2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P) | 979-8-3315-9493-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/EuroSP63326.2025.00028

LATTE: Layered Attestation for Portable Enclaved Applications

Haoxuan Xu*, Jia Xiang*, Zhen Huang, Guoxing Chen†, Yan Meng, Haojin Zhu
Shanghai Jiao Tong University, China
{*SJTUxhx20010613, jiaxcn99, xmhuangzhen, guoxingchen, yan_meng, zhu-hj*}*@sjtu.edu.cn*

***Abstract*—Trusted Execution Environment (TEE) has become increasingly popular in privacy-protected cloud computing, and its rapid development has led to the availability of various heterogeneous TEE platforms on cloud servers. To facilitate portable TEE applications on heterogeneous TEE platforms, portable languages or intermediate representations (IRs) with platform-dependent TEE runtimes are adopted. However, existing remote attestation solutions for portable TEE applications follow a nested attestation pattern, i.e., attesting only the TEE runtime and relying on the TEE runtime to measure the loaded portable application, leading to potential security issues. On the other hand, directly packing the TEE runtime and the portable application into an enclave for secure attestation undermines the portability of the portable TEE applications.**

In this paper, we introduce the concept of portable identities to identify portable TEE applications, and propose a layered attestation framework, LATTE, achieving both security and portability in attesting portable TEE applications. We provide a prototype implementation of LATTE to validate its practicality, with WebAssembly as the portable IR, and Intel SGX and RISC-V Penglai as the exemplar heterogeneous TEEs. The evaluation demonstrates that LATTE introduces minimal performance overhead compared with the nested attestation pattern.

1. Introduction

With the growing emphasis on privacy protection and the popularity of cloud computing, numerous works have proposed the applications of trusted execution environments (TEEs) in diverse cloud scenarios to preserve privacy [1], [2], [3], [4], [5], [6], [7]. TEE is a secure area (usually called *enclave*) within a CPU to protect the confidentiality and integrity of code and data loaded inside. Prominent examples include Intel SGX [8], Intel TDX [9], AMD SEV [10] and ARM CCA [11]. Recently, various RISC-V based TEEs have been developed, such as Penglai [12] and Keystone [13].

The rapid development of heterogeneous TEEs has spurred the demand for portable TEE applications [14], [15], [16], [17]. Particularly, portable TEE applications are developed using portable languages or recompiled to portable Intermediate Representations (IRs), and then executed by platform-dependent *TEE runtimes* on different TEE platforms. For example, WebAssembly Micro Runtime (WAMR) [16] selects WebAssembly (WASM) [18] as the portable IR and implements a lightweight standalone WASM runtime that currently supports Intel SGX. Similarly, Enarx [17] provides WASM runtimes supporting both Intel SGX and AMD SEV.

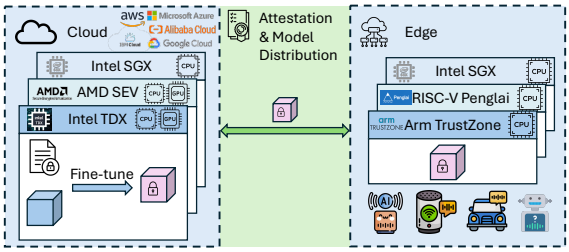

Figure 1: An example of securely deploying AI models using TEEs across heterogeneous devices.

This method enables seamless deployment of the same code (referred to as *portable payload* throughout this paper) across diverse heterogeneous TEE platforms, opening up many potential application scenarios. For instance, with the emergence of diverse heterogeneous TEE platforms on cloud servers, the concept of confidential serverless computing has been proposed and widely studied [19], [20], [21], [22]. This approach not only enables developers to disregard the complex specifications of TEEs, but also allows cloud providers to enhance server utilization based on availability. Furthermore, it supports the execution of uniform privacy-preserving applications across distributed or decentralized devices of diverse architectures. An illustrative example, as shown in Fig. 1, is the use of TEEs in securely deploying AI models: a user may wish to fine-tune a model using her own private data - such as a voice assistant - potentially on cloud servers equipped with TEEs and optional confidential GPU support. Once fine-tuned, the model can be distributed across her various devices (e.g. smart speaker, robots) for inference, which may lack significant computing power. Since both the cloud servers and the devices might use different architectures, the portable payload deployment method ensures that the fine-tuning and inference code need to be implemented once and can be executed with the TEE runtime of different architectures.

However, it is challenging to achieve both strong security and portability in authenticating portable TEE applications via existing remote attestation solutions (detailed in Sec. 3). Remote attestation is a mechanism that empowers one entity, denoted as the *verifier*, to verify the trustworthiness of the underlying hardware of a communicating enclave, denoted as the *attester*, and obtain the attester's *identity*. One major challenge in authenticating portable TEE applications is to securely obtain the portable payload's identity (which should be platform-

*Equal contribution.
†Corresponding author.

DOI 10.1109/EuroSP63326.2025.00028

independent), given that the attester enclave in typical remote attestation is commonly identified by its *measurement*, i.e., the cryptographic hash (using platform-dependent hashing algorithms) of its initial code and data including platform-dependent TEE runtimes. To tackle this problem, software-based *nested attestation* schemes have been introduced [23], [22], [24], [25]. Specifically, the attester enclave is firstly launched with only a vanilla TEE runtime. The verifier then attests the trustworthiness of the TEE runtime, and relies on the attested runtime to load the portable payload and calculate its hash. As long as different TEE runtimes use the same cryptographic hashing algorithm for the portable payload, the resulting hash values become portable across all these heterogeneous TEE platforms.

Nevertheless, such a nested attestation process has security issues when the isolation between the runtime and the payload is not securely enforced (detailed in Sec. 3.1): when an adversary could compose a malicious payload that could compromise the whole runtime and take control of it, the compromised runtime could impersonate a benign runtime and successfully complete the attestation process since the measurement of the runtime is fixed when it is launched, and matches the known-good measurement of a benign runtime. A recent CVE [26] reported a bug in a popular WASM runtime that could be abused to grant unintended read and write access to the payload even when software fault injection (SFI), a technique for isolating the runtime and its payload, is in place. As a result, taking the private AI model deployment in Fig. 1 as an example, a TEE runtime compromised by a malicious payload can be abused by the adversary to steal the private model.

The security issue of nested attestation might be addressed by hardcoding the portable payload into the initial data of the attester enclave along with the TEE runtime and enforcing the TEE runtime to load only the hardcoded payload, so that the measurement binds with the hardcoded portable payload. However, this undermines the portability of the TEE applications, since platform-dependent information such as the binding between the platform-dependent measurement and the portable payload might be integrated into the portable payload for the verification. For example, considering the portable TEE application P_A to fine-tune a private model with sensitive data and the portable TEE application P_B that receives the model for inference, for P_A to attest P_B's identity to send the private model, measurements of potential heterogeneous TEE runtimes with hardcoded P_B need to be pre-encoded into P_A. This method faces a hurdle when any runtime undergoes an update, as it mandates a corresponding update to the payload to align with the updated measurements (detailed in Sec. 3.2). Such a requirement deviates from the initial goal of facilitating portable development, revealing a deep-seated issue stemming from an inadequate decoupling between the runtime and the payload development. Moreover, when the user needs to update the inference code, the fine-tuning enclave must include measurements of all enclaves for that algorithm across various platforms, which is cumbersome.

In this paper, we introduce the concept of *portable identity*, i.e., the hash of the payload, and propose LATTE, a layered attestation framework devised to ensure both security and genuine portability throughout the entire lifecycle of portable TEE application development and deployment. In particular, this is facilitated with three key techniques: *restricted payload loading*, *identity-measurement binding*, and *layered reference-value derivation*. Firstly, restricted payload loading allows for only loading a payload whose portable identity matches a reference value hardcoded into a TEE runtime, so that the integrity of the expected payload can be reflected by the measurement of the launched enclave. Secondly, the identity-measurement binding permits the verification of the binding between a portable identity and a measurement. Lastly, the layered reference-value derivation enables the separation of the attestation process into two layers, one for verifying the underlying runtime and one for verifying the portable payload. This layering decouples components, enabling updates to one (e.g., the runtime) without requiring updates to the other (e.g., the portable payload), aside from lightweight rebuilds of affected enclaves, thereby addressing the aforementioned portability issue.

We provide a prototype implementation of LATTE by adopting WASM as the portable IR, WAMR as the TEE runtime, and Intel SGX and RISC-V Penglai are demonstrated as examples of heterogeneous TEEs with different ISAs (i.e., x86 and RISC-V, respectively).

In sum, the main contributions of this paper are:

- It proposes LATTE, a novel solution to address the security and portability issues in attesting portable TEE applications.
- It presents key techniques to realize layered attestation: restricted payload loading, identity-measurement binding, and layered reference-value derivation.
- It implements and evaluates a prototype of LATTE with WASM as the portable IR, and Intel SGX and Penglai as the exemplar heterogeneous TEEs. The prototype implementation and evaluation are available at https://github.com/Jiax-cn/latte.

Ethics Considerations. Our research poses no foreseeable harm to individuals, deployed systems, or stakeholders, and IRB consultation was not necessary for this work.

2. Background

2.1. Trusted Execution Environment

Trusted Execution Environment (TEE). A Trusted Execution Environment (TEE) on a CPU is a secure and isolated area to protect the confidentiality and integrity of the code and data within the area from malicious operating systems. To improve the security of data and code in applications, there are many TEEs implemented by different ISAs, like Intel Software Guard Extensions (SGX) [8], AMD Secure Encrypted Virtualization (SEV) [10], ARM TrustZone, Keystone [13] and Penglai [12] in RISC-V.

Intel Software Guard Extensions (SGX). SGX [8] is a shielded execution environment on an Intel processor. On an Intel SGX platform, a specified memory region, called Processor Reserved Memory (PRM), is reserved for enclave-only memory access. The data and code in the enclave are stored in the Enclave Page Cache (EPC) which is held by the PRM [27].

Penglai. Penglai [12] is a RISC-V TEE system that comprises three projects designed for various scenarios. One of these projects, Penglai-TVM, is a pure software enclave design version that supports fine-grained isolation between untrusted hosts and enclaves. Penglai-TVM consists of three key submodules: the Linux kernel, Software Development Kit (SDK) and RISC-V Open Source Supervisor Binary Interface (OpenSBI). The Linux kernel supports guard page table by utilizing RISC-V Trap Virtual Memory (TVM), while the SDK includes the host and enclave library and enclave driver. Penglai OpenSBI includes the secure monitor responsible for enclave management and maintaining security guarantees. Additionally, Penglai offers QEMU support [28] for enclave application execution on architectures other than RISC-V.

Remote Attestation. Remote attestation is a significant component of TEEs that relies on cryptographic signatures and endorsement certificates to establish trust between attesters and verifiers. In this process, an attester provides attestation evidence to a verifier, which undergoes two sequential verification phases: (1) verifying that the evidence originates from trusted hardware by validating the cryptographic signature produced by the attestation key, and (2) comparing the measurement in the evidence against pre-established reference values (e.g., known-good measurements) to confirm trustworthiness. In the case of Intel SGX, remote attestation evidence needs to be verified by the Intel Attestation Service (IAS) [29], allowing verifiers to obtain the information about the attester's hardware platform and software environment.

Measurement in Intel SGX. Intel SGX uses SHA-256, a Secure Hash Algorithm(SHA) to generate 256-bit digests. In Intel SGX, the measurement is calculated as enclave pages are loaded one by one during the enclave creating.

Measurement in Penglai. Penglai uses ShangMi 3 (SM3) [30], a Chinese National Standard cryptographic hash function, for enclave measurement calculation. SM3 has been recognized by the ISO/IEC international standard [31] and is considered similar to SHA-256 in both structure and security [32]. The enclave measurement is generated by a secure monitor which traverses the enclave memory from the lowest address to the highest address.

Reference Values. A reference value [33] is a pre-established value provided to the verifier to determine whether a received measurement is trustworthy. Reference values can also be called "known-good values" or "golden measurements". In this paper, a *reference measurement* is short for the reference value of a measurement, and similarly *reference portable identity* is short for the reference value of a portable identity.

MAGE. MAGE [34] enables a group of enclaves to mutually attest each other without relying on a trusted third party to provide reference measurements. The method leverages the sequential nature of cryptographic hash calculations in existing TEEs. Specifically, the hash computation process involves initializing an intermediate hash state, updating it sequentially with each block of enclave content, and finalizing the state to produce the final hash value (a detailed explanation of this property is provided in Appendix A). Importantly, the final hash value can be derived from the intermediate hash states and the remaining blocks to be processed. To facilitate this, MAGE introduces a shared common component among a group of trusted enclaves. This component stores the intermediate hash states of each enclave, which reflect their respective contents excluding the shared component itself. By retrieving the intermediate hash state of another enclave from the common component and updating it with the remaining enclave content (i.e., the shared component), each enclave can independently derive the reference measurements of its peers. This eliminates the need for a trusted third party while ensuring secure mutual attestation.

The idea of leveraging the sequential nature of cryptographic hash calculations for reference measurement derivation also inspires the design of LATTE.

2.2. Portable Programming

Portable Programming Languages with Runtimes. A portable programming language with a runtime (e.g., a virtual machine or interpreter) provides an execution environment that abstracts away platform-specific details. Applications developed with such languages can run on any system with the corresponding runtime, allowing developers to focus on the core logic of applications. Examples include Java, which relies on the Java Virtual Machine, and Python, which executes code via its interpreter.

Intermediate Representation (IR). Intermediate representation (IR) is an abstract expression used internally to represent source code. IR has the same logic as the applications written in the higher-level language, and it can be further processed to lower-level machine code or directly executed in the platform-specific runtime. IR can be both portable and non-portable. The portable ones are platform-agnostic and not specific to any particular machine or platform, such as WebAssembly.

WebAssembly. WebAssembly (WASM) is a lightweight binary-code instruction format and open standard developed by W3C Community Group [18]. It is also a portable IR. Applications developed with various programming languages (e.g. C/C++, Rust) can be compiled into unified WASM codes. While WASM is originally designed for the performance of applications on the Web, it has since extended to non-web environments through specialized runtimes such as Wasmer [35], Wasmtime [36], and WebAssembly Micro Runtime (WAMR) [16].

3. Motivation

In this section, we motivate our work by analyzing the limitations of existing and potential strawman solutions for attesting portable TEE applications.

Scope. We focus on WASM due to its popularity in both academia [14], [15] and industry [16], [17]. Additionally, we focus on more general scenarios where an application comprises multiple portable payloads, with one acting as the verifier for the others. Such scenarios include, but are not limited to secure software development, secure distributed and decentralized applications [34]. Scenarios where a single portable payload requires attestation by a relying party, where the verifier is not an enclave, represents simpler cases and can be addressed similarly as special cases of the scenarios discussed herein.

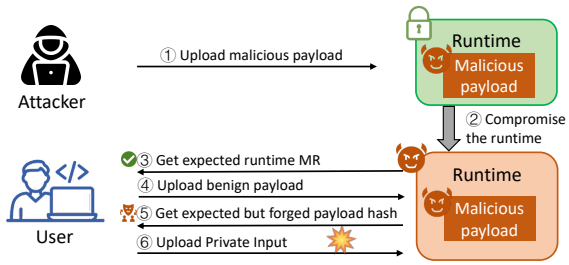

Figure 2: An undetectable attack for the *nested attestation*.

3.1. Nested Attestation Solutions

WAMR [16] and Enarx [17] are two prominent open-sourced WASM runtimes that support TEEs. Both adopt a nested attestation approach to attesting WASM applications. Specifically, a vanilla runtime without any portable payload is firstly launched and attested. The attested runtime is leveraged to load and attest a WASM payload. For example, an attested WAMR [24] runtime calculates the hash of the loaded portable payload and provides this hash to the verifier for subsequent verification, e.g., comparing the received hash with some reference value. Slightly differently, an attested Enarx [17] runtime directly assesses the integrity of the payload using a prefetched authoritative hash (reference value) by itself from drawbridge [37]. Drawbridge is a public repository service for Enarx, allowing developers to publish their developed applications. Deployed runtimes can also download applications from it and perform the aforementioned verification process.

As the hardware-based measurement only reflects the initial state when the vanilla runtime is launched, the security of such nested attestation approach highly relies on the correct and secure implementation of sandboxing techniques that isolate the runtime and the payload.

Taking WAMR as an example, without proper isolation between the runtime and the payload, an adversary could provide a malicious payload that could compromise the whole runtime and take control of it (Step ① to ②), as shown in CVE-2023-26489 [26]. Since the measurement is fixed after the runtime is initialized, it will persist even when the runtime is compromised later. Hence, the user could not distinguish between a genuine runtime and a compromised one solely by verifying the runtime's measurement (Step ③). This implies that when the user receives an expected portable measurement and assumes that both her remote runtime and payload are secure, she may not, in fact, realize that she is in a situation where the remote runtime has already been compromised by an attacker in advance, and the received payload hash is forged (Step ④-⑤). Consequently, sensitive data may be inadvertently exposed (Step ⑥), as illustrated in Fig. 2. Although the mechanism of Enarx differs slightly, it remains vulnerable to a similarly structured attack.

However, the correct and secure implementation of such sandboxing techniques is quite challenging, as various CVEs about sandbox escape vulnerability are continuously reported [38], [39], [40], casting shadow on the security of nested attestation solutions.

3.2. Strawman Solutions with Hardcoded Portable Payload

Hardcoding the portable payload into the initial data of the attester enclave could bind the resulting measurement with the hardcoded payload, thus resolving the above security issue. With such an approach, the same payload yields diverse measurements on different platforms. This variation necessitates that the verifier enclave (particularly its loaded payload who needs to determine the trustworthiness of an attester enclave before interacting with it) needs a reliable source to obtain the reference values of the binding information about the payload and the measurement for identifying the portable payload from these platform-dependent measurements. There are intuitively three approaches to realizing such reliable sources:

Introduce a trusted third party to maintain and provision the binding information. As an alternative, introducing a trusted third party to oversee the binding information could effectively decouple runtime-dependent data from the payload. This setup enables the portable payload to merely retain the necessary information for attesting the trusted third party. Upon receiving a measurement, the verifier enclave can query the trusted third party for the identity of its corresponding portable payload, thus determining its trustworthiness.

Nevertheless, this method introduces a single-point-of-failure and expands the TCB of the whole system. Specifically, centralizing binding information in one entity might also lead to reliability concerns if it experiences any failure. Moreover, adding this third party as a new component of the TCB introduces an additional element that must be trusted, further complicating security assurances.

Include the binding information into the portable payload. Straightforwardly, when the verifier receives a measurement from the attester enclave, it can refer to the hardcoded binding information to verify the trustworthiness of the corresponding portable payload run within the attester enclave.

However, this method undermines the portability. Since part of the binding information, specifically the resulting measurements, depends on both the portable payload and the potential underlying TEE runtimes. This inadequate decoupling might lead to portability issues. For example, when any platform's runtime is updated, the portable payloads would need to be updated to accommodate these changes, even when the logic of the payloads stays the same.

Include the binding information into the final enclave. Instead of including the binding information into the portable payload, it is also possible to include the binding information into the final enclave. Particularly, MAGE [34] introduces a measurement derivation mechanism that enables a group of enclaves to mutually derive each other's measurements without trusted third parties. Considering L portable payloads, each of which could be run on any one of N TEE runtimes, we can apply MAGE to enable a group of all $N \times L$ potential final enclaves to mutually derive each other's measurements and identify the corresponding portable payloads.

Nonetheless, this method requires the built final enclaves to facilitate the mutual measurement derivation,

thus suffering from similar portability issues as that includes the binding information into the portable payload. Particularly, the current design of MAGE does not support enclave updates, that is, once any of the portable payloads get updated, all the final enclaves need to be rebuilt to reenable the mutual attestation.

Summary. Nested attestation solutions adopted by WAMR and Enarx suffer from security issues, while strawman approaches with hardcoded payloads, including MAGE, struggle with portability. These limitations lead to the research problem: ***can we achieve both security and portability in attesting portable TEE applications?***

4. LATTE Overview

This section outlines the threat model, followed by an overview of the attestation workflow using LATTE and three key enabling techniques.

4.1. Threat Model

We assume that TEEs have been properly implemented at the hardware level of target TEE platforms, and the remote attestation mechanisms provided by these TEE implementations are secure. That is, the adversary could not compromise the hardware-backed measurement mechanism, nor collude with the manufacturer-backed remote attestation service to forge a benign measurement for a malicious enclave to bypass the remote attestation. We assume the adversary has full control over the hypervisor, operating system and other system softwares outside the enclave, and thus could create and run any malicious application on the host platform. However, we do not take into account micro-architectural side-channel attacks [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], nor denial-of-service attacks targeting the mechanisms responsible for the identity-measurement binding and the derivation of reference values.

In line with the potential attack presented in Sec. 3.1, we do not assume secure isolation (e.g., correctly and securely implemented SFI) between the TEE runtime and the payload application. That is, the TEE runtime might have software vulnerabilities that could be abused by a carefully crafted payload with malicious code to take over control of the TEE runtime, as demonstrated by the CVE-2023-26489 [26]. However, we do not assume such malicious code to be prevalent in benign payloads. That is, the adversary could not leverage an arbitrary benign payload to compromise the TEE runtime. Hence, the security goal of LATTE is to ensure that the authenticity of such benign payloads executed within TEE runtimes is endorsed by the hardware-backed measurements.

4.2. Attestation Workflow

Similarly to nested attestation solutions adopted by WAMR and Enarx, and mutual attestation solution MAGE, LATTE does not replace the traditional attestation workflow, but works alongside it. The attestation workflow with LATTE is illustrated in Fig. 3. Step ❶ and ❷ are generally from the traditional attestation workflow, while step ❸ and ❹ are introduced by LATTE.

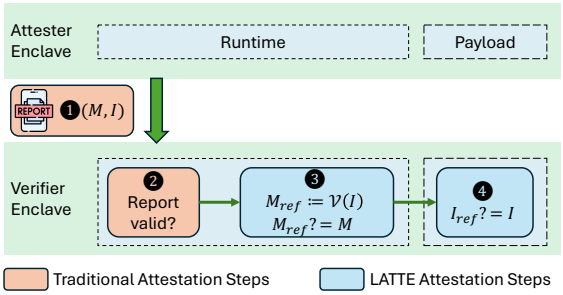

Figure 3: LATTE attestation workflow, where $\mathcal{V}$ is a derivation function that takes a portable identity I as input and outputs the corresponding reference measurement M_{ref}.

Step ❶. An attester enclave follows the traditional attestation process to generate attestation evidence, in the form of an attestation report containing (1) the measurement M of the attester enclave collected by the underlying hardware, and (2) a portable identity I provided by the software running within the enclave. The attestation report is forwarded to the verifier enclave.

Step ❷. The verifier enclave verifies the authenticity of the attestation report by verifying its associated signature, which is signed using the attestation key from the trusted TEE platform that runs the attester enclave. This verification step is still part of the traditional attestation process.

Step ❸. The verifier enclave (particularly the TEE runtime within) then verifies that the binding between the measurement M and the portable identity I within the attestation report is authentic. Verification involves the derivation of a reference measurement M_{ref} of an enclave within which a specified *trusted TEE runtime* runs a portable payload with the portable identity I. If the received measurement M equals to M_{ref}, the binding between M and I can be trusted and I will be forwarded to the portable payload within the verifier enclave.

Step ❹. The portable payload within the verifier enclave derives a reference portable identity I_{ref} to be compared with the portable identity I from its underlying TEE runtime. If they are equal, the trust on the portable payload within the attester enclave can be established.

4.3. Key Techniques

As discussed in Sec. 3, we aim to achieve both security and portability in attesting portable TEE applications. Specifically, the final enclave should be built from both the runtime and the portable payload to ensure security. For portability, the portable identity serves as an additional binding element alongside the measurement, so that the runtime and portable payload are decoupled to allow for their independent updates.

To achieve the above goals, LATTE introduces three key techniques as follows:

- *Restricted Payload Loading.* LATTE incorporates the portable payload or its identity in the final enclave, and enforces the runtime to either fetch the payload internally or verify the identity of an externally received payload before execution. Consequently, the hardware-calculated measurement of the enclave is bound to

the payload, addressing the security issues with nested attestation solutions.

- *Identity-Measurement Binding.* To facilitate portability, LATTE introduces a portable identity which is independent of the underlying platform and solely associated with the portable payload. A mechanism is introduced to enable the verifier enclave to securely verify the binding between a portable identity and its underlying TEE runtime.
- *Layered Reference-Value Derivation.* To decouple the runtime and portable payload for allowing their independent updates, LATTE introduces a layered reference-value derivation mechanism, enabling the runtime and portable payload within the verifier enclave to derive reference values of the hardware-backed measurement and the portable identity, respectively. The derived reference measurement is used to verify the trustworthiness of the TEE runtime within the attester enclave, while the derived reference portable identity is used to authenticate the portable payload within the attester enclave.

4.4. Comparison with State-of-the-Art

We compare LATTE with WAMR and MAGE, representing nested attestation solutions and strawman approaches with hardcoded portable payloads, respectively. This comparison highlights LATTE 's advantages in both security and portability.

Security. WAMR fails to provide strong security due to the potentially insecure isolation between the TEE runtime and the payload application, as discussed in Sec. 3.1. In contrast, both MAGE and LATTE ensure security by constructing the final enclave from both the runtime and the payload. This guarantees that the hardware-calculated enclave measurement authenticates both components.

Portability. WAMR achieves portability by fully decoupling the payload from the runtime, allowing that updates to one component do not affect the other. In contrast, MAGE lacks portability because the common components of all enclaves within the application must be rebuilt whenever any of them is updated. Since the enclave is built from both the portable payload and the runtime, any modification to either necessitates rebuilding all enclaves. LATTE addresses this limitation through a layered reference-value derivation mechanism (detailed in Sec. 6), which decouples runtime and payload verification. For example, runtime updates require modifications only to the runtime layer, leaving payloads unchanged, and vice versa. A detailed analysis of LATTE 's portability will be presented in Sec. 6.5.

5. Problem Formulation

In this section we formulate the problem of attesting portable TEE applications with strong security and portability guarantees. Notations used in this paper are listed in Table 1, with those in blue related to *portable* payloads and those in red representing *platform-dependent* components encompassing the runtime, build functions and the final built enclave.

TABLE 1: Notations in this paper.

Symbol	Description
P, P_l	Portable code
I, I^P	Portable identity
$\overline{I^P}$	Intermediate hash state of portable identity
R_i	TEE runtimes
E_i^P	Enclave content
M_i^P	Enclave measurement
$\overline{M_i^P}$	Intermediate hash state of measurement
$\mathcal{B}_i$	Build functions
$\widetilde{\mathcal{B}_i}$	Simplified versions of build functions
M_i	Cryptographic hash functions for measurements
I	Cryptographic hash function for portable identities
$\mathcal{P}$	Portable identity generation function
$\mathcal{V}$	Identity-measurement verification function
$\mathcal{G}^p$	Portable payload common part generation function
$\mathcal{F}^p$	Reference portable identity derivation function
$\mathcal{G}^r$	TEE runtime common part generation function
$\mathcal{F}^r$	Reference measurement derivation function

We first formally define the process of building enclaves from portable payloads and TEE runtimes for heterogeneous TEE platforms, portable identities and measurements.

Definition 1. *Consider a portable payload* `PPld` *that has portable code* P *and could be run on any one of* N *TEE platforms* $\texttt{TPtf}_i$ $(i = 1, \ldots, N)$*, each of which has a platform-specific TEE runtime* R_i*. We define a build function* $\mathcal{B}_i$ *that is used to build the content* E_i^P *of the enclave* $\texttt{Encl}^P{}_i$ *that could execute* `PPld` *on* $\texttt{TPtf}_i$*:*

$$E_i^P \leftarrow \mathcal{B}_i(R_i, P)$$

The enclave's measurement M_i^P *and the portable identity* I *of* `PPld` *are defined as the cryptographic hash values of* E_i^P *and* P*, using cryptographic hash functions* M_i *and* I*, respectively:*

$$M_i^P \leftarrow \mathsf{M}_i(E_i^P); \quad I^P \leftarrow \mathsf{I}(P).$$

Since existing remote attestation provides a hardware-level integrity guarantee of the measurement, we need additional mechanisms to verify the binding between portable identities and measurements. On the one hand, an attester enclave should be able to provide the portable identity of the payload executed within its TEE runtime so that the portable identity could be included in the attestation evidence for verification. On the other hand, similar to that one enclave could also act as the verifier to verify the integrity of a received measurement of another enclave, it is desired that the built enclaves in our design could also act as the verifier to verify the integrity of the binding between a portable identity and a measurement. Specifically, when given a pair of portable identity and measurement (I, M), a verifier enclave could verify whether M is the measurement of an attester enclave `Encl` that executes `PPld` with portable identity I running on some `TPtf`.

Hence, the ***security*** of the research problem in this paper can be formalized and addressed by the mechanism (denoted by identity-measurement binding mechanism) defined as follows:

Definition 2. *Consider a portable payload* `PPld` *that has portable code* P *and could be run on any one of* N *TEE platforms* $\texttt{TPtf}_i$ $(i = 1, \ldots, N)$*, each of which has a*

platform-specific TEE runtime R_i and a build function $\mathcal{B}_i$. An identity-measurement binding mechanism consists of two functions ($\mathcal{P}$, $\mathcal{V}$):

- *$\mathcal{P}$ is called portable identity generation function, which takes as input an enclave built from P and a TEE runtime R_i, and outputs the portable identity of P, specifically, $\mathcal{P}$ satisfies*

$$\mathcal{P}(\mathcal{B}_i(R_i, P)) = \mathsf{I}(P) \tag{1}$$

- *$\mathcal{V}$ is called identity-measurement verification function, which takes as input a final enclave built from P and a TEE runtime R_i, a TEE runtime index j ($= 1, \ldots, N$), a portable identity I and a measurement M, and outputs 1 only when there exists a portable payload with portable code $\hat{P}$ running within a TEE runtime R_j such that $I = \mathsf{I}(\hat{P})$ and $M = \mathsf{M}_j(\mathcal{B}_j(R_j, \hat{P}))$, specifically, $\mathcal{V}$ satisfies*

$$\mathcal{V}(\mathcal{B}_i(R_i, P), I, M, j) = \begin{cases} 1, & \exists \hat{P} : I = \mathsf{I}(\hat{P}) \wedge \\ & \quad M = \mathsf{M}_j(\mathcal{B}_j(R_j, \hat{P})) \\ 0, & otherwise \end{cases} \tag{2}$$

Specifically, $\mathcal{P}$ is leveraged by a final enclave (acting as an attester enclave) to attest itself to another entity while $\mathcal{V}$ is utilized by a final enclave (acting as a verifier enclave) to verify the integrity of the binding between a portable payload and another final enclave.

Note that the identity-measurement binding mechanism does not help determine whether the verified portable identity and measurement are trustworthy or not. Usually, some *reference values* are needed to be compared with to determine the trustworthiness.

To achieve high *portability*, it is desired the trustworthiness of the portable identity and the measurement can be determined by the portable payload and the underlying TEE runtime individually, so that the development of portable payloads and the development of the TEE runtimes can be decoupled.

Particularly, a TEE runtime $R_i, i = 1, \ldots, N$ should be able to derive the reference measurement of a final enclave built from some portable payload with a given portable identity I and a TEE runtime $R_j, j = 1, \ldots, N$. While $i \neq j$, additional information are needed for the derivation as discussed by Chen *et al.* [34]. Inspired by their solution which does not require any trusted-third party to provide such information, we assume each TEE runtime is in the form of $R_i = R_i' || R_{\text{common}}$, i.e., the concatenation of its specific part R_i' and a common part R_{common}. Particularly, the specific part includes all necessary functionalities to execute the payload on the underlying platform, while the common part is shared by all N TEE runtimes to be used for reference measurement derivation. Ideally, R_{common} should be generated solely from the specific parts of the TEE runtimes without any knowledge of the potential payloads.

Similarly, when a group of L portable payloads $\texttt{PPld}_l$ ($l = 1, \ldots, L$), each of which with portable code P_l would like to derive the reference portable identities of the other portable payloads, we also assume the portable code P_l is in the form of $P_l = P_l' || P_{\text{common}}$, i.e., the concatenation of its specific part P_l' and a common part P_{common} shared by all L $\texttt{PPld}$s. The common part enables portable payloads to mutually attest each other without any trusted third party, which is desired for multi-enclave applications, as discussed by Chen *et al.* [34]. Note that portable TEE applications without the need of mutual attestation can be treated as a special case when $L = 1$. Ideally, the generation of P_{common} should be independent of the underlying TEE platforms.

Hence, the ***portability*** of the research problem in this paper can be formalized and addressed by the mechanism (denoted by reference-value derivation mechanism) defined as follows:

Definition 3. *Consider a group of L portable payloads $\texttt{PPld}_l$ ($l = 1, \ldots, L$), each of which has portable code $P_l = P_l' || P_{common}$ and could be run on any one of N TEE platforms $\texttt{TPtf}_i$ ($i = 1, \ldots, N$), each of which has a platform-specific TEE runtime $R_i = R_i' || R_{common}$ and a build function $\mathcal{B}_i$. Cryptographic hash functions M_i and I are adopted to calculate measurements and portable identities. A reference-value derivation mechanism of two pairs of functions (($\mathcal{G}^p, \mathcal{F}^p$), ($\mathcal{G}^r, \mathcal{F}^r$)):*

- *$\mathcal{G}^p$ is called portable payload common part generation function, which takes as input only the specific parts of L portable payloads, and outputs the portable payload common part P_{common}, specifically, $\mathcal{G}^p$ satisfies*

$$P_{common} \leftarrow \mathcal{G}^p(P_1', \ldots, P_l') \tag{3}$$

- *$\mathcal{F}^p$ is called reference portable identity derivation function, which takes as input the portable payload common part P_{common} and an index k ($= 1, \ldots, L$), and outputs the portable identity of $\texttt{PPld}_k$, specifically, $\mathcal{F}^p$ satisfies*

$$\mathcal{F}^p(P_{common}, k) = \mathsf{I}(P_k) \tag{4}$$

- *$\mathcal{G}^r$ is called TEE runtime common part generation function, which takes as input only the specific parts of N TEE runtimes, and outputs the TEE runtime common part R_{common}, specifically, $\mathcal{G}^r$ satisfies*

$$R_{common} \leftarrow \mathcal{G}^r(R_1', \ldots, R_N') \tag{5}$$

- *$\mathcal{F}^r$ is called reference measurement derivation function, which takes as input the TEE runtime common part R_{common}, a TEE runtime index j ($= 1, \ldots, N$), a portable identity I. If there exists a portable payload with portable code $\hat{P}$ such that $I = \mathsf{I}(\hat{P})$, $\mathcal{F}^r$ outputs the measurement of the final enclave built from $\hat{P}$ and TEE runtime R_j. Otherwise, $\mathcal{F}^r$ outputs $\perp$ (indicating the failure of measurement derivation. Specifically, $\mathcal{F}^r$ satisfies*

$$\mathcal{F}^r(R_{common}, j, I) = \begin{cases} \mathsf{M}_j(\mathcal{B}_j(R_j, \hat{P})), & \exists \hat{P} : I = \mathsf{I}(\hat{P}) \\ \perp, & otherwise \end{cases} \tag{6}$$

Note that $\mathcal{G}^p$ and $\mathcal{F}^p$ can be solely adopted by portable TEE application developers, while $\mathcal{G}^r$ and $\mathcal{F}^r$ can be employed by TEE runtime developers on their own, thus achieving the portability.

6. LATTE Design

In this section, we detail the design of LATTE for attesting portable TEE applications with strong security

and portability. Specifically, we present solutions to address the following challenges: (1) how to enable a measurement to guarantee the integrity of loaded portable payload (Sec. 6.1), (2) how to generate and verify the binding between a portable identity and a measurement (Sec. 6.2), (3) how to facilitate the portable payload and the TEE runtime to derive reference values for verification separately (Sec. 6.3). Combining these techniques, we describe the workflow of LATTE (Sec. 6.4) and analyze its security and portability (Sec. 6.5).

6.1. Restricted Payload Loading

To address the security issues of nested attestation, the identity of the portable payload to be loaded in the TEE runtime should be bound to the hardware-backed measurement to enjoy the architecture-level integrity guarantee. There are generally two methods to achieve this:

- *Hardcoding Portable Payload.* A straightforward approach is to hardcode the entire portable payload into the enclave. After launch, the runtime within the enclave executes only the hardcoded payload, as discussed in Sec. 3.2.
- *Hardcoding Portable Identity.* An alternative approach is to hardcode only the portable identity into the enclave. Upon receiving a portable payload, the enclave calculates the portable identity of the incoming payload, and proceeds with loading and execution only if this calculated portable identity matches the hardcoded value. A simplified version of the build function $\widetilde{\mathcal{B}}_i$ that takes as input the portable identity instead of the portable payload can be defined as

$$E_i^P \leftarrow \mathcal{B}_i(R_i, P) = \widetilde{\mathcal{B}}_i(R_i, \mathsf{I}(P))$$

Both methods could securely bind the identity of the portable payload with the hardware-backed measurement. One obvious difference is in the sizes of the built enclaves. Hardcoding the entire portable payload results in larger enclaves, thus incurring more storage and/or communication overhead than hardcoding only the portable identity. For example, to support one portable payload on N TEE platforms, N final enclaves will be built and maintained. Each final enclave has a full copy of the portable payload within its initial data. On the other hand, hardcoding only the portable identity requires to store only one copy of the portable payload. Throughout this section, we will discuss more advantages of hardcoding the portable identity over hardcoding the entire portable payload, and adopt the former in our final design.

One concern is that *such restricted payload loading undermines the TEE runtime's ability to load various payloads based on user requests.* Particularly, will it hamper the sharing of the TEE runtime across different portable payloads, which requires the deployment of multiple enclaves with the same TEE runtime, thus increasing the memory overhead?

To address this concern, we emphasize that the restricted payload loading aims to *restrict the sharing of the TEE runtime across mutually-distrusting users*. Binding a portable payload with the hardware-backed measurement enables the user to verify that the initial state of the launched enclave is as expected. A user with the need of sharing the same TEE runtime across her multiple payloads could firstly develop a specialized portable payload (acting as her customized library OS or loader) to be built into the enclave, and leverage it to load her other portable payloads after the enclave is launched and attested. On the other hand, framework-level support for such TEE sharing will be left to future work.

6.2. Identity-Measurement Binding

To facilitate the portability in attesting portable TEE applications, portable identity is incorporated into the attestation evidence as a binding element alongside the measurement. As outlined in Sec. 5, it is necessary to realize a portable identity generation function ($\mathcal{P}$) within attester enclaves and an identity measurement verification function ($\mathcal{V}$) within verifier enclaves.

Portable Identity Generation. When the entire portable payload is hardcoded into the built enclave, $\mathcal{P}$ can be realized by applying I to the content of the hardcoded portable payload to generate the corresponding portable identity. When the portable identity is hardcoded into the built enclave, $\mathcal{P}$ could simply extract the hardcoded value, and rely on the enclave to ensure that the loaded payload's identity matches the hardcoded value.

Identity-Measurement Verification. When receiving a pair of portable identity and measurement (I, M) along with the evidence, the verifier enclave needs to verify the integrity of the measurement and the binding. Firstly, following the traditional remote attestation process, the verifier verifies the evidence's trustworthiness by confirming its signature was generated with the enclave's attestation key. This step establishes the authenticity and the secure origin of the enclave's evidence. Afterwards, $\mathcal{V}$ is employed to ascertain the integrity of the measurement-portable identity binding. According to Eq. (6), $\mathcal{V}$ needs to check the existence of a portable payload $\hat{P}$ such that $I = \mathsf{I}(\hat{P}) \wedge M = \mathsf{M}_j(\mathcal{B}_j(R_j, \hat{P}))$.

In the case of hardcoding the entire portable payload, it is challenging for the built verifier enclave to check the existence of a portable payload $\hat{P}$ such that $I = \mathsf{I}(\hat{P})$ and $M = \mathsf{M}_j(\mathcal{B}_j(R_j, \hat{P}))$ without extra knowledge about $\hat{P}$. Potential solutions include (1) the attester enclave provides an additional pointer to its payload located in some public repositories (e.g., Drawbridge[37] for Enarx[17]); (2) the attester enclave provides its payload with the attestation evidence. Besides the communication overhead for obtaining the potential payload, the former requires additional cost for running such a public repository while the latter might raise privacy concerns about disclosing the payload to unwelcome verifiers whose identities haven't been verified by the attester enclave.

In contrast, in the case of hardcoding the portable identity, the verifier enclave needs to implement $\mathcal{V}$ to check the existence of a portable payload $\hat{P}$ such that $I = \mathsf{I}(\hat{P})$ and $M = \mathsf{M}_j(\widetilde{\mathcal{B}_j}(R_j, \mathsf{I}(\hat{P})))$. Note that one could determine the result without knowing the concrete content of $\hat{P}$. Particularly, executed by the verifier enclave $E_i^P \leftarrow \widetilde{\mathcal{B}}_i(R_i, \mathsf{I}(P))$, $\mathcal{V}$ could directly compare the received measurement M against the derived reference measurement using the received portable identity I, that is, $\mathsf{M}_j(\widetilde{\mathcal{B}_j}(R_j, I))$, and output 1 if $M = \mathsf{M}_j(\widetilde{\mathcal{B}_j}(R_j, I))$.

The missing information for the derivation of the reference measurement is the knowledge about the TEE runtime R_j, which will be covered in Sec. 6.3.

Due to the advantages of hardcoding the portable identity over hardcoding the portable payload, we focus on the former for the rest of the paper.

6.3. Layered Reference-Value Derivation

To achieve portability in attesting portable TEE applications, we propose layered mutual attestation, enabling the portable payload and TEE runtime of a verifier enclave to derive reference values for verifying the received portable identity and measurement, respectively.

As illustrated in Sec. 5, the intuition for reference-value derivation functions is to enable the portable payload and runtime to independently derive the reference values from a pre-generated common part. To realize portability, the generation of this common part for portable payloads and runtime requires only their specific parts respectively as input, as outlined in Eq. (3) and Eq. (5).

Reference-Value Derivation for Portable Payload. One straightforward design of ($\mathcal{G}^p$, $\mathcal{F}^p$) is adapted from MAGE [34], which is based on the observation that the calculation process of cryptographic hash functions used in TEEs is usually deterministic and sequential. Therefore, it is possible to derive the final hash value if one knows the intermediate hash state of the prefix portion and the content of the remaining portion to be updated. Particularly, consider a group of L portable payloads $\texttt{PPld}_l$ ($l = 1, \ldots, L$), each of which with portable code P_l (in the form of $P_l = P'_l \| P_{\text{common}}$) would like to derive the reference portable identities of the other portable payloads, ($\mathcal{G}^p$, $\mathcal{F}^p$) can be defined as follows:

$$\begin{aligned} P_{\text{common}} \leftarrow \mathcal{G}^p(P'_1, \ldots, P'_l) &= \left(\overline{I^{P'_1}}, \ldots, \overline{I^{P_l}}'\right) \\ \mathcal{F}^p(P_{\text{common}}, k) &= \mathsf{I}\left(P_{\text{common}}, \overline{I^{P_k}}'\right) \\ &= \mathsf{I}\left(P'_k \| P_{\text{common}}\right) = \mathsf{I}(P_k) \end{aligned} \tag{7}$$

Note that the use of intermediate hash states in MAGE is due to the restriction that the measuring process performed by hardware cannot be modified. The memory overhead includes additional metadata such as the size of the processed message. When it comes to portable identities, we can reduce memory overhead by redefining the definition of portable identity I^P of a portable payload $P = P' \| P_{\text{common}}$ as follows:

$$I^P \leftarrow \mathsf{I}(\mathsf{I}(P') \| P_{\text{common}}) \tag{8}$$

And an alternative design of ($\mathcal{G}^p$, $\mathcal{F}^p$) is as follows:

$$\begin{aligned} P_{\text{common}} \leftarrow \mathcal{G}^p(P'_1, \ldots, P'_l) &= (\mathsf{I}(P'_1), \ldots, \mathsf{I}(P'_l)) \\ \mathcal{F}^p(P_{\text{common}}, k) &= \mathsf{I}\left(\mathsf{I}(P'_k) \| P_{\text{common}}\right) = I^{P_k} \end{aligned} \tag{9}$$

Both of the these two designs are compatible with the other components in LATTE, as we properly decouple the development of portable payloads and TEE runtimes.

Reference-Value Derivation for TEE Runtime. Different from the design of ($\mathcal{G}^p$, $\mathcal{F}^p$), besides the content of the TEE runtime, the final enclave also needs to include

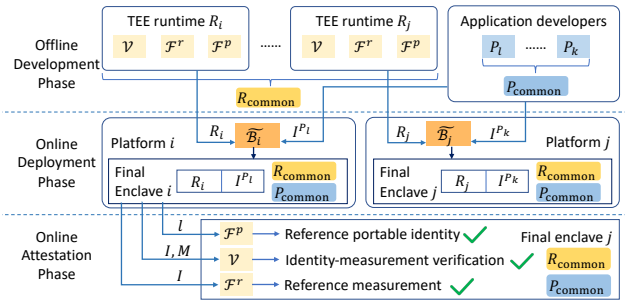

Figure 4: LATTE Workflow Overview.

information about the portable payload, which is irrelevant to the TEE runtime, and thus introduce challenges into the design of the build function and ($\mathcal{G}^r$,$\mathcal{F}^r$). The choice of hardcoding only the portable identity pays off here. It results in the following neat designs. The build function is defined as $\widetilde{\mathcal{B}}_i(R_i, I) = R_i \| I = R'_i \| R_{\text{common}} \| I$ and its measurement can be denoted as:

$$\begin{aligned} \mathsf{M}_i(\widetilde{\mathcal{B}}_i(R_i, I)) &= \mathsf{M}_i(R'_i \| R_{\text{common}} \| I) \\ &= \mathsf{M}_i\left(R_{\text{common}} \| I, \overline{M_i^{R'_i}}\right) \end{aligned}$$

From the above equation, we can measure the specific part of the TEE runtimes of all N platforms first to collect the intermediate hash states $\overline{M_i^{R'_i}}$ in advance, and group them into a list as the common part of the runtime, i.e.,

$$R_{\text{common}} \leftarrow \mathcal{G}^r(R'_1, \ldots, R'_N) = \left(\overline{M_1^{R'_1}}, \ldots, \overline{M_N^{R'_N}}\right)$$

Subsequently, by retrieving the corresponding intermediate hash state for a given platform j from R_{common}, the remaining part of the hash calculation involves only the common runtime content, readily accessible to the verifier since it is identical to that of the attester enclave, and the portable identity, which is part of the verifier's received binding. Thus, the verifier is able to derive the corresponding measurement of the portable identity effectively, using $\mathcal{F}^r$ defined as follows,

$$\begin{aligned} \mathcal{F}^r(R_{\text{common}}, j, I) &= \mathsf{M}_j\left(R_{\text{common}} \| I, \overline{M_j^{R'_j}}\right) \\ &= \mathsf{M}_j\left({R_j}' \| R_{\text{common}} \| I\right) \\ &= \mathsf{M}_j(\widetilde{\mathcal{B}_j}(R_j, I)) \end{aligned}$$

6.4. LATTE Workflow

By integrating all the aforementioned techniques, we now establish the workflow of LATTE, as depicted in Fig. 4. The workflow includes three phases as follows:

Offline Development Phase. The initial phase is the offline development phase. TEE runtimes with the corresponding build functions $\widetilde{\mathcal{B}}_i$, are developed publicly, potentially as open-source projects, allowing users to assess their security. It is important to note that the identity-measurement verification function $\mathcal{V}$ and the reference measurement derivation function $\mathcal{F}^r$ are incorporated into the runtime as a library. Then, the common part R_{common} can be constructed from the runtime of all potential TEE

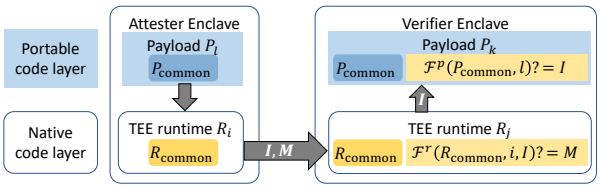

Figure 5: The Online Attestation Phase.

platforms using $\mathcal{G}^r$, and concatenated with each runtime. On the other hand, application developers develop TEE applications separately. Once the portable TEE application developer completes the development of a portable TEE application consisting of L portable payloads, she could integrate the reference portable identity derivation function $\mathcal{F}^p$ into these portable payloads, build the common part using $\mathcal{G}^p$ and generate the final portable payloads.

Deployment Phase. The second phase is the deployment phase, which might differ considerably depending on concrete application scenarios. In the case of confidential serverless computing [19], [20], [21], [22], upon receiving deployment requests and portable payloads, the confidential serverless computing platform determines the optimal server allocation for each payload based on current availability. On the chosen servers, the corresponding build function $\widetilde{\mathcal{B}}$ is used to hardcode the portable identities of the requested portable payloads into the final enclaves, along with the corresponding TEE runtime. Following this, each enclave is launched and initiated with a portable payload; if the portable identity calculated from the input payload matches the hardcoded value, the runtime proceeds to load and execute the payload.

Online Attestation Phase. The third phase is the online attestation phase, as shown in Fig. 5. When one final enclave $E_i^{P_l}$ with portable payload P_l on platform i, intends to attest itself to another enclave $E_j^{P_k}$ with payload P_k on platform j, it employs the portable identity generation function $\mathcal{P}$, sending its portable identity $\mathsf{I}(P_l)$ with other attestation evidence generated by the conventional remote attestation process, e.g., quote, to the verifier $E_j^{P_k}$. Upon receipt, $E_j^{P_k}$ utilizes the identity-measurement verification function $\mathcal{V}$ to verify the integrity of the received binding. This process is handled by the runtime with $\mathcal{F}^r$ to derive the reference measurement from its runtime's common part. Following successful integrity verification, the received portable identity is forwarded to the portable payload P_k to ensure that it corresponds to one of the portable payloads within the application. The payload then employs $\mathcal{F}^p$ to derive the reference portable identity from its payload's common part. Once the derived portable identity matches the received one, the attestation process is considered successfully completed.

6.5. Analysis of LATTE

Now we analyze the security and portability of LATTE. For security, we explore potential vulnerabilities, attacks, and the trusted computing base (TCB) of LATTE. For portability, we discuss LATTE's role in decoupling the development of portable payloads and runtimes.

Security Analysis. We first analyze how malicious payloads could affect LATTE. Due to the restricted payload loading, the executed payload will be bound to the hardware-backed measurement. To launch an enclave that could load a malicious payload, the malicious payload's identity will be built into the enclave, and reflected by the resulting measurement, which will not match the reference measurement derived by target enclaves executing benign payloads, thus mitigating the attack described in Sec. 3.1. Moreover, to thwart attacks targeting the hash functions, LATTE can adopt a secure hash function, such as SHA-256, for portable identities. This selection allows for defending against preimage attacks and second-preimage attacks. For length extension attacks, which is primarily used to calculate a message containing a secret without prior knowledge of the secret, they hold no utility within the context of LATTE.

We then discuss the TCB of LATTE. Note that the TCB of nested attestation solutions (as discussed in Sec. 3.1) includes the software-based compartmentalisation for isolating the runtime from the payload. These isolation techniques usually involve instrumenting payloads to prevent access to beyond its current security layer, resulting in a *linearly* increased code size, e.g., ranging from 14% to 32% reported by Zhao *et al.* [22]. As a comparison, LATTE removes the need of such software-based isolation from the TCB. On the other hand, the increased TCB includes the implementation of the identity-measurement binding mechanism and reference-value derivation mechanism, resulting in a *fixed* increased code size, e.g., 993 lines of code in our prototype implementation.

Portability Analysis. In LATTE, both the development and verification of the runtime and portable payload are fully decoupled. During the development phase, two common part generation functions, $\mathcal{G}^r$ and $\mathcal{G}^p$, process the specific components pertaining to the runtime and portable payload, respectively. This separation enables developers of the runtime and those working on portable TEE applications to independently complete and finalize their work without having to take into account each other's progress. While in the online attestation phase, the verification of runtime measurements uses inputs solely from the common part of the runtime. Similarly, the verification of the portable identity depends entirely on inputs from the common part of the portable payload. This decoupling significantly enhances the system's modularity, enabling more straightforward updates and independent scalability.

We then look into the portability issues associated with updates, as discussed in Sec. 3.2, for concrete analysis. When an update is required for a runtime on any platform, the changes are confined to the common part of the runtimes, meaning that no changes are required for the payloads due to the well-designed verification process employed by LATTE. On the other hand, updating a portable payload does not require recompilation for runtimes. The only requirement is to regenerate the common part of payloads and rebuild the associated enclaves. Therefore, LATTE successfully adheres to the initial objective of facilitating portable development, allowing independent updates for runtimes and payloads.

7. Implementation

In this section, we present our prototype implementation as a demonstration of how to instantiate LATTE for real heterogeneous TEEs. We select SGX and Penglai as exemplar heterogeneous TEEs, WASM as the portable language for developing portable TEE applications, and the WAMR as the underlying runtime.

Our implementation is based on Intel SGX SDK (version 2.23), Penglai TVM (commit $bf52983$), and WAMR (commit $72b34ea$). While WAMR already supports executing WASM payloads within SGX enclaves, Penglai is not supported, and consequently, we have adapted it to Penglai platform. Furthermore, we have developed a standalone library, named `lib_latte`, which encompasses implementations for measurement derivation and includes the necessary utility tools for LATTE. This library features unified API specifications for heterogeneous TEE platforms to facilitate layered attestation within LATTE, and we provide the corresponding implementation within WAMR for both SGX and Penglai platforms.

For portable identities, we adopt SHA-256 [55] as its cryptographic hash algorithm, and the latter derivation design discussed in Sec. 6.3 (Eq. (8) and Eq. (9)). SHA-256 is one of the most secure hash functions nowadays. Thus, the generated portable identities can identify the portable codes while successfully protecting their integrity.

Next, we will first discuss the necessary modification of SGX and Penglai SDKs and detailed implementation of mechanisms in `lib_latte`, adhering to the sequence outlined in Sec. 6. Then, we will present the APIs in `lib_latte` to support layered attestation.

7.1. Restricted Payload Loading

Given the benefits outlined in Section 6, we have opted to hardcode the portable identity to facilitate the restricted payload loading. This section discusses our techniques for reserving specific sections for portable identities on SGX and Penglai platforms, and it describes how we implement the build functions to hardcode portable identities into these sections to produce the final enclaves.

For SGX, we provide an SDK library `libsgx_wasm`, which reserves a read-only section called `.sgx_wasm` for the portable identity. For the process of hardcoding the portable identity into the `.sgx_wasm` section, we extended the signing tool of the Intel SGX SDK with an additional mode `SIGN_WASM`, in which the modified signing tool hardcodes the input portable identity into the `.sgx_wasm` section before signing the enclave.

For Penglai, we modified the enclave driver in Penglai SDK to reserve a `.penglai_wasm` section at a predefined high address `0xFFFFFFF000000000` (the reason for choosing this address will be detailed in Sec. 7.3). Subsequent to this reservation, the portable identity input is copied directly to the designated page within the enclave for the purpose of hardcoding.

In conjunction with these platform-specific modifications, we provide APIs in both SDKs, `self_portable_identity()`, allowing for direct access to these reserved sections. This enables runtimes to compare the calculated portable identity of a given portable payload and the hardcoded value for verification before the loading and execution of the given portable payload.

7.2. Identity-Measurement Binding

In LATTE, the identity-measurement binding is transmitted alongside the conventional evidence to facilitate layered attestation for portable TEE applications. For an attester enclave, it is necessary to generate its own portable identity using the function $\mathcal{P}$ to produce the binding. On the other hand, a verifier enclave is required to assess the integrity of the received identity-measurement bindings using the function $\mathcal{V}$.

Portable Identity Generation. Portable identities are hardcoded into designated reserved sections, with APIs available in both the SGX and Penglai SDKs. These APIs can be directly invoked to retrieve the corresponding portable identities of attester enclaves.

Identity-Measurement Verification. As outlined in Sec. 6.2, the verification of an identity-measurement binding involves two primary steps. The first is to determine the authenticity of the received evidence by checking its signature, akin to the procedure used in conventional remote attestation. The second step involves using the reference measurement derivation function for TEE runtime to directly derive the reference measurement for the given portable identity and then comparing it with the received measurement, which is straightforward. Further details of the derivation function will be discussed subsequently.

7.3. Layered Reference-Value Derivation

As illustrated in Sec. 6.3, the reference-value derivation functions are utilized to enable portable payload and runtime to independently derive the reference values from a pre-generated common part, thereby allowing the verifier enclave to assess the trustworthiness of the received portable identities and measurements. To implement this derivation mechanism and ensure the derived result aligns with the correct measurement or portable identity values, three key tasks are highlighted: (1) generating the common part, (2) concatenating this common part *after* the specific part in the hash calculation process, and (3) simulating the hash calculation process based on intermediate hash states extracted from the corresponding common part. The description of how these tasks are achieved for both the portable payload and runtime is as follows.

Reference-Value Derivation for Portable Payload. To generate the common part of a set of WASM programs, we provided a tool named `insert-wasm-latte` in our library `lib_latte`, integrated with a crypto library modified from OpenSSL [56]. This tool calculates the SHA-256 hash for each WASM file, groups these hashes into a list which then forms the common part, and encodes this list into an additional custom section named `portid` for all input WASMs.

For the second task, `insert-wasm-latte` places this section at the end of the raw WASM files to form the final WASM file, ensuring that it becomes the final material included in the portable identity update process. It should be noted that the size of this section does not

require alignment to 64B for SHA-256, as padding will be handled separately.

Regarding the third task, a specialized parsing function, `parse_portid_section()` has been developed within `lib_latte` that utilizes an index to extract the SHA-256 hash of a specified WASM component within the application. This is further invoked by another function in `lib_latte`, `derive_portable_identity()`, which continues the hash calculations on the indexed SHA-256 hash using the integrated crypto library.

Reference-Value Derivation for TEE Runtime. To generate the common part of runtimes, it is necessary to collect the intermediate hash states from both SGX and Penglai runtimes. For SGX, we extended the signing tool of the Intel SGX SDK by introducing one more mode `GEN_WASM_VM_MR`. With this mode, the intermediate hash value, the total number of blocks processed right before loading `.sgx_wasm`, with the offset of `.sgx_wasm`, all of which constitute the intermediate hash state of SHA-256, can be collected. For Penglai, we customized the implementation of `PLenclave_attest()` (the official API of Penglai SDK to get the enclave measurement and generate the enclave report for host program) such that when the portable identity section address (`0xFFFFFFF000000000`) is encountered, the intermediate hash value and the total number of bytes processed are output.

Once all intermediate hash states are collected, they are hardcoded into the TEE runtime and positioned at the end of the measurement calculation process to ensure accurate derivation.

For SGX, we streamlined the process by merging the hardcoding of the intermediate hash states with portable identities. The signing tool's extended mode, `SIGN_WASM`, takes two inputs: the calculated portable identity and a file containing intermediate hash states from both SGX and Penglai runtimes. The section to be hardcoded, `.sgx_wasm` is reserved with 8 KB, i.e. two pages, in our existing implementation. The signing tool first hardcodes the portable identity into the first page and the intermediate hash states into the second. This sequence is reversed compared to what we described in Sec. 5; however, since both components are known to the verifier, it does not affect their ability to derive the final measurement. If more intermediate states are required that exceed one page, the settings can be easily adjusted to accommodate the increased demand. Additionally, we must adjust the measurement calculation order to facilitate the correct derivation. On SGX, the enclave measurement is updated during the enclave creation, in which the enclave pages are created and initialized by loading from the enclave binary file. Therefore, we modified the enclave loader in Intel SGX SDK such that `.sgx_wasm` section is loaded at last, following the method introduced in MAGE [34]. In more detail, when a `.sgx_wasm` section is found during enclave creation, the modified loader would skip it in the original loading process and load the pages in the section in the end, just before the enclave initialization.

For Penglai, a similar approach is adopted: the portable identity and intermediate hash states are hardcoded into two consecutive pages at the address `0xFFFFFFF000000000`. Given that Penglai measurement is calculated by traversing the enclave memory via the page table from the lowest address to the highest address, data located at higher addresses is incorporated later in the measurement update sequence. We have checked that `0xFFFFFFF000000000` is the highest in the default memory layout of the enclave in Penglai SBI. Consequently, both the portable identity and the common part are included last into the measurement calculation, ensuring accurate derivation results.

For the third step, which involves completing the remaining hash calculations, two APIs in `lib_latte`, `sgx_derive_measurement()` and `penglai_derive_measurement()`, are straightforwardly implemented for that. The former utilizes the SHA-256 algorithm for SGX measurements, while the latter employs the SM3 hash algorithm for Penglai measurements.

7.4. `lib_latte_wamr` Attestation APIs

With the aforementioned modifications in SGX SDK and Penglai SDK/SBI, we specify two unified APIs in `lib_latte` to facilitate the entire layered attestation process in LATTE, and implement them within WAMR for both SGX and Penglai.

`latte_attest()`. When one portable payload needs to attest itself, it can invoke this API to generate its identity-measurement binding. This information with the platform and portable payload index, is then sent to the intended verifier. It is implemented directly by leveraging `self_portable_identity()` to acquire its portable identity, and uses the conventional attestation APIs in the SGX and Penglai SDK to obtain its measurement.

`latte_verify()`. For verifier enclaves, this API is provided for creating a socket for incoming attesting requests. When there is one request received, it invokes `sgx_derive_measurement()` or `penglai_derive_measurement()` with the received portable identity to validate the integrity of the received measurement. Then, it calls `derive_portable_identity()` to determine if this portable identity matches the specified indexed portable payload within the application through comparison.

8. Evaluation

In this section, we present the evaluation results of our prototype implementation. Particularly, we conduct a comparison of `lib_latte` with the nested attestation solution adopted by WAMR, in Sec. 8.1, and demonstrate the application of LATTE in a heterogeneous attestation scenario in Sec. 8.2.

Experimental setup. All the evaluations are performed on a Lenovo Thinkpad X1 Carbon (8-th Gen) laptop with a 4-core Intel CPU i5-10210U and 16 GB memory. The host OS is Ubuntu 20.04.5, with Linux kernel 5.10. Note that all experiments related to Penglai enclave are booted by penglai-qemu's commit $39ddb39$.

8.1. Performance Evaluation

Since our prototype is based on WAMR, which adopts a nested attestation solution (in the form of a

TABLE 2: LATTE Performance Evaluation.

	Size	WAMR-librats	LATTE-lib_latte	Overhead (G.M.)
Start-up	200 KB	95.7 ms	96.2 ms	0.57%
	400 KB	98.0 ms	98.5 ms	0.49%
	600 KB	100.1 ms	100.6 ms	0.48%
	800 KB	102.3 ms	102.6 ms	0.38%
	1000 KB	104.8 ms	105.2 ms	0.39%
Evidence generation		119.3 ns	123.6 ns	3.6%

library `librats`) for attesting WASM payloads on Intel SGX, we compare the performance of `lib_latte` with `librats` on Intel SGX.

Overhead of Start-up Latency. We first evaluate the start-up latency due to restricted payload loading. While the start-up process for WAMR-`librats` involves launching an enclave with the WASM runtime which further loads a given WASM payload and calculates its hash, the start-up process for LATTE-`lib_latte` additionally requires comparing the calculated hash with the hardcoded reference value. We measured the start-up latency by loading a series of WASMs ranging from 200 KB to 1000 KB in size, averaging the results over 1000 iterations and documenting the outcomes in Table 2. It is noteworthy that WAMR supports multiple WASM modules within a single runtime, and WAMR-`librats` employs a mutex lock to synchronize hash calculations across these modules. However, to align with LATTE's current single-module-per-runtime design, we removed this mutex from WAMR-`librats` in our evaluation, ensuring an equivalent comparison baseline. As shown in Table 2, the additional overhead induced by start-up latency in our system ranges from 0.38% to 0.57%, with a geometric mean of 0.456%, which is minimal. This slight increase is attributed primarily to the extra steps involved in retrieving the reference measurement and conducting validation checks; these are relatively inexpensive compared to hash calculation. Moreover, since the number of these extra steps is fixed, the overhead percentage actually decreases (from 0.57% to 0.39%) as the size of the WASM payload increases.

Overhead of Evidence Generation. We measure latency in generating attestation evidence within `latte_attest()`, averaging results from 10000 iterations. To produce the attestation evidence, `librats` inserts the calculated hash of the payload into the generated report, while `lib_latte` extracts the hardcoded portable identity and attaches it to the generated report. Notably, similar to our previous evaluation settings, we have eliminated the unnecessary mutex lock from `librats`, alongside other inessential configurations, for a cleaner and fairer comparison between the two libraries. The overhead of evidence generation latency is presented in Table 2. The results show quite a small difference in latency between the two libraries, varying by a mere 4.3 ns.

Reference-Value Derivation Efficiency. Since `librats` does not support mutual attestation, we only measure the efficiency of the reference-value derivation mechanism introduced by LATTE. As described in Sec. 7.3, the time cost of deriving a reference measurement and a reference portable identity is highly related to the size of the TEE runtime's reserved section, and `portid` section in WASM, respectively. For the base case with a `.sgx_wasm`/`.penglai_wasm` section of 8KB (supporting up to 85 TEE runtimes) and a `portid` section of 4KB (supporting up to 126 portable payloads), it takes $41.8/68.0$ μs to derive a reference measurement, and 16.2 μs to derive a reference portable identity, respectively. We also evaluated the efficiency of reference-value derivation with regards to various sizes of these two sections, respectively. The results, presented in Fig. 6, demonstrate that the time consumption increases linearly with the size of the sections. This outcome is anticipated since the hash calculation process is sequential; larger section size leads to more data for remaining hash calculations from the intermediate state, thus increasing derivation time.

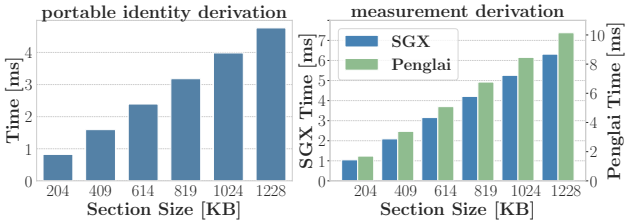

Figure 6: Time consumption for reference-value derivation.

8.2. Case Study: Genann

We now present a case study integrating LATTE into Genann [57], a minimal, well-tested open-source library for feedforward artificial neural networks (ANN) written in C. We first divide a Genann program into two distinct components: a trainer and a runner. The trainer is responsible for model training using training data, whereas the runner serves to run a model with private data for inference. This way ensures the provision of the model to only trusted parties while safeguarding the data privacy of both involved parties. The key lies in the necessity for mutual attestation between the trainer and the runner when the latter requests the well-trained model from the former and the former needs to attest the latter's identity for releasing the model.

To accomplish this, we compiled both the trainer and the runner to WASM, incorporated `lib_latte` attestation APIs, and executed them using WAMR integrated with LATTE. Our tests involved running two portable payloads (i.e., the trainer and runner) on two potential heterogeneous TEE platforms (i.e., SGX and Penglai), resulting in four scenarios: SGX-Penglai (indicating running the trainer on SGX and the runner on Penglai), Penglai-SGX, SGX-SGX, and Penglai-Penglai. In all cases, the identity and measurements of both the trainer and the runner were successfully derived and verified through the layered approach of LATTE, which facilitated the secure transmission of the model to the runner.

9. Discussion

Extensions to Other TEEs. While in this paper, LATTE is instantiated on SGX and Penglai, extending it to VM-

based TEEs like AMD SEV and Intel TDX is feasible: (1) For restricted payload loading, we can reserve the section and perform hardcoding for portable identity straightforwardly using C macros and `objcopy`. Furthermore, we can incorporate the runtime code as part of the VM image kernel so that it is included in the measurement calculation. (2) For identity-measurement binding, portable identity can be obtained directly by reading the reserved section. The identity-measurement verification can leverage traditional TEE attestation methods alongside the runtime's reference-value derivation mechanism (detailed below). (3) For layered reference-value derivation, the portable payload level is unrelated to the underlying TEE platform. The derivation for TEE runtime is the most challenging part. In AMD SEV and Intel TDX, measurement calculation follows a process similar to SGX: it updates the digest incrementally as pages are added to guest memory via platform-specific calls (`SNP_LAUNCH_UPDATE` for AMD SEV; `SEAMCALL[TDH.MEM.PAGE.ADD]` and `SEAMCALL[TDH.MEM.PAGE.EXTEND]` for Intel TDX). This allows for adjustments in the loading order, ensuring that the reserved section is added last. By manually performing the update process, the intermediate hash before this section can be precomputed. Once the intermediate hash states are obtained and the loading order is adjusted, the final measurement can be derived using the corresponding hash algorithms.

In conclusion, two attestation APIs in `lib_latte` can be implemented without too much effort after these modifications, enabling the extension.

Limitations on Enclave Updates. Similar to MAGE, the use of hardcoded common parts restricts the ability to perform online enclave updates. With MAGE, if the content of any enclave is modified, all enclaves must be rebuilt by developers to reflect the change. In contrast, LATTE 's decoupled design ensures that updating a portable payload requires regenerating payloads' common part, while runtimes avoid regenerating their own common parts. Runtime developers are not required to modify their runtimes; instead, the remaining steps are lightweight, limited to rebuilding associated enclaves with new portable identities and redeploying the updated payloads. This process could be managed by the cloud platform in the case of confidential serverless computing. Further, updating any TEE runtime involves regenerating the runtimes' common parts, redeploying the *same* portable payloads and rebuilding associated enclaves. This means that developers of portable payloads do not need to take any action during TEE runtime updates, as their payloads remain unchanged.

10. Related Work

Supporting WASM on TEE. There have been existing works on the confidential runtime to host portable applications developed in high-level language inside TEEs, and WASM is a popular choice as the portable language/IR. For example, TWINE [14] and WATZ [15] are two trusted WASM runtimes on Intel SGX and ARM TrustZone, respectively. Both of them rely on the open-source project, WAMR [16], which is also a foundation for our work. Enarx [17] is another open-source confidential computing framework for running WASM applications on two TEE platforms, i.e., Intel SGX and AMD SEV.

While all the above works focus on the problem of supporting executing WASM in enclaves, our work, LATTE, concentrates on an orthogonal problem of supporting attesting portable TEE applications in a both secure and portable way.

The work that is most related to ours is the software-based nested attestation scheme in WAMR [24]. It currently only supports SGX but has the potentials to be extended to other TEE platforms. However, as mentioned in Sec. 3.1. software-based nested attestation faces the security issue that we aim to address with LATTE.

Cross-TEE Portability and Attestation. To alleviate the deployment difficulties caused by the tight binding between TEE applications and platforms, numerous efforts have been made to detach TEE application development from the underlying hardware and support the execution of one application on heterogeneous TEE platforms. For instance, vSGX [58] provides support for the execution of SGX legacy applications on AMD SEV via virtualization, while Google's Asylo [59] and Microsoft's Open Enclave [60] aim at using a unified abstract enclave model which can be mapped to various TEE backends, and thus all applications developed with their SDKs can be compiled and executed on heterogeneous TEE hardware or even software backends. Additionally, Enarx [17] introduces WebAssembly as the portable language and provides a WebAssembly runtime with the support of Intel SGX and AMD SEV. Furthermore, Komodo [61] intends to delegate some core TEE hardware mechanisms to software, like memory encryption. It presents a framework on ARM TrustZone to achieve comparable security to SGX with a formal-verified software monitor.

The above works address the problem of portable software execution on heterogeneous TEEs. For the attestation problem among heterogeneous enclaves, MAGE [34] enables mutual attestation among enclaves without a trusted third party. We extended it into a layered attestation approach that ensures security while maintaining portability with a focus on *portable* TEE application attestation.

11. Conclusion

In this paper, we introduced the concept of portable identities for identifying the same portable payloads running on heterogeneous TEE platforms. Our presented LATTE, including three key mechanisms for layered attestation, achieves both security and portability to attest portable TEE applications. To demonstrate its practicality, we made a prototype implementation by adopting WASM as the portable IR, WAMR as the TEE runtime, and Intel SGX and RISC-V Penglai as representative heterogeneous TEEs. We have also shown that the performance of LATTE is reasonable with detailed evaluation.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. The work was partially supported by the National Natural Science Foundation of China under Grant No. 62472281, 62325207, and U24A20241.

References

[1] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 38–54.

[2] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "SGX-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1211–1228.

[3] M. Al-Bassam, A. Sonnino, M. Król, and I. Psaras, "Airtnt: Fair exchange payment for outsourced secure enclave computations," *arXiv preprint arXiv:1805.06411*, 2018.

[4] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, "Privacyguard: Enforcing private data usage control with blockchain and attested off-chain contract execution," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 610–629.

[5] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.

[6] G. Ayoade, V. Karande, L. Khan, and K. Hamlen, "Decentralized iot data management using blockchain and trusted execution environment," in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 2018, pp. 15–22.

[7] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious {Multi-Party} machine learning on trusted processors," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 619–636.

[8] Intel, "Intel Software Guard Extensions (Intel SGX) Services," https://api.portal.trustedservices.intel.com/, 2018.

[9] Intel, "Intel Trust Domain Extensions (Intel TDX)," https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html, 2023.

[10] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White paper, 2016, https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[11] Arm, "Arm Confidential Compute Architecture(Arm CCA)," 2021. [Online]. Available: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[12] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the {PENGLAI} enclave," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 275–294.

[13] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[14] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for webassembly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 205–216.

[15] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Watz: a trusted webassembly runtime environment with remote attestation for trustzone," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 1177–1189.

[16] B. Alliance, "WebAssembly Micro Runtime," 2023. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[17] Enarx, "Enarx," 2023, accessed on 16/1/2023. [Online]. Available: https://enarx.dev/

[18] W. C. Group, "Webassembly," 2022, accessed on 13/1/2023. [Online]. Available: https://webassembly.github.io/spec/core/intro/introduction.html

[19] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel SGX," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 185–199.

[20] W. Qiang, Z. Dong, and H. Jin, "Se-lambda: Securing privacy-sensitive serverless applications using SGX enclave," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2018, pp. 451–470.

[21] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "Acctee: A webassembly-based two-way sandbox for trusted resource accounting," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 123–135.

[22] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin, "Reusable enclaves for confidential serverless computing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4015–4032.

[23] T. C. Group, "Attestation architecture — trusted computing group," accessed on 05/2/2023. [Online]. Available: https://trustedcomputinggroup.org/resource/dice-attestation-architecture/

[24] B. Alliance, "WebAssembly Micro Runtime Improved Attestation Scheme," 2023, accessed on 17/1/2023. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1695

[25] Enarx, "Enarx," 2024, accessed on 09/4/2024. [Online]. Available: https://hackmd.io/@enarx/rJ55urrvo

[26] "CVE-2023-26489." Available from MITRE, CVE-ID CVE-2023-26489., Feb. 23 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26489

[27] V. Costan and S. Devadas, "Intel SGX explained," *Cryptology ePrint Archive*, 2016.

[28] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.

[29] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: EPID provisioning and attestation services," *White Paper*, vol. 1, no. 1-10, p. 119, 2016.

[30] X. Wang and H. Yu, "SM3 cryptographic hash algorithm," *Journal of Information Security Research*, vol. 11, pp. 983–994, 2016.

[31] the International Organization for Standardization and the International Electrotechnical Commission, "IT Security Techniques — Hash-functions — Part 3: Dedicated hash-functions," https://www.iso.org/standard/67116.html, 2019.

[32] X. Zheng, X. Hu, J. Zhang, J. Yang, S. Cai, and X. Xiong, "An efficient and low-power design of the SM3 hash algorithm for IoT," *Electronics*, vol. 8, no. 9, p. 1033, 2019.

[33] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "RFC 9334: Remote ATtestation procedureS (RATS) Architecture," 2023.

[34] G. Chen and Y. Zhang, "{MAGE}: Mutual attestation for a group of enclaves without trusted third parties," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4095–4110.

[35] Wasmer Community Group, "Wasmer," 2023. [Online]. Available: https://github.com/wasmerio/wasmer

[36] Bytecode Alliance, "wasmtime," 2023. [Online]. Available: https://github.com/bytecodealliance/wasmtime

[37] Enarx, "Enarx drawbridge," 2024, accessed on 21/4/2024. [Online]. Available: https://github.com/enarx/drawbridge

[38] "CVE-2023-2136." Available from MITRE, CVE-ID CVE-2023-2136., 17 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-2136

[39] "CVE-2023-26919." Available from MITRE, CVE-ID CVE-2023-26919., Feb. 27 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26919

[40] "CVE-2023-32314." Available from MITRE, CVE-ID CVE-2023-32314., May 08 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-32314

[41] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[42] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[43] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[44] M. Hähnel, W. Cui, and M. Peinado, "{High-Resolution} side channels for untrusted operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 299–312.

[45] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[46] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 557–574.

[47] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1041–1056.

[48] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 640–656.

[49] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 317–328.

[50] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[51] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGXpectre: Stealing intel secrets from SGX enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[52] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[53] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[54] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-Resistant CPUs." New York, NY, USA: Association for Computing Machinery, 2019, p. 769–784.

[55] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Secure hash standard (shs)," *Federal Information Processing Standards Publications*, Aug 2015.

[56] OpenSSL, "OpenSSL," 2024, accessed on 27/4/2024. [Online]. Available: https://github.com/openssl/openssl

[57] Lewis Van Winkle, "Genann," 2020. [Online]. Available: https://github.com/codeplea/genann

[58] S. Zhao, M. Li, Y. Zhangyz, and Z. Lin, "vSGX: Virtualizing SGX enclaves on AMD SEV," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 321–336.

[59] Google, "Asylo," 2022, accessed on 29/1/2023. [Online]. Available: https://asylo.dev/

[60] O. Enclave, "Open Enclave SDK," 2023, accessed on 29/1/2023. [Online]. Available: https://openenclave.io/sdk/

[61] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 287–305.

Appendix A.
Cryptographic Hash Function

A cryptographic hash function, represented as $\mathcal{H}$, is a mathematical function used to map data of arbitrary size (usually called *message*) to a value with fixed size (usually called *digest*). The function must be deterministic, meaning that it will always generate the same output with the same input. Besides, when it is used as a cryptography tool, it is desired to be preimage resistant, which means that given an output, it should be computationally infeasible to get its original input.

More precisely, the hash function computation runs by the following steps. First, the inputs are padded to several ordered fix-length blocks as $I^{(1)} \| \cdots \| I^{(N)}$. Then the hash function operates in three steps, the initialization step $\mathcal{H}_{init}$, the update step $\mathcal{H}_{upd}$, and the finalization step $\mathcal{H}_{fin}$. Each step maintains a hash state H representing necessary information for generating the final digest, including the intermediate hash value and processed block count.

The hash initialization starts from a fixed initial hash state $H^{(0)}$:

$$H^{(0)} = \mathcal{H}_{init}()$$

In the hash update stage, the hash update function $\mathcal{H}_{upd}$ takes as input the previous hash state $H^{(i-1)}$ and a single block of input $I^{(i)}$, and outputs the updated hash state $H^{(i)}$. The process repeats multiple times until no input block is left:

$$H^{(i)} = \mathcal{H}_{upd}(I^{(i)}, H^{(i-1)}),\ 0 < i \leq N$$

The hash finalization takes as input the last hash state $H^{(N)}$ from the hash update $\mathcal{H}_{upd}$, and outputs the final hash digest. Therefore, the hash function workflow is as follows:

$$\begin{aligned}\mathcal{H}(I) =& \mathcal{H}_{fin}(H^{(N)}) \\ =& \mathcal{H}_{fin}(\mathcal{H}_{upd}(I^{(N)}, \mathcal{H}_{upd}(\cdots \mathcal{H}_{upd}(I^{(1)}, \mathcal{H}_{init}()))))\end{aligned}$$

Since the hash calculation process is sequential, knowing the intermediate hash state and the remaining unprocessed blocks is enough to derive the final hash digest. For convenience, we abuse the notation of $\mathcal{H}$ to represent two cases, i.e., calculating from the start or from an intermediate state $H_{pre}^{(L_{pre})}$, as follows:

$$\begin{aligned}&\mathcal{H}(I_{remain}, H_{pre}^{(L_{pre})}) \\ =& \mathcal{H}_{fin}(\mathcal{H}_{upd}(I^{(N)}, \mathcal{H}_{upd}(\cdots \mathcal{H}_{upd}(I^{(L_{pre}+1)}, H_{pre}^{(L_{pre})})))) \\ =& \mathcal{H}(I)\end{aligned}$$

where $I_{remain} = I^{(L_{pre}+1)} \| \cdots \| I^{(N)}$, $H_{pre}^{(L_{pre})} = \mathcal{H}_{upd}(I^{(L_{pre})}, \mathcal{H}_{upd}(\cdots \mathcal{H}_{upd}(I^{(1)}, \mathcal{H}_{init}())))$.

Appendix B.
Data Availability

Our implementation has been open-sourced at https://github.com/Jiax-cn/latte, which can be reproduced on machines with SGX support. Penglai-related implementations and experiments can be conducted using QEMU and the corresponding Penglai toolchain.