsgx-dcap-default-qpl-dbgsym \
    libsgx-dcap-ql-dbgsym \
    libsgx-dcap-quote-verify-dbgsym \
    libsgx-enclave-common-dbgsym \
    libsgx-launch-dbgsym \
    libsgx-pce-logic-dbgsym \
    libsgx-qe3-logic-dbgsym \
    libsgx-quote-ex-dbgsym \
    libsgx-ra-network-dbgsym \
    libsgx-ra-uefi-dbgsym \
    libsgx-tdx-logic-dbgsym \
    libsgx-uae-service-dbgsym \
    libsgx-urts-dbgsym \
    libtdx-attest-dbgsym \
    sgx-aesm-service-dbgsym \
    sgx-pck-id-retrieval-tool-dbgsym \
    sgx-ra-service-dbgsym \
    tdx-qgs-dbgsym \
    tee-appraisal-tool-dbgsym
```

6. **(Optional)** If you intend to run an application that uses an Intel SGX enclave requiring the Provision Key Access, your user needs to be added to the group `sgx_prv`. Note that any enclave obtaining an SGX Quote using the DCAP Quote Generation Library requires this access. A user <username> can be added to the group with the following command:

```
$ sudo usermod -aG sgx_prv <username>
```

A.2.3 Install Intel SGX/TDX DCAP

The Intel SGX/TDX Data Center Attestation Primitives (DCAP), the Provisioning Certificate Caching Service (PCCS), and the Quote Provider Library (QPL) must be configured before using remote attestation. The PCCS and QPL work together to cache DCAP attestation collateral and make it available to the DCAP Quote Generation Library (`libsgx-dcap-ql`).

1. [**Optional**] Set up the Provisioning Certificate Caching Service (PCCS). Note that Intel TDX and Intel SGX share the same PCCS instance.

   Note: If an external infrastructure provider (e.g., a Cloud Service Provider) is used, verify whether a specific collateral caching service and QPL are required. For example, Microsoft Azure provides the Trusted Hardware Identity Management (THIM) service as an alternative to the Intel-provided PCCS. As in this specific case the VM used for the framework is hosted on Azure, the following steps must to be followed Configuration of Intel QPL with Azure THIM.

2. Install the DCAP QPL package:

   ```
   sudo apt-get install libsgx-dcap-default-qpl
   ```

Intel DCAP Quote Generation and Quote Verification:

To compile SGX applications that use DCAP quote generation and verification, the follwing packages must also be installed, since they provide the necessary header files and linking libraries:

```
$ sudo apt-get install libsgx-dcap-ql-dev libsgx-dcap-quote-verify-dev \
    libsgx-dcap-default-qpl-dev libsgx-headers
```

A.2.4 Intel SGX Application Developer

In addition to installing the Intel Software Guard Extensions Platform Software (Intel SGX PSW), we should also install the Intel SGX Software Development Kit (Intel SGX SDK) and the prerequisite software.

1. Install the prerequisite software. For to install the required tools to build the Intel(R) SGX SDK, the following command must be used:

   ```
   $ sudo apt-get install build-essential python-is-python3
   ```

2. Download the Intel SGX SDK and install it.

   (a) Insert the appropriate Linux distribution and its version, the Intel SGX SDK version, and build and run the command:

   ```
   $ wget - https://download.01.org/intel-sgx/sgx-linux/2.25/
   distro/ubuntu20.04-server/sgx_linux_x64_sdk_2.25.100.3.bin
   ```

   (b) Adjust the file permissions:

   ```
   $ chmod +x sgx_linux_x64_sdk_2.25.100.3.bin
   ```

   (c) Start interactive setup by running the following command:

   ```
   $ sudo ./sgx_linux_x64_sdk_2.25.bin
   ```

   (d) When the question **Do you want to install in the current directory? [yes/no]** appears, choose one of the following:

   - If you want to install the components in the current directory, type yes and press Enter.
   - If you want to provide another path for the installation, type no and press Enter.

   Note: The Intel SGX SDK is installed now in the directory chosen by the user during the installation process. In this location, we can also find a generated script ***uninstall.sh****, which can use to uninstall the Intel SGX SDK*

   (e) Run the following command to set all environment variables:

   ```
   $ source <User Input Path>/sgxsdk/environment
   ```

3. Install the appropriate developer package:

   ```
   $ sudo apt-get install libsgx-enclave-common-dev
   ```

Appendix B

The Sleuth Kit Installation and Configuration

B.1 System Requirments

TSK has been tested and verified on multiple operating systems:. The platforms on which TSK has been tested and can be developed are:

- FreeBSD 2.x - 6.x
- Linux 2.x and later
- OpenBSD 2.x - 3.x
- Mac OS X
- SunOS 4.x - 5.x
- Windows

To verify the operating system version on Ubuntu:

```
# Check Ubuntu version
$ lsb_release -a
```

Build System Requirments To compile TSK from a source distribution, the following tools are required:

- C/C++ compiler (C++14 standard required)

```
# Verify GCC version and C++14 support
$ g++ --version

#Install make
$ sudo apt-get install make
```

- GNU Make

```
# Verify installation
$ make --version

# Install GNU Make
$ sudo apt-get install make
```

Note: The `build-essential` package includes all previously mentioned build tools (C/C++ compiler and GNU Make).

```
$ sudo apt-get install build-essential
```

Development System Requirements To extend TSK functionality or compile from the source repository, additional tools are required:

- GNU autoconf
- GNU automake
- GNU libtool
- All build system requirements (C/C++ compiler, GNU Make)

Install on Ubuntu:

```
$ sudo apt-get install autoconf automake libtool
```

B.2 Installation

Note: For Windows installation instructions, refer to the official documentation at https://github.com/sleuthkit/sleuthkit/blob/develop/README_win32.txt.

TSK employs GNU autotools for its build and installation process, which encompasses several sequential steps:

1. Install the dependencies:

   ```
   $ sudo apt update
   $ sudo apt install build-essential
   ```

2. Download the last version of TSK here https://www.sleuthkit.org/sleuthkit/download.php. The version used for this work is TSK 4.14.0 (`sleuthkit-4.14.0.tar.gz`).

3. Excract:

   ```
   $ tar -xzf sleuthkit-4.14.0_tar.gz
   $ cd sleuthkit-4.14.0
   ```

4. Run the 'configure' script in the root TSK directory. See the CONFIGURE OPTIONS section for useful arguments that can be given to 'configure' :

   ```
   $ ./configure
   ```

5. If there were no errors, then run 'make':

   ```
   $ make
   ```

6. The 'make' process will take a while and will build the TSK tools. When this process is complete, the libraries and executables will be located in the TSK sub-directories. To install them, type 'make install':

   ```
   $ make install
   ```

   By default, TSK installs to the `/usr/local/` directory hierarchy, with executables placed in `/usr/local/bin`.

7. After installation, update the system's shared library cache to enable the linker to locate TSK libraries:

   ```
   $ sudo ldconfig
   ```

Appendix C

Azure VM Installation and Configuration

In the following section, all the steps that were followed to create and configure the Azure VM are reported.

C.1 Installation and Configuration of an Azure VM

From the home page at https://portal.azure.com/#home, the first step to create a VM is to click on `Virtual Machines` and select `Create`. In the `Basics` section, the important configurations are the following:

- **Region:** In this case, `Italy North` was selected, but this is a preference depending on the user's location. Note that not all VM sizes are available in all regions; therefore, the nearest region that supports the desired VM size should be selected.
- **Availability options:** No infrastructure redundancy required.
- **Security type:** Trusted launch virtual machines. Note: for the specific framework developed in this work, the Confidential virtual machine option is also sufficient.
- **Image:** Ubuntu Server 22.04 LTS - x64 Gen2.
- **VM architecture:** x64.
- **Size:** `Standard_DC2s_v3` - 2 vCPUs, 16 GiB memory. In this section, a VM size that supports the intended workload should be selected, as the chosen size determines factors such as processing power, memory, and storage capacity. In order to have Intel SGX support, a VM from the DCsv3 series must be selected.

In the `Disks` section, a disk size of 64 GiB was selected.

In the `Networking` section, the `Public IP` option was enabled, and SSH (port 22) and RDP (port 3389) were selected as allowed inbound ports.

In all the remaining sections, the default options were kept.

C.2 Remote Desktop Control with RDP

Since a server version of Ubuntu is installed, the only default access method is via Secure Shell (SSH). To enable a graphical desktop environment, this section describes how to configure Remote Desktop on the Azure VM using the RDP protocol.

Note: Using Remote Desktop over the internet will introduce noticeable lag (input latency) compared to local desktop use. This latency does not reflect the actual performance of the VM itself.

Most Linux VMs in Azure do not have a desktop environment installed by default, as they are commonly managed through SSH connections. However, several desktop environments can be installed. The following steps describe how to install the `xfce4` desktop environment and configure RDP access on an Ubuntu 22.04 LTS VM.

1. **Connect to the VM via SSH:**

   ```
   $ ssh azureuser@myvm.westus.cloudapp.azure.com
   ```

2. **Install the required packages:**

   ```
   $ sudo apt update
   $ sudo apt install xfce4 xfce4-session xrdp
   $ sudo systemctl enable xrdp
   ```

3. **Grant certificate access to the xrdp user:**

   ```
   $ sudo adduser xrdp ssl-cert
   ```

4. **Set xfce4 as the default desktop environment for RDP sessions:**

   ```
   $ echo xfce4-session > ~/.xsession
   ```

5. **Restart the xrdp service for the changes to take effect:**

   ```
   $ sudo systemctl restart xrdp
   ```

6. **(Optional) Set a password for the local user account**, if this was not done during the VM creation:

   ```
   $ sudo passwd azureuser
   ```

7. **Create a Network Security Group (NSG) rule for RDP traffic.** To allow Remote Desktop traffic to reach the VM, port 3389 must be opened in the Networking section of the Azure portal, if not previously configured.

8. **Connect to the VM using a Remote Desktop client.** In this case, `Remmina` was used as the RDP client.

   Note: To install `Remmina` on the local machine (Ubuntu), the following commands can be used:

   ```
   $ sudo apt-add-repository ppa:remmina-ppa-team/remmina-next
   $ sudo apt update
   $ sudo apt install remmina remmina-plugin-rdp remmina-plugin-secret
   ```

C.3 Configuration of Intel QPL with Azure THIM

In order to use Intel QPL with the Azure Trusted Hardware Identity Management (THIM) service, the Intel QPL configuration file `sgx_default_qcnl.conf` must be modified. This file defines the endpoints used to retrieve the attestation collateral required for quote generation and verification.

The configuration file is located at:

`/etc/sgx_default_qcnl.conf`

The file must be edited to point to the Azure THIM endpoint instead of the default Intel PCS. The following is the configuration used in this work:

```
{
    "pccs_url": "https://global.acccache.azure.net/sgx/certification/v4/",
    "use_secure_cert": true,
    "pccs_api_version": "3.1",
    "retry_times": 6,
    "retry_delay": 10,
    "local_pck_url":
        "http://169.254.169.254/metadata/THIM/sgx/certification/v4/",
    "pck_cache_expire_hours": 168,
    "verify_collateral_cache_expire_hours": 168,
    "local_cache_only": false,
    "custom_request_options": {
        "get_cert": {
            "headers": {
                "metadata": "true"
            },
            "params": {
                "api-version": "2023-06-01-preview"
            }
        }
    }
}
```

After modifying the configuration file, the AESM service must be restarted for the changes to take effect:

```
$ sudo systemctl restart aesmd
```

Appendix D

Forensic-SGX Installation and Usage Guide

This guide provides all the steps required to test and configure Forensic-SGX. Note that to execute the described procedures, the hardware requirements must be met. Refer to Appendix A and Appendix B to install the necessary libraries to work with the framework. If no compatible device is available, refer to Appendix C for instructions on configuring a VM on Azure.

After installing the libraries, the first step is to download the framework from GitHub with the following command:

```
$ git clone https://github.com/Andrea9991/Forensic-SGX.git
```

Then, the following steps must be followed inside the `Forensic-SGX` folder:

```
$ source <User Input Path>/sgxsdk/environment
$ make SGX_MODE=HW
```

After that, it is possible to choose which mode to execute.
Note: for the Secure Mode, only the FAT32 filesystem can be analysed.

Standard Mode:

```
$ ./app analyse <disk_image> <AnalystId> <CaseId>
```

Secure Mode:

```
$ ./app analyse-secure <disk_image> <AnalystId> <CaseId>
```

The result of these modes is a sealed file. Once generated, the following command can be used to export the evidence:

```
$ ./app export <sealed_file.bin> [output.csv]
```

This command returns the `.csv` file and the `.sig` file. To verify the DCAP quote and Public Key binding:

```
$ ./app verify-quote <signature_file.sig>
```

This command verifies the DCAP quote and the public key binding. The verification of the ECDSA signature on the CSV file is not included in this command, but was manually verified during the testing phase using OpenSSL.

To clean the build framework:

```
$ make clean
```

Note: every time the source code is modified, a recompilation is required. To do so, run `make clean` followed by `make` with the desired mode (e.g., `make SGX_MODE=HW`).

Bibliography

[1] B. Guttman, D.R. White, "Digital Evidence Preservation: Considerations for Evidence Handlers", NIST Interagency Report 8387, National Institute of Standards and Technology, September 2022

[2] C. Shepherd and K. Markantonakis, "Trusted Execution Environments", Springer, 2024, ISBN: 978-3-031-55561-9

[3] Arm and Pulse, "Confidential Computing: A Pulse Survey on the Future of Security Technology", tech. rep., Arm, 2021. Data collected from December 2020 to February 2021. https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/confidential-computing-pulse-survey.pdf

[4] J.B. Hong and D.S. Kim, "Towards scalable security analysis using multi-layered security models", Journal of Network and Computer Applications, vol. 75, November 2016, pp. 156–168, DOI 10.1016/j.jnca.2016.08.024

[5] H.X. Do and V.H. Ha and V. Tran and É. Renault, "The Technique of Locking Memory on Linux Operating System: Application in Checkpointing", 2019 6th NAFOSTED Conference on Information and Computer Science (NICS), December 2019, DOI 10.1109/NICS48868.2019.9023816

[6] CVE Details, "Linux Kernel Security Vulnerabilities", https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2026

[7] StackScale, "The Linux Kernel surpasses 40 Million lines of code: A historic milestone in Open-Source software", https://www.stackscale.com/blog/linux-kernel-surpasses-40-million-lines-code/, January 2025

[8] N. Abu-Ghazaleh and D. Ponomarev and D. Evtyushkin, "How the Spectre and Meltdown Hacks Really Worked", IEEE Spectrum, vol. 56, March 2019, pp. 42–49, DOI 10.1109/MSPEC.2019.8651934

[9] TechTarget, "Data in use", https://www.techtarget.com/whatis/definition/data-in-use, 2024

[10] Confidential Computing Consortium, "A technical analysis of confidential computing", tech. rep., Confidential Computing Consortium, 2022. https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf

[11] D. Feng, Y. Qin, W. Feng, W. Li, K. Shang, H. Ma, "Survey of research on confidential computing", IET Communications, vol. 18, April 2024, pp. 535–556, DOI 10.1049/cmu2.12759

[12] M. Russinovich and M. Costa and C. Fournet and D. Chisnall and A. Delignat-Lavaud and S. Clebsch and K. Vaswani and V. Bhatia, "Toward Confidential Cloud Computing", Communications of the ACM, vol. 64, June 2021, pp. 50–61, DOI 10.1145/3453930

[13] S. Anasuri, "Confidential Computing Using Trusted Execution Environments", International Journal of AI, BigData, Computational and Management Studies, vol. 4, no. 2, 2023, pp. 97–110, DOI 10.63282/3050-9416.IJAIBDCMS-V4I2P111

[14] Microsoft, "Trusted Computing Base in Azure Confidential Computing", https://learn.microsoft.com/en-us/azure/confidential-computing/trusted-compute-base, 2025, Last updated May 7, 2025

[15] "TEE Initial Configuration v1.1", Tech. Rep. GPD_GUI_069, GlobalPlatform, December 2016. https://globalplatform.org/specs-library/tee-initial-configuration/

[16] "TEE Protection Profile v1.3", Tech. Rep. GPD_SPE_021, GlobalPlatform, September 2020. https://globalplatform.org/specs-library/tee-protection-profile/

[17] B. Ngabonziza and D. Martin and A. Bailey and H. Cho and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", 2016 IEEE 2nd International Conference on Collaboration and Internet Computing, Pittsburgh, PA, USA, November 2016, DOI 10.1109/CIC.2016.63

[18] J. Schütz, "General overview of Intel SGX", Friedrich-Alexander-Universität Erlangen-Nürnberg, Course Material, 2023, https://sys.cs.fau.de/extern/lehre/ws22/akss/material/intel-sgx.pdf

[19] V. Costan and S. Devadas, "Intel SGX Explained", Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.pdf

[20] Intel Corporation, "Intel SGX Developer Guide", May 2025. https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Guide.pdf

[21] J. Ménétrey and C. Göttel and A. Khurshid and M. Pasin and P. Felber and V. Schiavoni and S. Raza, "Attestation Mechanisms for Trusted Execution Environments Demystified", 22nd International Conference on Distributed Applications and Interoperable Systems (DAIS), Lucca, Italy, 2022, DOI 10.1007/978-3-031-16092-9_7

[22] A. Nilsson and P. Nikbakht Bideh and J. Brorsson, "A Survey of Published Attacks on Intel SGX", arXiv preprint arXiv:2006.13598, June 2020

[23] Systems Software & Security Lab, "SGX 101", https://sgx101.gitbook.io/sgx101/

[24] Intel Corporation, "Intel SGX Developer Reference Linux 2.26 Open Source", May 2025. https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Reference_Linux_2.26_Open_Source.pdf

[25] A.A. Khan and A.A. Shaikh and A.A. Laghari and M.A. Dootio and M.M. Rind and S.A. Awan, "Digital forensics and cyber forensics investigation: security challenges, limitations, open issues, and future direction", International Journal of Electronic Security and Digital Forensics, vol. 14, no. 2, 2022, pp. 124–150, DOI 10.1504/IJESDF.2022.122139

[26] J.P. Daly, "The Computer Fraud and Abuse Act: A New Perspective - Let the Punishment Fit the Damage", The John Marshall Journal of Computer & Information Law, vol. 12, no. 3, 1993, pp. 445–478

[27] E. Casey, "Computer Crime and Digital Evidence", Encyclopedia of Forensic Sciences, pp. 429–435, Oxford, UK: Elsevier Ltd., 2005

[28] D. Ayers, "A second generation computer forensic analysis system", Digital Investigation, vol. 6, 2009, pp. S34–S42, DOI https://doi.org/10.1016/j.diin.2009.06.013. The Proceedings of the Ninth Annual DFRWS Conference

[29] Scientific Working Group on Digital Evidence, "Data Integrity Within Computer Forensics", tech. rep., Scientific Working Group on Digital Evidence, April 2006. Version 1.0. https://www.swgde.org/wp-content/uploads/2023/11/2006-04-12-SWGDE-Data-Integrity-Within-Computer-Forensics-V1-0.pdf

[30] A. Ghosh, "Guidelines for the Management of IT Evidence", http://unpan1.un.org/intradoc/groups/public/documents/APCITY/UNPAN016411.pdf, 2004, Accessed: August 18, 2008

[31] Y. Prayudi and A. SN, "Digital Chain of Custody: State of the Art", International Journal of Computer Applications, vol. 114, March 2015, pp. 1–9

[32] J. Cosic and Z. Cosic, "Chain of Custody and Life Cycle of Digital Evidence", Computer Technology and Application, vol. 3, February 2012, pp. 126–129. Published: February 25, 2012

[33] P.S. Vinayagam, "Digital Forensic Tools for Cybercrime Investigation: A Comparative Analysis", International Journal of Applied Information Systems, vol. 12, April 2025, pp. 25–

[34] G. Horsman, "An "Order of Data Acquisition" for Digital Forensic Investigations", Journal of Forensic Sciences, vol. 67, January 2022, pp. 1215–1220, DOI 10.1111/1556-4029.14979

[35] M.M. Pollitt, "Triage: A Practical Solution or Admission of Failure", Digital Investigation, vol. 10, no. 2, 2013, pp. 87–88, DOI 10.1016/j.diin.2013.04.004

[36] ERMProtect, "The Guide to Digital Forensics", tech. rep., ERMProtect, 2020. https://ermprotect.com/wp-content/uploads/2020/06/The-Guide-to-Digital-Forensics.pdf

[37] K. Kent and S. Chevalier and T. Grance and H. Dang, "Guide to Integrating Forensic Techniques into Incident Response", NIST Special Publication 800-86, National Institute of Standards and Technology, August 2006. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-86.pdf

[38] R. Montasari and R. Hili and V. Carpenter and A. Hosseinian-Far, "The Standardised Digital Forensic Investigation Process Model (SDFIPM)", Digital Forensic Investigation of Internet of Things (IoT) Devices, 2019. University of Northampton Repository.https://nectar.northampton.ac.uk/id/eprint/11862/

[39] J. Kävrestad and M. Birath and N. Clarke, "Fundamentals of Digital Forensics: A Guide to Theory, Research and Applications", Texts in Computer Science, Springer Nature Switzerland AG, 3rd ed., 2024, ISBN: 978-3-031-53648-9. Series ISSN: 1868-0941 (print), 1868-095X (electronic)

[40] M.D. Kohn and M.M. Eloff and J.H.P. Eloff, "Integrated Digital Forensic Process Model", Computers & Security, vol. 38, October 2013, pp. 103–115, DOI 10.1016/j.cose.2013.05.001

[41] A. Faizal and A. Luthfi, "Comparison Study of NIST SP 800-86 and ISO/IEC 27037 Standards as A Framework for Digital Forensic Evidence Analysis", Journal of Information Systems and Informatics, vol. 6, June 2024

[42] A. Ajijola and P. Zavarsky and R. Ruhl, "A Review and Comparative Evaluation of Forensics Guidelines of NIST SP 800-101 Rev.1:2014 and ISO/IEC 27037:2012", 2014 World Congress on Internet Security (WorldCIS), London, UK, December 2014, pp. 66–73, DOI 10.1109/WorldCIS.2014.7028169

[43] "Information technology – Security techniques – Guidelines for identification, collection, acquisition, and preservation of digital evidence", International Standard ISO/IEC 27037:2012, International Organization for Standardization, October 2012. https://www.iso.org/standard/44381.html

[44] K. Ghazinour and D.M. Vakharia and K.C. Kannaji and R. Satyakumar, "A Study on Digital Forensic Tools", 2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI), Chennai, India, September 2017, pp. 2090–2095, DOI 10.1109/ICPCSI.2017.8392304

[45] T. Wu and F. Breitinger and S. O'Shaughnessy, "Digital forensic tools: Recent advances and enhancing the status quo", Forensic Science International: Digital Investigation, vol. 34, 2020, p. 300999, DOI 10.1016/j.fsidi.2020.300999

[46] K.M.M. Salih and N.B. Ibrahim, "Digital Forensic Tools: A Literature Review", Journal of Education and Science, vol. 32, March 2023, pp. 109–124, DOI 10.33899/edusj.2023.137420.1304

[47] R.K.M. Galvão, "Computer Forensics with The Sleuth Kit and The Autopsy Forensic Browser", The International Journal of Forensic Computer Science (IJOFCS), vol. 1, 2006, pp. 41–44. https://web.archive.org/web/20181222113925id_/http://ijofcs.org:80/V01N1-P05%20-%20Computer%20Forensics%20with%20the%20Sleuth%20Kit.pdf

[48] M. Pollitt, "A History of Digital Forensics", Advances in Digital Forensics VI, Berlin, Heidelberg, 2010, pp. 3–15, DOI 10.1007/978-3-642-15506-2_1

[49] B. Carrier, "The Sleuth Kit: Open Source Digital Forensics", https://www.sleuthkit.org/, 2023

[50] B. Carrier, "The Sleuth Kit (TSK) Library User's Guide and API Reference". The Sleuth Kit, 2020. Version 4.13.0. https://sleuthkit.org/sleuthkit/docs/api-docs/4.13.0/index.html

[51] ELM ChaN, "FAT Filesystem", https://elm-chan.org/docs/fat_e.html, Accessed: 2026

[52] M. Busch and F. Nicolai and F. Fleischer and C. Rückert and C. Safferling and F. Freiling, "Make Remote Forensic Investigations Forensic Again: Increasing the Evidential Value of Remote Forensic Investigations", Digital Forensics and Cyber Crime. ICDF2C 2020, Cham, February 2021, DOI 10.1007/978-3-030-68734-2_2

[53] V. Karande and E. Bauman and Z. Lin and L. Khan, "SGX-Log: Securing System Logs With SGX", Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17), Abu Dhabi, United Arab Emirates, April 2017, DOI 10.1145/3052973.3053034

[54] R. Paccagnella and P. Datta and W. Ul Hassan and A. Bates and C.W. Fletcher and A. Miller and D. Tian, "CUSTOS: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution", Network and Distributed Systems Security (NDSS) Symposium 2020, San Diego, CA, USA, February 2020, DOI 10.14722/ndss.2020.24065

[55] G.M. Hamidy and P. Philippaerts and W. Joosen, "T3E: A Practical Solution to Trusted Time in Secure Enclaves", Network and System Security (NSS 2023), Cham, 2023, DOI

10.1007/978-3-031-39828-5_17

[56] N. Kaaniche and S. Belguith and M. Laurent and A. Gehani and G. Russello, "Prov-Trust: Towards a Trustworthy SGX-based Data Provenance System", 17th International Conference on Security and Cryptography (SECRYPT 2020), Paris, France, July 2020, pp. 225–237, DOI 10.5220/0009889302250237

[57] F. Toffalini and A. Oliveri and M. Graziano and J. Zhou and D. Balzarotti, "The Evidence Beyond the Wall: Memory Forensics in SGX Environments", Forensic Science International: Digital Investigation, vol. 39, 2021, p. 301313, DOI 10.1016/j.fsidi.2021.301313

[58] Intel Corporation, "Input Types and Boundary Checking in Enclave-Definition Language (EDL) Files", 2017. White Paper. https://software.intel.com/en-us/sgx

[59] M.U. Sardar and D.L. Quoc and C. Fetzer, "Towards Formalization of Enhanced Privacy ID (EPID)-based Remote Attestation in Intel SGX", 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 2020, pp. 604–607, DOI 10.1109/DSD51259.2020.00099

[60] M.U. Sardar and R. Faqeh and C. Fetzer, "Formal Foundations for Intel SGX Data Center Attestation Primitives", Formal Methods and Software Engineering (ICFEM 2020) (S.W. Lin and Z. Hou and B. Mahony, ed.), Cham, 2020, pp. 268–283, DOI 10.1007/978-3-030-63406-3_16

[61] Microsoft, "Trusted Hardware Identity Management", https://learn.microsoft.com/en-us/azure/security/fundamentals/trusted-hardware-identity-management, 2025, Microsoft Azure Documentation

[62] Intel Corporation, "Intel® Software Guard Extensions (Intel® SGX) Data Center Attestation Primitives", https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html, Intel Support Article ID: 000057968

[63] A. Havet and R. Pires and P. Felber and M. Pasin and R. Rouvoy and V. Schiavoni, "SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing", Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17), June 2017, pp. 124–133, DOI 10.1145/3093742.3093927