EXIA: Trusted Transitions for Enclaves via External-Input Attestation

Zhen Huang[*], Yidi Kao[†], Sanchuan Chen[†], Guoxing Chen[*✉], Yan Meng[*], Haojin Zhu[*]
[*]Shanghai Jiao Tong University, {xmhuangzhen, guoxingchen, yan_meng, zhu-hj}@sjtu.edu.cn
[†]Auburn University, {yzk0078, schen}@auburn.edu

***Abstract*—Trusted Execution Environment (TEE) has been adopted to secure computation outsourced to untrusted clouds, and the associated remote attestation mechanism enables the user to verify the integrity of the outsourced computation at launch time. However, memory corruption attacks break TEE's security guarantees without being detected after launch-time attestation. While control-flow attestation (CFA) schemes aim to detect runtime compromises, most existing CFA schemes lack concrete verification methods and can be bypassed by data-only attacks. In this paper, we propose the concept of External-Input Attestation to attest all writes to TEE-protected applications, based on the observation that memory corruption attacks typically start with unintended writes. This approach ensures a trusted enclave state by verifying all writes match expectations, transforming security issues, such as control-flow hijacking, into reliability issues, such as a software crash due to unexpected input. For efficient reference measurement derivation and verification, the current version of External-Input Attestation is limited to enclaved applications whose inputs are known to the verifier. This design is validated by implementing and evaluating prototypes on AMD SEV-SNP and Penglai, where security and performance evaluations show a minimal performance overhead in case studies, including secure model training, model inference, database workloads, and key management.**

I. INTRODUCTION

Outsourcing computation to the cloud is becoming increasingly prevalent as the need for more intense computation, such as artificial intelligence (AI) model inference, grows rapidly [91], [75]. Meanwhile, to alleviate concerns about the confidentiality and integrity of outsourced computation to honest-but-curious or even potentially malicious clouds, a viable solution is to adopt the Trusted Execution Environment (TEE), which protects sensitive code and data within a shielded environment, usually called an *enclave*, against untrusted privileged software such as the operating system (OS), hypervisor, or even rogue administrator.

To prove that the outsourced computation is indeed protected by an enclave running on a cloud server, a mechanism called remote attestation is usually introduced along with TEE. Particularly, remote attestation uses a cryptographically signed *measurement* (*i.e.*, the cryptographic hash) of the *initial* code and data loaded into an enclave as *evidence* of the integrity of the outsourced computation at *launch* time.

However, such launch-time attestation alone is inadequate to ensure the security of the whole process of the outsourced computation, as demonstrated by various memory corruption attacks, *e.g.*, control-flow hijacking attacks [77], [23] and data-only attacks [30], [50], that break the integrity and confidentiality guarantees. To mitigate memory corruption attacks during runtime, various control-flow attestation (CFA) [10], [39] and data-flow attestation (DFA) [37], [94], [74] schemes have been proposed to collect information about control flows and data flows during runtime and assume a remote verifier could analyze the collected information to detect attacks. These two approaches have some limitations. Specifically, CFA frameworks require multiple program execution traces to learn the correct control-flow patterns, and the granularity of this data collection presents a trade-off between verification accuracy and storage overhead. Meanwhile, DFA frameworks typically need specialized program analysis tools or modifications to the program itself, and their results are often difficult to verify independently.

As pointed out by Ammar *et al.* [19], all control-flow hijacking and data-only attacks start with an unintended write, *i.e.*, the unexpected (to the user) content provided by the host to enter the enclave. While existing CFA/DFA schemes focus on whether the control/data flow transits to unexpected states, *little attention has been paid to detecting potential attacks by checking at an earlier stage whether the external content accessed by the enclave has been manipulated.*

In this paper, we propose the concept of *External-Input Attestation* to attest all writes to the enclaved application from its external world for attack detection. Particularly, a measurement dubbed *External-Input Measurement (EIM)* is maintained: each time the host passes an input to the enclaved application, its content will be updated to the EIM. The order of the inputs will also be reflected by the EIM to detect memory corruption attacks due to the manipulated order of inputs [89].

To illustrate the security guarantees ensured by External-Input Attestation, we draw an analogy to launch-time at-

✉ Guoxing Chen is the corresponding author

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242421
www.ndss-symposium.org

testation in the trusted boot. In the trusted boot, launch-time attestation ensures that the enclave begins in a known, expected state by verifying a measurement of its initial code and data. Similarly, External-Input Attestation aims to ensure that all state transitions triggered by external inputs follow an expected path. By validating the EIM against a *reference*, we could assert that the enclave's execution remains within its intended behavior. While this does not preclude bugs triggered by legitimate inputs, such issues are treated as *reliability concerns*, rather than *security violations*. In this paper, we focus on enabling such *trusted state transitions* throughout execution, with the following goal:

> With External-Input Attestation, **security** issues (*e.g.*, adversarial control-flow hijacking) are reduced to **reliability** issues (*e.g.*, software crashes due to unexpected user inputs).

The challenges in realizing the security guarantees of External-Input Attestation are two-fold. The first is how to measure all writes to the enclaved application from its external world, including inputs via the shared buffer, interrupts, and exceptions, to capture potential attacks such as Time-Of-Check-To-Time-Of-Use (TOCTOU) and replay, and secure the evidence (*i.e.*, measurements reflecting unexpected writes) even after the enclaved application is compromised.

The second challenge lies in how to verify whether a received EIM is as expected efficiently. Unlike launch-time attestation, which involves determining a reference measurement in advance to verify the initial state of the enclaved application, EIM relies on all writes in execution time and may vary across individual executions.

To address the above challenges, we propose EXIA, a framework to realize External-Input Attestation. In particular, (1) EXIA introduces a privileged attesting environment to ensure all writes to the enclaved application are properly measured and secure the integrity of the EIM even when the enclaved application is compromised due to memory corruption attacks. (2) For efficient reference EIM derivation, the current version of EXIA focuses on enclaved applications whose inputs are known to the verifier, including applications using publicly known data (*e.g.*, publicly released machine learning training datasets), and single-user applications where the input is private to individual users (and multi-user applications where all inputs within the same applications are shared among participating users) who are also the verifier (Sec. IV).

To demonstrate the generality of EXIA, two prototype implementations based on AMD SEV and Penglai, respectively, are provided. Our analysis and experimental validation confirm that EXIA could defend against attacks (via 2 PoC attacks) originating from unintended writes caused by a compromised operating system, CPU, or host program. Furthermore, real-world case studies demonstrate minimal performance impact, including secure ML training and inference, database workloads, and key management. For CNN models, the overhead of EXIA is $0.80-5.89\%$ (LeNet-5) and $0.44-4.72\%$ (AlexNet) for training, and $1.15-18.30\%$ (LeNet-5) and $0.53-2.58\%$ (AlexNet) for inference, depending on input batch sizes. For transformer models, EXIA introduces 0.47% overhead to the training of a 124M-parameter GPT-2 model. Beyond machine learning, EXIA imposes a $0.29-1.38\%$ overhead on database workloads using common SQL commands with SQLite3 library. In key management, EXIA with MBedTLS library introduces a $0.06-1.94\%$ overhead for key operations and $2.13-2.14\%$ overhead for message signing and signature verification tasks.

In sum, the contributions of this paper are as follows.

- It proposes a novel concept of External-Input Attestation and a security property of trusted transitions for detecting memory corruption attacks.
- It introduces EXIA, a framework that can be adapted to various TEE existing implementations to enable External-Input Attestation.
- It implements and evaluates two prototypes[1] of EXIA based on AMD SEV and Penglai to demonstrate the generality, effectiveness, and performance.

II. BACKGROUND

A. Trusted Execution Environment

Trusted Execution Environment (TEE) constitutes a secure enclave within a CPU that is designed to safeguard the confidentiality and integrity of data and code at the hardware level, thereby providing a defense against a malicious operating system. The TEE protects code and data by remote attestation, supported by a hardware-based Root of Trust (RoT) embedded directly within the chip. Various implementations of TEEs have emerged over the years, including Intel Software Guard Extensions (SGX) [53] and Trust Domain Extensions (TDX) [55], AMD Secure Encrypted Virtualization (SEV) [58], ARM TrustZone [13] and Confidential Compute Architecture (CCA) [68], as well as Keystone [65] and Penglai [42].

1) Remote Attestation: Remote attestation is a process that confirms the integrity and authenticity of a TEE platform's hardware, associated driver, firmware, and microcode. Before moving forward, users are provided with a signed report from the vendor that certifies the TEE's validity, authenticity, and proper configuration.

2) Measurements: The measurement of the enclave, typically represented by the cryptographic hash value of its contents, serves as a method for identifying the enclave. Successful verification of enclaves generally indicates that the measurements of the enclave are as expected.

B. AMD SEV-SNP

AMD SEV-SNP (Secure Encrypted Virtualization - Secure Nested Paging), the third generation of SEV, enhances its predecessors (SEV and SEV-ES) by adding integrity protection to the existing memory and register encryption that safeguards

[1]Prototype Implementations: https://github.com/xmhuangzhen/Exia

data confidentiality against hypervisor vulnerabilities. SEV-SNP counters page mapping manipulation attacks through a Reverse Map Table (RMP), which tracks and validates mappings between system physical addresses and guest physical addresses, enforcing strict access checks for read/write operations on private memory to block unauthorized access and ensure memory integrity.

1) Virtual Machine Privilege Level (VMPL): AMD SEV-SNP introduces the VMPL feature to enable at a higher privilege level than the guest OS, isolating critical code modules. VMPL has four hierarchical privilege levels (VMPL0 to VMPL3, with VMPL0 as the highest). Memory permissions (read/write/execute) in VMPLs are managed hierarchically via the RMP table, where higher VMPLs control lower ones. The Secure VM Service Module (SVSM) [15] at VMPL0 uses the `RMPADJUST` command to adjust memory permissions, enforcing strict access policies.

2) CVM-Hypervisor Communication: Communication between a CVM and the hypervisor is handled by the Guest-Hypervisor Communication Buffer (GHCB) [17]. For the CVM with the VMPL feature, each privilege level can communicate directly with the hypervisor using the GHCB protocol. For example, to switch VMPLs, the code in the current VMPL sends a specific request via the GHCB, which then directs the hypervisor to transition the CVM to the new VMPL.

3) Attestation Process: A SEV-SNP VM starts with an unencrypted initial image (*e.g.*, boot code without secrets). During the launch, the hypervisor directs AMD-SP to install guest pages, which measure both content and metadata into a cryptographic launch digest, ensuring memory layout and content integrity. After the launch, the guest owner can provide a signed identity block containing VM identification and the expected digest, which AMD-SP verifies against the VM's actual digest and includes in attestation reports to validate the trusted configuration [59].

C. Penglai

Penglai is a TEE based on the RISC-V Instruction Set Architecture. Leveraging the RISC-V Trap Virtual Memory (TVM) capability, Penglai-TVM deploys the guard page table mechanism, facilitating page-level isolation. This isolation is critical to maintaining a secure separation between the untrusted host environment and the trusted enclave.

1) Host-Enclave Communication: The Relay Page is a communication mechanism between an enclave and a host in Penglai, where a secure monitor guarantees that a memory page can be mapped exclusively to one owner at a time. This approach mitigates security vulnerabilities, such as TOCTOU attacks in host-to-enclave interactions, while enabling zero-copy communication.

2) Measurements: Penglai utilizes the ShangMi 3 (SM3) [1] cryptographic hash function for calculating enclave measurements. SM3 is comparable to SHA-256 in structure and security [111]. The enclave measurement is produced by a secure monitor that scans the enclave memory from the lowest to the highest address.

3) Attestation Process: The host initiates the attestation request, which is then sent to the operating system and subsequently to the secure monitor to perform the attestation service. The attestation service accesses the secure memory manager, which employs the guarded page table to traverse the memory space and obtain a measurement of the entire memory content. After calculating the measurement, the attestation service generates an unforgeable signature of the measurement, creates a report, and then sends the report back to the enclave or host.

III. Overview

A. Problem Formalization

In this paper, we aim to address the problem of attesting the integrity of execution traces of sensitive data processing protected by TEE, beyond the attestation of only the initial state. Though VM-based TEEs enable running legacy applications without modifications, including a sophisticated guest OS in the Trusted Computing Base (TCB) poses significant security concerns [12], [102].

Hence, in this paper, we focus on process-based TEEs and VM-based TEEs with compartmentalization solutions to exclude the guest OS from the TCB [12], [102]. The compartmentalization secures the enclaved application even when the guest OS running inside the CVM is compromised. The justification includes that commodity kernels make the TCB particularly large [12] and that a sophisticated OS raises the possibility of dynamically creating and executing arbitrary code [102].

Since the untrusted privileged software cannot directly write data to the enclave memory nor modify the enclave code due to the integrity protection of TEE, an enclaved application can be modeled as a state machine, and all "writes" to it from its external world can be modeled as inputs to the state machine, as follows.

Definition 1. *An enclaved application is defined as a state machine $Encl = (\Sigma, S, s_0, F, \otimes, \Delta, \epsilon)$ where*

- Σ*: Input alphabet (a finite non-empty set of symbols);*
- S*: Set of enclave states;*
  - $s_0 \in S$*: Initial enclave state;*
  - $F \subseteq S$*: Set of final enclave states;*
    - $\otimes \in F$*: Abnormally terminated enclave state;*
- Δ*: State-transition function $\Delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow S$;*
  - ϵ*: A symbol indicating spontaneous transitions that occur without any inputs.*

Note that the input alphabet Σ models all potential inputs to the enclave by the untrusted privileged software, including both (1) user inputs presented to the enclave by the untrusted privileged software on behalf of the remote user, and (2) interrupt events generated by the privileged software and underlying hardware that could potentially occur when the enclaved application is in any state.

Next, we define execution traces of the above state machine.

Definition 2. *An execution trace Tr of a state machine $Encl$ is a sequence $Tr = (s_0, e_1, s_1, e_2, s_2, \ldots, e_n, s_n)$ where*

- *s_0: Initial enclave state;*
- *$e_i \in (\Sigma \cup \{\epsilon\})$: An event (either an input $\sigma \in \Sigma$ or ϵ) that causes the i-th transition.*
- *s_i: State after the i-th transition;*
- *$s_n \in F$: A final enclave state.*

The i-th transition can be represented as a triple (s_{i-1}, e_i, s_i). The subtrace $Tr_{\neg\epsilon}$ of non-ϵ transitions within trace Tr is defined as

$$Tr_{\neg\epsilon} = \{(s_{j-1}, e_j, s_j) \,|e_j \neq \epsilon, (s_{j-1}, e_j, s_j) \in Tr\}$$

Since spontaneous transitions occur without inputs, we now focus on modeling the integrity of the non-ϵ subtrace:

Definition 3. *Given a non-ϵ subtrace $Tr_{\neg\epsilon} = \{(s_{j-1}, e_j, s_j) \,|e_j \neq \epsilon, (s_{j-1}, e_j, s_j) \in Tr\}$, its EIM is defined as the output of a measuring (hashing) function $\mathcal{M}_{EIM}$ taking the sequence of inputs $\{e_j | \,(s_{j-1}, e_j, s_j) \in Tr_{\neg\epsilon}\}$.*

While the introduction of EIM has the potential for verifying whether all inputs received by the enclaved application are as expected (*i.e.*, not manipulated by the untrusted privileged software), directly asking the enclaved application to measure all inputs is not enough to guarantee the integrity of the execution traces.

Firstly, the design should ensure the integrity of EIM whenever the enclave is compromised during runtime. Since this paper considers an enclaved application that could potentially be compromised by manipulated inputs, an adversary controlling such an application might try to directly modify the value of EIM or prepare benign inputs to be updated to EIM.

Secondly, the design should enable the user (with knowledge of inputs sent to the enclaved application) to reason about the integrity of the execution traces from a received EIM without knowledge of potential interrupt events that occurred during runtime.

These challenges have motivated our research:

Can we design a framework for attesting the integrity of execution traces with only user-side knowledge?

B. Threat Model

In this paper, we aim to attest the execution traces of enclaved applications that are potentially vulnerable to memory corruption attacks that break software integrity. As shown in Fig. 1, existing memory corruption attacks against TEE can be divided into three categories: (1) code-injection [104], [69], (2) control-flow hijacking [7], [20], [66], [107], [6], [22], [85], [52], [99], [95], [9], [8], [80], [109], [108], and (3) data-only attacks [107], [6], [2], [56], [22], [52], [99], [95], [9], [80], [108]. Note that these attacks all start from unintended writes, which are different from what the user expects. Information leaks caused by unintended reads are considered orthogonal [19].

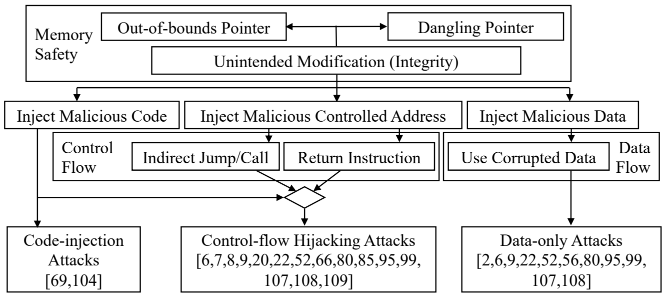
Memory Safety
Out-of-bounds Pointer
Dangling Pointer
Unintended Modification (Integrity)
Inject Malicious Code
Inject Malicious Controlled Address
Inject Malicious Data
Control Flow
Indirect Jump/Call
Return Instruction
Use Corrupted Data
Data Flow
Code-injection Attacks [69,104]
Control-flow Hijacking Attacks [6,7,8,9,20,22,52,66,80,85,95,99, 107,108,109]
Data-only Attacks [2,6,9,22,52,56,80,95,99, 107,108]

Fig. 1: Memory corruption-based attacks considered in this paper. Adapted from Ammar *et al.* [19].

Following the standard TEE threat model, we consider an adversary who has complete control over the hypervisor, operating system, and all system software outside the enclave, allowing them to deploy and execute any malicious applications on the host platform. The goal of the adversary is to launch memory corruption-based attacks on enclaved applications without being detected by EXIA.

On the other side, we assume that the CPU processor is correctly implemented, particularly in preventing unauthorized access to the enclave from external sources such as other enclaves or the operating system, enforcing protection mechanisms, and executing all required steps for the launch-time remote attestation protocol. Transient-execution attacks (such as Meltdown [70] and Spectre [61]) that leverage hardware vulnerabilities are out of scope. Interrupt events caused by the CPU are authentic and unforgeable, *i.e.*, the kernel code filters fake interrupts [82], [83]. We also assume the components of EXIA are trusted (*i.e.*, within the TCB).

Further, micro-architectural side-channel attacks [101], [86], [24], [48], [45], [67], [97], [105], [88], [96], [28], [98], [84], [26], denial-of-service attacks [44], and physical attacks [64], [72], [43] are also considered orthogonal and can be addressed separately by existing solutions [81], [11], [35], [38], [31].

IV. DESIGN

EXIA operates on the query-response paradigm. The user communicates queries and responses with the platform (including the privileged software and the host program) using a traditional secure channel such as TLS/SSL. Queries are forwarded to the enclaved application for processing, and responses along with EIM are returned to the user afterwards.

To ensure that EIM could capture the integrity of the execution traces, EXIA incorporates three components: (1) Privileged Attesting Environment for protecting EIM from potentially compromised enclaved application (Sec. IV-A); (2) Trusted Input Gateway for capturing all writes to the enclaved application (Sec. IV-B); (3) Trusted Interrupt Handling for preventing privileged attacks via manipulating interrupts.

The workflow of EXIA is illustrated in Fig. 2. Firstly, the attesting environment at the privileged level initializes itself and the EIM (**Step** ①). During query processing, the attesting environment receives the input, digests a measurement of it,

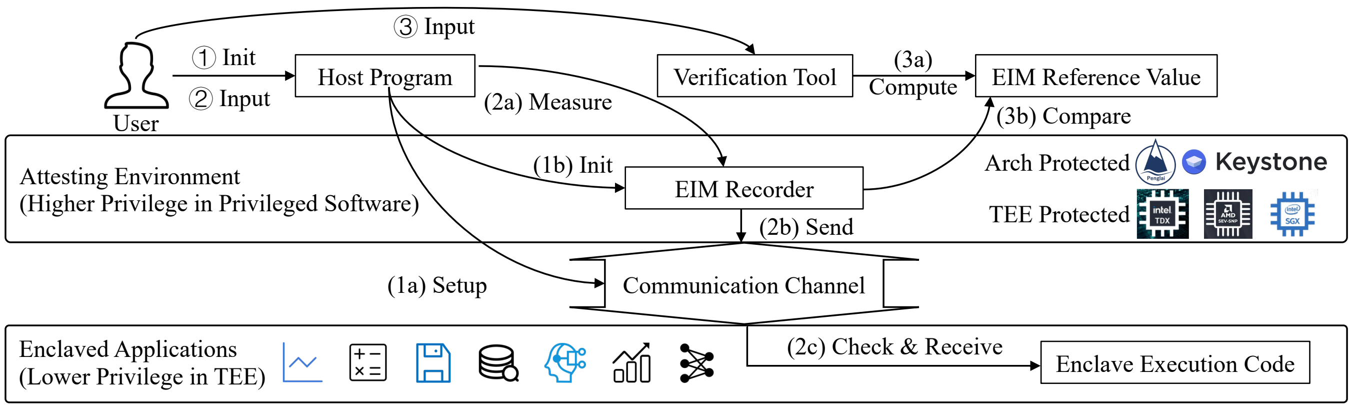

Fig. 2: Design overview.

and appends that measurement to the EIM (**Step** ②). The verification workflow requires the user to then locally derive a reference EIM from the query they originally sent and compare it with the EIM received in the system's response (**Step** ③). If the two values do not match, it indicates that the privileged software received some unintended queries to the enclaved application, rendering the response untrustworthy.

Application Scope. EXIA is used to verify whether all external inputs are *as expected* and applies to two primary scenarios. (1) For applications where the input is known offline, such as publicly known training datasets, any user with a specific ML training task could use EXIA to verify that the consumed training datasets are as expected, *i.e.*, she can confirm that the correct and unaltered dataset is being used for their tasks. (2) For single-user applications where the input is private to individual users (and multi-user applications where users within the same applications are willing to share input data), it is the user's own interest to verify whether the enclave processes the data without being affected by malicious input manipulated by the host. In these cases, the user can derive a reference EIM online from the private data she sends to the enclave for each run of the enclaved application, without needing to restrict the range of data content and size in advance.

A. *Privileged Attesting Environment*

An attesting environment needs to maintain and secure the EIM throughout the execution of the enclaved application. Particularly, the attesting environment needs to satisfy the following invariants:

- **Inv. 1-1**: The attesting environment needs to be isolated and protected from the enclaved application to avoid being polluted by a compromised enclaved application.
- **Inv. 1-2**: The attesting environment needs to measure user inputs first before passing them to the enclaved application for processing to avoid handoff attacks.

One straightforward way to meet both of the above invariants is to utilize an existing (or introduce a new) more privileged level for the attesting environment than that of the enclaved application. Potential options are as follows.

Off-the-shelf Privilege Leveling. For commercialized TEE implementations with intra-enclave privilege separation, such as AMD SEV (with SVSM [15]) and Intel TDX (with L1 VMM [55]), the attesting environment can be realized directly at a higher privileged level than that of the enclaved application. The integrity of such a hardware-software co-design attesting environment can be verified through existing remote attestation mechanisms. Taking AMD SEV as an example, the SVSM is designed to be loaded and measured during the initialization of the guest image. This ensures that the attesting environment is incorporated into the measurement of the SVSM binary, enabling the verification of the attesting environment's integrity during the launch-time attestation.

Software-based Isolation. For commercialized TEE implementations without intra-enclave privilege separation, such as Intel SGX, utilizing software-based isolation techniques, such as Software Fault Isolation (SFI), could create a more privileged compartment for the attesting environment within the same enclave as the enclaved application. SFI is usually implemented via LLVM instrumentation. The instrumented enclave results in a new launch-time measurement that can be used to authenticate the integrity of the attesting environment during the launch-time attestation.

Architectural Extension. Besides the above two options, for open-sourced and customizable TEE implementations such as RISC-V based Penglai, it is also feasible to extend the trusted hardware/firmware to realize the privileged attesting environment, which will be included in the TCB of the underlying platform. Taking the RISC-V architecture as an example, the TCB is captured by the Security Version Number (SVN) included in the attestation evidence generated by the Root-of-Trust for Reporting (RTR) [78]. This SVN serves as metadata representing the security posture of the TCB and is designed as a monotonically increasing number that is incremented whenever relevant security updates or changes are made to the TCB components.

Depending on the concrete hashing algorithm used for EIM, the initial hash state is hardcoded in the attesting environment, designated as the initial value of EIM, *i.e.*, EIM_0. Upon launch, the enclaved application enters an initial state s_0.

B. Trusted Input Gateway

Upon the successful launch of the enclaved application, the EIM is initialized within the attesting environment. This ensures its integrity remains preserved, even in the event of an enclaved application compromise. Throughout the enclaved application lifecycle, the EIM is dynamically updated whenever user input is provided, capturing the content of such interactions.

To ensure that all writes to the enclaved application can be captured, the following invariants need to be achieved.

- **Inv. 2-1**: The enclaved application does not access addresses directly shared with the host program to mitigate TOCTOU attacks.
- **Inv. 2-2**: The freshness of measured inputs must be verified to mitigate replay attacks.

To satisfy **Inv. 2-1**, the attesting environment could adopt either *dual-page transfer* or *shared-page permission flipping*. These mechanisms aim to secure the communication between the host program and the enclaved application, ensuring the integrity of input data by preventing the host program from modifying inputs after they have been measured.

Dual-Page Transfer. TEEs, such as AMD SEV VMPL, ensure that higher privilege levels can access lower privilege level memory, but not vice versa. To counter TOCTOU attacks, a dual-page transfer mechanism can be employed for secure host program to enclaved application communication: the attesting environment shares pages with the host program to receive the input data written by the latter, while simultaneously sharing separate pages (inaccessible and immutable by the host program) with the enclave to transmit and measure the host program input securely. To prevent TOCTOU attacks, the attesting environment sets a freshness flag only after it has completed measuring the inputs. This ensures that if an interrupt from the OS or CPU occurs during the measurement phase, the process will be incomplete and the flag will not be set. So the absence of the freshness flag indicates that the input measurement is not finished or recorded, thereby mitigating the attack.

Shared-Page Permission Flipping. TEEs, such as Penglai, employ a shared page permission flipping mechanism (called *relay page* in Penglai) to communicate user inputs between the host program and the enclaved application securely. The process operates as follows: (1) The host program prepares the user input in a dedicated buffer page, referred to as the relay page; (2) Upon switching to enclave mode, the platform creates a page table entry that maps virtual addresses within the enclave to the relay page, and simultaneously revokes the page table entry mapping virtual addresses in the host program to the same page. This ensures that the host program can no longer modify the content of the relay page. By leveraging the relay page mechanism, EIM can be efficiently updated whenever a relay page is added to the enclave. This approach provides a secure and streamlined method for transferring inputs between the TEE and the host program. To avoid TOCTOU attacks, the attesting environment must ensure that the input data measurement is performed after the permission switch for the shared page has been correctly executed and before the enclaved application starts to process it, so that the measurement reflects the corresponding input.

To satisfy **Inv. 2-2**, a freshness flag can be introduced to indicate the recency of inputs and detect potential replays.

Freshness Verification. To counteract host program replay attacks where the inputs are reused to bypass the attesting environment's measurement (for scenarios when the host program could invoke the enclaved application directly without notifying the attesting environment), a freshness flag within the communication buffer can be introduced to signify input recency. Specifically, upon receiving an input, the attesting environment measures it and then sets this freshness flag in the buffer destined for the enclaved application. Before utilizing any input, the enclaved application needs first to verify that the freshness flag is set. After receiving the inputs, the flag is cleared. When the enclaved application receives user input e_i, it transitions from its current state s_i to the next state s_{i+1} only if the input passes a freshness flag verification. Otherwise, if the verification fails, the application will enter an abnormal state $\otimes$.

C. Trusted Interrupt Handling

The execution of outsourced computations may be interrupted by various events from time to time. These interrupts can be categorized as external interrupts, software interrupts, and exceptions. Detailed categorization is in Appendix A.

Event Transparency. To analyze how different interrupt events affect the execution of outsourced computations within an enclaved application, we categorize these events based on whether their handlers are transparent to the enclaved application or not:

- *Transparent Exit Events* (`TEx`): These are interrupt events that the OS can fully manage without modifying the execution context of the enclaved application, including (1) External interrupts; (2) Software interrupts that do not modify the control flow and data flow of the enclaved application; (3) Faults without a customized handler within the enclaved application, such as page faults that the corresponding OS can resolve entirely; (4) Faults that the OS/CPU cannot inject; (5) Traps.
- *Non-Transparent Exit Events* (`NTEx`): These are interrupt events that require customized handling within the enclaved application that can be affected by the host program, including (1) Software interrupts that require host program inputs, *i.e.*, system calls; (2) Faults that have customized handlers within the enclaved application and require host program inputs.

Since aborts typically lead to process termination and can cause denial-of-service attacks, which are considered orthogonal to the scope of this paper, we exclude them from the following discussion. The detailed architecture-defined interrupts are discussed in Appendix B.

In contrast, NTEx such as syscall is handled through a controlled process: the enclaved application sets the required syscall number and transfers control to the guest OS. After processing the request, the guest OS first provides the syscall details to the attesting environment for security measurement and then signals the enclaved application to resume its program execution. To securely handle NTEx, EXIA needs to satisfy the following invariants:

- **Inv. 3-1**: For NTEx without host program inputs, the authenticity of the inputs needs to be checked within the customized handler.
- **Inv. 3-2**: For NTEx with host program inputs, the host program inputs need to be synchronized with the user and integrated into the request-response protocols.

To handle an NTEx, an enclaved application can utilize a custom handler if the underlying architecture, such as Intel SGX/TDX AEX feature [54], provides support for it. Upon encountering an NTEx interrupt, this handler empowers the enclaved application to decide its course of action. The customized handler can either choose to terminate its execution or signal to the host program that it should continue, potentially after receiving further input, thus maintaining control over how it responds to such events.

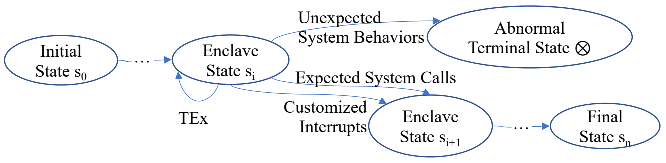

Fig. 3: Enclaved application state transition via trusted interrupt handling.

As shown in Fig. 3, an enclaved application in state s_i responds to an interrupt with one of four potential state changes. If the interrupt is a TEx, representing the transition (s_i, ϵ, s_i), the application returns to its original state. For a customized interrupt or an expected system call, the enclaved application state advances to s_{i+1}. However, an unexpected interrupt forces the enclaved application into an abnormal terminal state $\otimes$. The detailed system calls are discussed in Appendix C.

D. EIM Verification

After properly selecting the input transfer methods and checking the communication buffer, the EIM calculation and verification processes are as follows.

EIM Calculation Process. The calculation of the EIM needs to be performed within the attesting environment. This process begins with an initial value EIM_0. Upon receiving the i-th input from the user, the attesting environment computes a measurement of that input by hashing it. Then, it calculates the current EIM state as

$$\mathit{EIM}_i = \mathcal{H}(\mathit{EIM}_{i-1}, \{e_i | \, (s_{i-1}, e_i, s_i) \in Tr_{\neg\epsilon}, \}).$$

The EIM effectively chains each new input measurement to the previous EIM state through iterative hashing. This mechanism records the EIM in the form of a cryptographic hash chain. When the execution phase concludes after processing a total of n external inputs, the resulting final EIM value is EIM_n.

EIM Verification Process. The verification process for EIM operates as follows: when a user sends input to the enclave, the verification SDK automatically records this input and calculates an EIM value using the same process as the attesting environment. Upon completion of execution, the SDK generates an EIM reference value designed to compare it against the value produced by the attesting environment. Significantly, this verification SDK is not restricted to placement solely at the user's sending point; it can be deployed at any location capable of recording or receiving the relevant input. Regardless of where the reference value is generated, the user can compare it against the value provided by the attesting environment.

E. Security Analysis

To analyze the security of EXIA, one lemma followed by a theorem is proved as follows. Note that the analysis excludes reliability issues (which are not the focus of this paper) and focuses on security issues. That is, we assume that the benign external inputs to the enclaved application will not violate EXIA's design invariants while the adversary intentionally tries to violate them to launch memory-corruption attacks, by providing malicious inputs.

Lemma 1. *The first malicious input causing violations of* EXIA*'s design invariants will be captured by EIM.*

Proof. Since the design of EIM in EXIA follows a measure-before-process pattern, the first malicious input will be updated into EIM before it takes effect to violate the design invariants and compromise the enclaved application. Particularly, directly compromising **Inv. 1-1** or **Inv. 1-2** is considered out of scope, since the attesting environment is within the TCB. To violate **Inv. 2-1**, the adversary needs to have (1) a shared memory with the enclaved application and (2) trigger the latter to read from it to bypass EIM. Since the enclaved application is developed using EXIA provided APIs for receiving inputs from the designated buffer that is not accessible to the adversary-controlled software stack, triggering the enclaved application to read from other addresses requires providing malicious addresses to the enclaved application in advance, which will be updated to the EIM. **Inv. 2-2** and **Inv. 3-1** will not be violated since the host program cannot provide any input to the enclaved application, while **Inv. 3-2** resembles the case of **Inv. 2-1**. Hence, at least the first malicious input will be updated to EIM. □

Theorem 1. *If the received initial state s_0 and EIM match the reference values the user derives locally from its benign inputs,* *the user can ensure that the resulting state of the enclaved application is trusted.*

Proof. Using existing launch-time attestation, the user could authenticate the initial state s_0. By verifying that the EIM matches the reference value derived from benign inputs, the user could ensure that no malicious inputs have been consumed by the enclaved application (Lemma 1). Note that the execution trace of the enclaved application starts from s_0, which will be verified first. Each transition from a state s_{i-1} to the next state s_i is prompted by an event e_i, which is either an external input or a spontaneous transition. `TEx`s could occur when the enclaved application is at any state but will not transit the enclaved application to a different state since the handling of `TEx` results in a self transition (s, ϵ, s). The handling of a `NTEx` without host program inputs results in a sequence of spontaneous transitions, each of which transit the enclaved application from a trusted state s_{i-1} to a trusted state s_i. Meanwhile, all benign inputs will also trigger trusted transitions. Therefore, by construction, as long as the received initial state s_0 and EIM match the reference values, the enclaved application continues to transition to a trusted state. □

V. Exia Implementation for AMD SEV-SNP

In this section, we describe a prototype implementation of Exia based on AMD SEV-SNP, dubbed Exia-SEV. This implementation comprises approximately 2000 LoC in the TCB.

This prototype is based on Veil [12][2], a framework that shields sensitive programs from a potentially buggy guest OS. Particularly, the enclave operates in user mode at VMPL2, while the guest OS and host program run at VMPL3 and thus cannot access the enclave's memory.

The attesting environment is implemented within SVSM at VMPL0. The attesting environment adopts dual-page transfer for trusted input gateway such that inputs are routed from the host program to the attesting environment, measured, and then transferred to the enclave through an Enclave-SVSM buffer. System calls designated as `NTEx` listed in Table V are supported similarly via dual-page transfer.

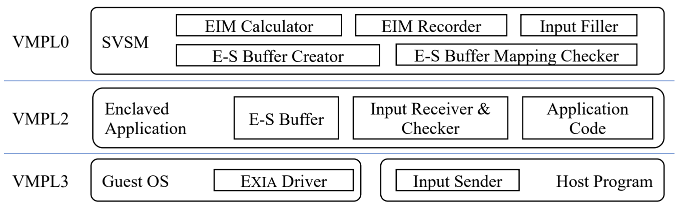

Fig. 4: Exia-SEV design.

The workflow of this prototype is shown in Fig. 4: (1) the Enclave-SVSM buffer is initialized by the E-S buffer creator, and the EIM recorder is initialized by the hash algorithm (Sec. V-A); (2) host program transfers inputs to the attesting environment via Input sender, and the attesting environment measures input via the EIM calculator and the EIM recorder, and fills the Enclave-SVSM buffer. The enclave is then invoked to validate and process the input (Sec. V-B), and (3) the response along with EIM is forwarded to the user for verification (Sec. V-D).

[2]The open-sourced implementation of Veil (commit 45a9615) is based on Ubuntu 22.04. We made necessary modifications to the KVM, guest OS, and driver components to adapt it to support Ubuntu 24.04.

A. Initialization of E-S Buffer and EIM Recorder

To establish a secure communication channel between the SVSM and an enclave to transfer input, an Enclave-SVSM buffer is initialized to ensure trusted data exchange by isolating operations from lower privilege levels.

Potential attacks from the untrusted guest OS (VMPL3), host program (VMPL3), or hypervisor include accessing or modifying the buffer's corresponding page directly, or maliciously remapping the buffer's user virtual address to another physical address, causing unintended access or modification of malicious memory locations.

Therefore, the buffer should prohibit all access permissions for VMPL3 while also validating the mapping between the virtual addresses of the E-S buffer and the corresponding physical addresses to ensure secure memory access and prevent unauthorized manipulation.

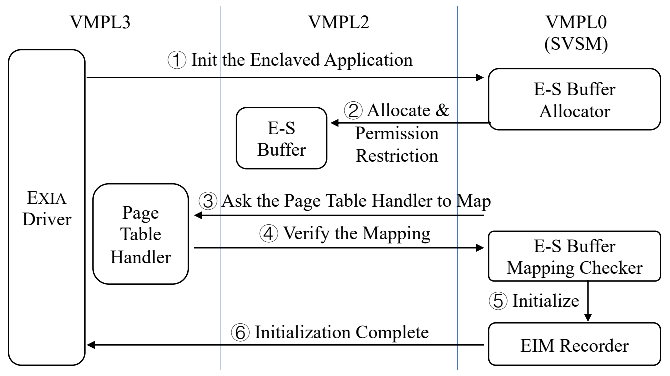
VMPL3
VMPL2
VMPL0 (SVSM)
① Init the Enclaved Application
E-S Buffer Allocator
② Allocate & Permission Restriction
E-S Buffer
EXIA Driver
Page Table Handler
③ Ask the Page Table Handler to Map
④ Verify the Mapping
E-S Buffer Mapping Checker
⑤ Initialize
⑥ Initialization Complete
EIM Recorder

Fig. 5: Initialization of E-S Buffer and EIM recorder process.

As shown in Fig. 5, the initialization of the buffer is triggered by the host program to SVSM through a request (**Step** ①). Upon receiving the request, the SVSM allocates the specified buffer size as the Enclave-SVSM buffer, and then configures the access permissions for these pages, granting full read, write, and execute privileges to VMPL 0, VMPL 1, and VMPL 2 (**Step** ②). Conversely, VMPL3, where the guest OS and host program operate, is explicitly restricted from accessing the pages, with no read, write, or execute permissions permitted. This modification of access permissions for these pages can be done using `RMPADJUST` as mentioned in Sec. II-B1. After the SVSM allocates the buffer pages, it sends the physical address of the buffer pages to the enclave's page table handler. The handler then maps the enclave buffer's user virtual address to this physical address (**Step** ③). This enables the enclave to access the buffer via the user virtual address. Regardless of where the enclave's page table handler resides, whether in VMPL0 (SVSM) or VMPL3 (guest OS, where VEIL is implemented), the SVSM can access the enclave's page table because it retains knowledge of the enclave's CR3 register. The CR3 register is a processor register in x86 architectures that stores the base physical address of the page table for the current execution context, enabling virtual-to-physical address translation. By leveraging the enclave's CR3 value, the SVSM can verify that the enclave's user virtual address for Enclave-SVSM buffer is correctly mapped to the SVSM-allocated physical address of the buffer (**Step** ④). After the optional mapping verification, the SVSM will initialize the measurement recorder (**Step** ⑤), as the measurements are recorded in the form of a hash chain. The initialization concludes afterwards (**Step** ⑥).

B. Secure Input Transmission and EIM Update

After initialization, the outsourced computation begins by transmitting external inputs to the enclave via the attesting environment and then invoking the enclave to process them.

Potential attacks from the untrusted guest OS involve (1) *input manipulation*, *i.e.*, sending more, fewer, or modified input than what the user originally submitted to the platform, resulting in the enclave receiving incorrect, excessive, or insufficient input; and (2) *input replay*, *i.e.*, resuming enclave execution without involving the attesting environment to properly measure the input and fill it to the Enclave-SVSM buffer, bypassing SVSM input control.

To address input manipulation attacks, EXIA measures the input upon each transmission and allows the user to verify after each transmission by EIM. If the measurement is unexpected, the user will find the input manipulation attack and stop transferring the data.

To detect input replay attacks, the enclaved application checks a freshness flag before each execution to determine whether to proceed. If this flag is not set, the enclaved application identifies the execution as a potential replay attack. It immediately halts its execution to prevent the system from further attack.

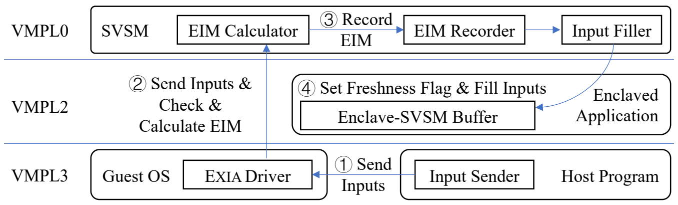

Fig. 6: Input transfer workflow of EXIA-SEV.

As shown in Fig. 6, the workflow of the input transfer is initiated by the host program, and forwarded through the guest OS to the attesting environment. Then the attesting environment zeros out the buffer, copies the input size and input data into the buffer, and performs a SHA-512 measurement on the buffer contents. After the measurement calculation, the attesting environment updates the measurement hash chain with the resulting digest. Finally, the attesting environment sets the buffer's freshness flag to true, marking the data as valid and measured while preventing unauthorized reuse.

The enclaved application is then invoked to process the inputs. To prevent potential attacks, such as a replay attack, the enclaved application first verifies a freshness flag. If the freshness flag is set, the enclaved application reads the subsequent input size and data, clears the buffer, and then clears the flag to prevent the data from being used again. However, if the flag is not set, the enclaved application treats this as a replay attack and the enclaved application immediately stops execution.

C. SDK Extensions and Developer APIs

We have integrated this feature into the VEIL SDK level, eliminating the need for developers to implement it separately, as the functionality is now natively available within the development toolkit. Existing enclaved applications that could run on vanilla VEIL can run on EXIA-SEV after recompilation, without any source code modification. The substitution of input functions is handled at the SDK level during compilation, making the process seamless and transparent to developers.

The original VEIL framework requires developers to manually partition their applications into a host program, responsible for system calls and initialization, and an enclave component, which contains the core trusted logic. To adopt EXIA-SEV, the modifications to the host program include calling the initialization function `init_EXIA()` and the input transfer function `transfer_input_EXIA(input_addr,input_size)`. Meanwhile, the modification to the enclaved application is reading the input from the specific Enclave-SVSM buffer `get_input_EXIA()`, rather than directly from the host-provided buffer as is done in the original VEIL implementation.

D. Verification of EIM Recorder Report

After the execution of the enclave program, the EIM recorder generates a report containing hash values of the measurement chain formed by all input measurements. We provide a verification code, so the verifier can compute a reference value by providing all enclave inputs and then compare the code-generated reference value with the hash value in the report to validate the execution.

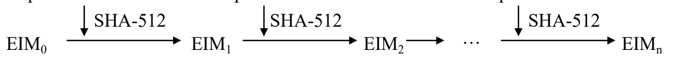

Fig. 7: EIM hash chain.

The process of measurement verification involves validating the hash value generated by SVSM's hash chain, which encompasses all input measurements. As illustrated in Fig. 7, this hash chain operates by having the SVSM compute a measurement for each received input. This measurement then becomes a component of the hash chain, which is iteratively processed using SHA-512 to produce the final hash value.

To verify this value, the verifier must possess full knowledge of all enclave inputs. This enables the recreation of each input measurement, subsequent recalculation of the hash chain, and comparison of the resulting hash value to validate authenticity. We support user applications in retaining a local backup before submitting inputs to verify the EIM, while also enabling third-party validation of the EIM when all input data is accessible. This ensures that the application and external parties can independently confirm the EIM, provided they have the complete input information required for verification.

VI. Exia for Penglai

In this section, we describe another prototype implementation based on Penglai, dubbed Exia-Penglai, to demonstrate the generality of Exia. Exia-Penglai comprises approximately 200 lines of code in TCB. The implementation in Penglai is also transparent to application developers, so existing enclaved applications running on vanilla Penglai could run on Exia-enhanced Penglai after recompilation.

The attesting environment is implemented within the secure monitor, which operates within the highest-privilege mode, specifically machine mode in the RISC-V architecture. Since Penglai adopts relay pages for the communication method between the host program and the enclaved application, as introduced in Sec. II-C1, the attesting environment measures relay pages upon each write to the enclaved application.

Potential attacks from the untrusted OS on the input flow include the following:

- Manipulated input. The operating system may modify the inputs provided to the enclave.
- Falsified input. The operating system may add or reduce the inputs to the enclave.
- TOCTOU attack on the input. The operating system modifies the content of the relay page after the secure monitor processes the input.

To address these threats, an EIM recorder is integrated into the attesting environment to capture parameters and transmit data upon entering an enclave. Within the EIM recorder, the content of the relay page (*i.e.*, the input data) is measured to record the transfer process execution. This approach enables precise tracking and verification of critical information as it transitions into the enclave. The TOCTOU vulnerability associated with the relay page is mitigated through a relay page design that restricts ownership to a single user at any given time, as outlined in Sec. II-C1.

The workflow of the input transfer is illustrated in Fig. 8. The host program sends input to the OS, which places it into a relay page. The secure monitor then transfers access rights to the relay page to the enclave, measures the relay page's contents, and records the EIM. Following this, the enclave can read the measured input from the relay page, ensuring integrity and confidentiality throughout the process.

The verification process of EIM in Penglai aligns with that of SEV-SNP. The host program, the user, or other data owners

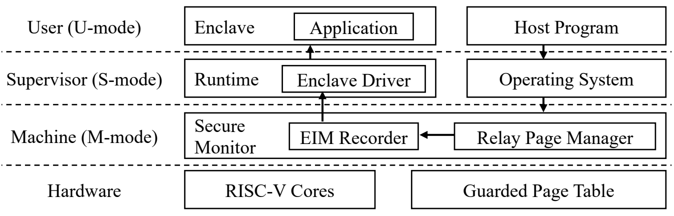

Fig. 8: Workflow of Exia-Penglai.

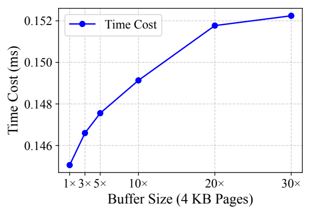

Fig. 9: Initialization time cost.

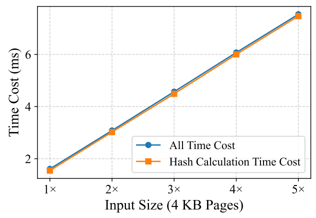

Fig. 10: Transmission time cost.

can generate a reference value for the execution trace using their data, which is then compared against the corresponding value produced by the secure monitor to ensure the integrity of the enclave's execution trace.

VII. Evaluation

This section presents evaluation results of our implementations through multiple microbenchmarks and case studies.

AMD SEV-SNP Setup. All AMD SEV-SNP experiments were conducted on a server equipped with an AMD EPYC 9474F 48-Core CPU and 2.2 TB of DDR5 memory. Within this environment, we configured an SEV-SNP virtual machine with SVSM support, allocating 4 vCPUs, 4 GB of memory, and 100 GB of storage, enabling VirtIO. The host OS ran Ubuntu 24.04 with a kernel version 6.7.0-snp-host (using AMD's svsm-preview-hv-v4 branch [16]), while the guest OS ran Ubuntu 24.04 with a kernel based on version 6.8.0-snp-guest (using AMD's svsm-preview-guest-v4 branch). Both the guest and host OSs used GCC version 13.3.0.

Penglai Setup. The Penglai experiments were conducted using Docker version 28.0.1 and a qemu-system-riscv64 (version 4.1.1) instance configured with 4 GB of memory. This setup was based on the Penglai-Enclave-TVM [40] and ran on an Ubuntu 16.04 system with GCC version 5.4.

A. Performance Overhead

The performance overhead mainly consists of two parts: one-time initialization (for Exia-SEV only, since Exia-Penglai does not need an initialization process) and per-query input transfer (for both Exia-SEV and Exia-Penglai). Results are averaged over 1000 repetitions to ensure reliability.

Exia-SEV Initialization. According to Sec. V-A and Fig. 5, the initialization process involves two main stages: creating a buffer and verifying its mapping. Fig. 11 details the time cost of these operations, where each application-driver context switch takes 0.004 ms, an SVSM call or return costs 0.015 ms, and setting up the creation and permissions for a single 4 KB buffer page requires 0.055 ms. Therefore, the total time cost is (0.004 ms$\times$4) + (0.015 ms$\times$4) + (0.055 ms$\times N$), where N is the number of 4 KB buffer pages. Notably, this overhead is a one-time cost incurred only during the initial setup phase and does not recur in subsequent operations.

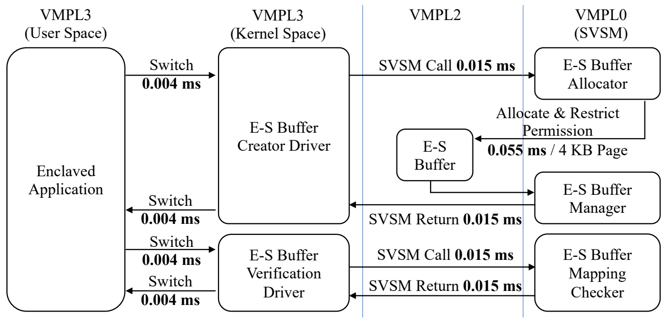

Fig. 11: Initialization main time cost in SEV.

Exia-SEV External Input Transmission. As described in Sec. V-B, the external input transmission process consists of two stages: transmitting input to the SVSM and performing a measurement calculation. The performance evaluation of this process (illustrated in Fig. 10) reveals that the highest cost is the measurement calculation, which utilizes the SHA-512 hash function and requires 1.51 ms for each 4 KB input page. Meanwhile, as previously mentioned, the overhead for each SVSM call or return is 0.015 ms, and the application-driver context switch is 0.004 ms. Therefore, the total time cost is roughly determined by the formula: (0.004 ms$\times$2) + (0.015 ms$\times$2) + (1.51ms$\times$ # 4 KB input pages).

Exia-Penglai External Input Transmission. As detailed in Fig. 8 and Sec. VI, the external input transmission process in Penglai focuses on measuring the input by applying the SM3 hash algorithm. This is the same hash function used for Penglai's attestation, and the time cost for this operation is 0.17 ms per 4 KB page of external input.

B. Security Evaluation

Our system could detect malicious inputs that attempt to exploit vulnerabilities in enclaves to launch attacks. Such attacks include overflow (*e.g.*, buffer overflow, heap overflow, stack overflow, and integer overflow), dangling pointer misuse (*e.g.*, use-after-free and invalid memory references), and format string vulnerabilities (exploiting improperly sanitized format specifiers[3]) [19]. By examining the vulnerable code patterns that can be leveraged to launch existing memory corruption attacks against the TEE listed in Fig. 1, we devised two Proof-of-Concept (PoC) programs to evaluate Exia's effectiveness against overflow and dangling pointer misuse.

[3]Format string vulnerabilities are usually due to improper use of `printf()`, which is unlikely needed in outsourced computation and thus excluded in the evaluation.

Listing 1: Vulnerable overflow PoC.

```
1 int *ptr; char buffer[10];
2 buffer=get_input();//cause buffer overflow attack
3 *ptr = xxx; // arbitrary write
4 *ptr(); // arbitrary execute
```

Listing 1 shows a PoC program executed in the enclaved application that contains a use-after-free vulnerability. In detail, the program allows the host program or guest OS, acting as the attacker, to modify (Line 3) any information or execute (Line 4) unintended behavior without affecting the output.

Listing 2: Vulnerable use-after-free PoC.

```
1 //host program and enclaved application shared pointer
2 int *shared_ptr=malloc(shared_size);
3 free(shared_ptr); //host program free the pointer
4 //enclaved application visits the shared ptr
5 *shared_ptr=xxx; //cause Use-After-Free
```

Listing 2 presents a vulnerable Use-After-Free PoC program that allows an attacker to free the shared pointer before the enclaved application visits the shared pointer (Line 5). The enclave code then directly reads this dangling pointer, which can cause unpredictable behavior.

Exia mitigates the above attacks by processing all input through a specific API, rather than relying on standard C functions that could insecurely interpret data provided by the host program. This approach avoids the direct use of the inputs from the host program (Listing 1) and the shared pointer (Listing 2).

As mentioned in Table V, our system's security design accommodates file system, I/O, and networking system calls, while recommending that memory management calls be handled via our input transfer design. We support `open()`, `openat()`, `close()`, `read()`, `write()`, `lseek()`, `stat()`, `unlink()`, and `socket()` system calls. Any attacker utilizing these supported system calls to transfer malicious inputs will be captured and recorded in EIM. For example, if an attacker attempts to transfer a harmful file or send incorrect socket information, this action will be recorded. To prevent unauthorized system call attacks (*e.g.*, using malicious file content), the enclaved application will immediately halt execution if it receives any unexpected or unsupported system calls.

TABLE I: Comparison with existing work.

Attestation Target	Unintended Write	Attacks on Interrupts
Control Flow [10], [39], [106], [94], [36], [51], [18]	◐	○
Data Flow [37], [94], [74], [103], [110]	◐	○
External Input (this work)	●	●

As indicated in Table I, while CFA/DFA frameworks could detect unintended writes that alter the program's control/data flow, they could not detect unintended writes that leave the control and data flow unmodified, denoted as ◐. Meanwhile, they can not identify attacks targeting system interrupts, such as interrupt manipulation, denoted as ○.

VIII. Case Studies

In this section, we demonstrate Exia's applicability through case studies in the two application scenarios outlined in Sec. IV: training of CNN and transformer models using publicly known datasets (Sec. VIII-A), ML inference with private inference queries (Sec. VIII-B), database workloads with private user commands (Sec. VIII-C), and key management services that handle private queries and data (Sec. VIII-D).

A. Model Training with Publicly Known Datasets

CNN Model Training. For CNN model training, we implemented LeNet-5 [63] (812 LoC) and AlexNet [62] (965 LoC) CNN models for handwritten digit classification on the MNIST dataset [63], running them within AMD SEV-SNP using the Veil framework [12]. Because MNIST is a publicly known dataset, the reference EIM can be derived by any verifier for attestation.

We evaluated the performance of implemented CNN models against baselines without the external-input attestation design and further isolated the specific overhead caused by the EIM calculation process. As shown in Fig. 12, the performance overhead for both LeNet-5 and AlexNet decreases as the input batch size increases. With a batch size of 128, the overhead is reduced to 0.80% for LeNet-5 and 0.44% for AlexNet.

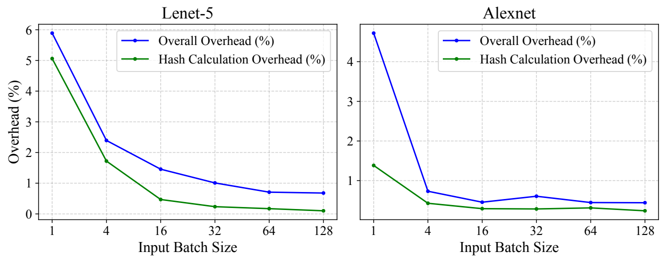

Fig. 12: CNN models training overhead.

Transformer Model Training. For transformer model training, we implemented the 124M-parameter version of GPT-2 [76] training on the public tiny_shakespeare dataset [60] in Penglai with 1891 LoC. Our implementation is based on the llm.c project [4], and confines the entire training process in an enclaved application, leaving the host program to only transfer parameters and input data. Similarly, the dataset is publicly available for reference EIM derivation. With a batch size of 1, Exia already achieves a low overhead of only 0.47% (averaged across 40 epochs).

B. Private Model Inference

The private inference process of LeNet-5 (791 LoC) and AlexNet (944 LoC) is evaluated using the same comparison framework as the CNN model training phase. The user, who is also the verifier, maintains a reference EIM for verification after sending each private inference query, and verifies the received EIM associated with the response before deciding whether to issue the next query.

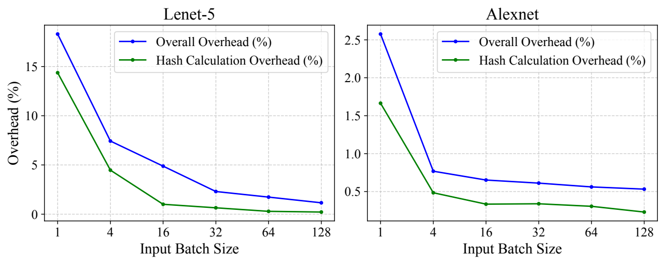

Fig. 13: CNN models inference overhead.

SQL Commands	Overhead(%)
`CREATE`	1.38
`INSERT`	0.57
`SELECT`	0.29
`UPDATE`	0.53
`DELETE`	0.39
`COMMIT`	0.45

TABLE II: Database workload overhead.

Key Management Operations	Overhead(%)
Generate Key	0.06
Find Key	1.89
Delete Key	1.95
Sign Data	2.14
Verify Signature	2.13

TABLE III: Key management overhead.

The performance overhead for both models is shown in Fig. 13. The overhead also diminishes as the input batch size grows. With an input batch size of 128, the inference overhead drops to 1.16% for LeNet-5 and 0.53% for AlexNet.

C. Database Workloads

For database workload scenarios, we presented an enclaved application (459 LoC) within Penglai that utilizes the SQLite3 library to execute SQL operations in an in-memory database. This allows a user to directly call the SQLite3 library within the enclaved application to perform SQL commands, including `CREATE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`, and `COMMIT`. The reference EIM for attestation can be derived by the user (the verifier) upon sending each SQL command.

As illustrated in Table II, when evaluated using the Chinook dataset [3], Exia incurs a minimal performance overhead for measuring the SQL command inputs, ranging from 0.29% to 1.38%.

D. Key Management

To handle key management, we implemented an enclaved application (605 LoC) within Penglai utilizing the MbedTLS library [5]. This library provides essential functionalities for key management, including key pair initialization, key pair generation (RSA, ECC, OPAQUE, *etc*), and key pair freeing. The library also provides secure message operations, such as calculating hashes, signing data, and verifying signatures. The reference EIM is derived by the user (the verifier), similarly to verifying the integrity of the past key operations.

As demonstrated in Table III, performance overhead evaluated with 1000 different RSA 2048-bit keys reveals that Exia introduces minimal overhead. The overhead for key management operations ranges from 0.06% to 1.94%, while the overhead for message signing and verification is also low, at 2.13% and 2.14% respectively.

IX. Discussion

Acceleration of Hash Function Calculation. According to the evaluation in Sec. VII, the hash function execution is the primary performance bottleneck, making its acceleration the critical priority for improving the design. This can be achieved through several methods, such as parallelizing the calculation with multiple threads [46], [49] or employing hardware acceleration with specialized CPU instruction sets or FPGAs [41], [90]. An alternative strategy offers a trade-off: randomly sampling inputs for the hash function reduces computation time but also increases the risk of failing to detect manipulated inputs [87], [100].

Concurrent Hashing to Improve Performance. To further enhance performance, the hash function can be calculated concurrently with the main program's execution. This approach allows the hash computation to run in parallel with the enclaved application execution, rather than waiting for the hash computation to finish. By overlapping these two tasks, the total execution time will be effectively reduced, resulting in a significant performance boost.

Extensions to Other TEEs. Adapting Exia for other TEEs requires different approaches tailored to their architectural designs. For instance, extending support to Intel SGX [53] or other process-based TEEs would necessitate using SFI to create an attesting environment that controls input to the enclaved application. In contrast, supporting RISC-V TEEs, such as Keystone [65] and Sanctum [34], involves modifying the architecture-level design to serve as the attesting environment. For other VM-based TEEs, such as Intel TDX [55], the L1 Virtual Machine Monitor (L1 VMM) [55] can be leveraged to separate the enclaved application from the attesting environment by placing them at different privilege levels.

TLS or Signed Files Approaches. Simpler security protocols, such as TLS and signed files, are insufficient for providing complete protection, as they remain vulnerable to memory corruption attacks. It has been demonstrated that TLS [32], [33], [29] (TaLoS, Rust SGX SDK's tlsclient, and WolfSSL) might contain memory vulnerabilities that a malicious host can exploit to compromise the TLS connection and leak future user inputs undetected. To address this, Exia captures malicious inputs that cause the compromise and informs the user of potential TLS issues.

X. Related Work

Control-Flow Attestation. CFA aims to ensure device security during runtime, while previous methods cannot detect runtime attacks after initial checks. BLAST [106] uses SFI to store path measurements constructed by a context-free grammar-based representation to verify the whole-program control-flow path. OAT [94] instruments after each conditional branch to collect the trace, and then runs symbolic execution to verify the trace. C-FLAT [10] instruments at the end of the basic block to calculate the hash of the block, and then compares the accumulated hash with an existing known database. ZEKRA [36] proposes a cryptographic approach for CFA that enables outsourcing execution path verification to an untrusted worker via an arithmetic circuit, allowing verifiers to check program integrity in zero-knowledge without compromising privacy or scalability. MGC-FA [51] is a probabilistic model-based approach that identifies vulnerable sections within a program's control flow and utilizes execution-profiling control flow graphs for remote attestation. CFA+ [18] is a hardware-assisted mechanism that ensures runtime integrity by leveraging ARMv8.5-A's BTI. CFA mainly relies on trained models of execution paths or critical variables, and attacks within valid flows can bypass it. Exia performs verification with the EIM without needing to know the inputs in advance or keeping a copy of them.

Data Flow Attestation. DFA is designed to verify the integrity and security of data as it is transmitted and processed within systems. This method focuses on the path of data movement, how data is handled along this path, and any changes in its state, ensuring data protection. Raft [103] presents a hardware-assisted Dynamic Information Flow Tracking framework for RISC-V embedded systems to provide runtime protection with a hybrid byte and variable granularity storage mechanism. PrivacyScope [110] is a static code analyzer designed to detect private data leaks in TEE applications, effectively identifying both explicit and implicit information leaks through non-reversibility. DFA needs precise static analysis and either modifications to or instrumentation of the program. In contrast, Exia is transparent to the developers and does not need previous analysis of the source code.

Proof of Execution. Proof of Execution is a cryptographic protocol that verifies a specific computation or task has been accurately completed without revealing the underlying data or details of the process. Backes *et al.* [21] verifiably delegate computations on expanding outsourced data by introducing the first practical protocol that ensures security and input-independent efficiency, unlimited storage, and independence from the function computed. ALIBI [27] introduces a trusted, minimal reference monitor beneath the service provider's platform to track and report resource allocation to customer virtual machines, enabling verifiable reconciliation. APEX [73] is a formally verified security service for low-end embedded devices that enables remote, untrusted provers to generate tamper-proof proofs of software execution. Parma [57] is an architecture that enables secure, lift-and-shift deployment of unmodified containers by using VM-based TEEs and execution policies that ensure integrity and confidentiality, even when hosted on untrusted servers. Previous work primarily focused on streamlined instruction sets and low-end devices, whereas our approach concentrates more on integrity protection within the TEE. Additionally, some studies have introduced relatively high-overhead cryptographic technologies.

Integrity Measurement Architecture. The Linux Integrity Measurement Architecture (IMA) [79] hooks into the kernel to check files against TPM-protected measurements before they are accessed. TZ-IMA [92] introduces a framework that utilizes the secure world of TrustZone to verify an application's integrity before execution. Container-IMA utilizes Platform Configuration Registers to establish a hardware-based root of trust for the measurement partition of each container. XFilter [71] provides IMA with a fine-grained policy mechanism that supports extended attributes to reflect file configurations. Dimac [93] utilizes cross-world memory mapping in ARM TrustZone to perform integrity measurements on code pages executing within containers. In contrast to these works, which utilize IMA to track application files before and after execution, EXIA enables us to measure data in use directly rather than relying on file-based measurements via IMA.

Discovery of Memory Corruption Vulnerabilities. Discovering memory corruption vulnerabilities in TEEs mainly focuses on examining the host-enclave boundary. Van Bulck *et al.* [25] use manual analysis of the TEE runtimes, revealing that the entry points into the attack surface are more extensive than argument pointers. Automated tools are also an approach to the discovery. EnclaveFuzz [29] utilizes fuzzing to analyze source code by generating complex, valid inputs that pass sanity checks. TeeRex [33] analyzes enclave binary code with symbolic execution to discover vulnerabilities introduced at the host-to-enclave interface. While these papers primarily analyze the source code to detect whether it contains memory corruption attacks, EXIA focuses on detecting memory corruption attacks caused by malicious inputs during runtime.

XI. CONCLUSION

In this paper, we introduce External-Input Attestation to mitigate memory corruption attacks on enclaved applications by measuring all external writes to the enclaved application. This approach provides proof of trusted state transitions, effectively converting security threats into more manageable reliability issues, such as an application crash. Our prototype implementations on AMD SEV-SNP and Penglai are evaluated to validate the design, demonstrating robust security and practical performance.

ETHICAL CONSIDERATION

We acknowledge that our work involves the demonstration of proof-of-concept (PoC) attacks, which warrants careful ethical consideration. The purpose of these demonstrations is to highlight the practical severity of the threats and to provide a baseline for evaluating our proposed solution. It is crucial to state that all attacks presented are based on well-known attacks, and this paper does not disclose any new zero-day vulnerabilities. Since this research did not involve any human participants, it was exempt from further review by the Human Research Ethics Committee at the authors' institutions.

ACKNOWLEDGEMENT

We thank our anonymous reviewers for their insightful comments, which have significantly improved the paper. The work was partially supported by the National Natural Science Foundation of China under Grant No. 62472281, No. 62325207, and U24A20241. Yidi Kao and Sanchuan Chen were partially supported by Lambda, Inc.

REFERENCES

[1] SM3 cryptographic hash algorithm. http://www.gmbz.org.cn/upload/2018-07-24/1532401392982079739.pdf, 2018.

[2] KubeTEE trusted function framework. https://github.com/SOFAEnclave/trusted-function-framework, 2021.

[3] Chinook database. https://github.com/lerocha/chinook-database, 2025.

[4] LLM.c project. https://github.com/karpathy/llm.c, 2025.

[5] Mbed TLS library. https://github.com/Mbed-TLS/mbedtls, 2025.

[6] Mbedtls-SGX: a TLS stack in SGX. https://github.com/bl4ck5un/mbedtls-SGX, 2025.

[7] SGX_SQLite. https://github.com/yerzhan7/SGX_SQLite, 2025.

[8] SGXwallet: SKALE SGX-based hardware crypto wallet. https://github.com/skalenetwork/sgxwallet, 2025.

[9] Wolfssl embedded SSL/TLS library. https://github.com/wolfSSL/wolfssl, 2025.

[10] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 743–754. ACM, 2016.

[11] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[12] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. Veil: A protected services framework for confidential virtual machines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 378–393, 2023.

[13] Tiago Alves. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3:18–24, 2004.

[14] AMD. AMD64 architecture programmer's manual: Volumes 1–5. 2020.

[15] AMD. Secure VM Service Module (SVSM) for SEV-SNP Guests. https://github.com/AMDESE/linux-svsm, 2022.

[16] AMD. Linux kernel source code of SVSM host. https://github.com/AMDESE/linux/tree/svsm-preview-hv-v4, 2024.

[17] AMD. Guest Hypervisor Communication Block (GHCB) standardization. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf, 2025.

[18] Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu. On bridging the gap between control flow integrity and attestation schemes. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[19] Mahmoud Ammar, Adam Caulfield, and Ivan De Oliveira Nunes. SoK: Integrity, attestation, and auditing of program execution. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 77–77. IEEE Computer Society, 2024.

[20] Pierre-Louis Aublin, Florian Kelbert, Dan O'keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. TaLoS: Secure and transparent TLS termination inside SGX enclaves. *Imperial College London, Tech. Rep*, 5(2017):01, 2017.

[21] Michael Backes, Dario Fiore, and Raphael M. Reischuk. Verifiable delegation of computation on outsourced data. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013*, pages 863–874. ACM, 2013.

[22] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1213–1227. USENIX Association, 2018.

[23] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 30–40. ACM, 2011.

[24] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:

SGX cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[25] Jo Van Bulck, David F. Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1741–1758. ACM, 2019.

[26] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. page 769–784, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Chen Chen, Petros Maniatis, Adrian Perrig, Amit Vasudevan, and Vyas Sekar. Towards verifiable resource accounting for outsourced computation. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '13, Houston, TX, USA, March 16-17, 2013*, pages 167–178. ACM, 2013.

[28] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXpectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[29] Liheng Chen, Zheming Li, Zheyu Ma, Yuan Li, Baojian Chen, and Chao Zhang. Enclavefuzz: Finding vulnerabilities in SGX applications. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[30] Shuo Chen, Jun Xu, and Emre Can Sezer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium, USENIX Security 2005, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[31] Weijie Chen, Yu Zhao, Yinqian Zhang, Weizhong Qiang, Deqing Zou, and Hai Jin. Reminiscence: Trusted monitoring against privileged preemption side-channel attacks. In *European Symposium on Research in Computer Security*, pages 24–44. Springer, 2024.

[32] Tobias Cloosters, Oussama Draissi, Johannes Willbold, Thorsten Holz, and Lucas Davi. Memory corruption at the border of trusted execution. *IEEE Secur. Priv.*, 22(4):87–96, 2024.

[33] Tobias Cloosters, Michael Rodler, and Lucas Davi. Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 841–858. USENIX Association, 2020.

[34] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium, USENIX Security 2016, Austin, TX, USA, August 10-12, 2016*, pages 857–874. USENIX Association, 2016.

[35] Shujie Cui, Haohua Li, Yuanhong Li, Zhi Zhang, Lluís Vilanova, and Peter Pietzuch. Quanshield: Protecting against side-channels attacks using self-destructing enclaves. *arXiv preprint arXiv:2312.11796*, 2023.

[36] Heini Bergsson Debes, Edlira Dushku, Thanassis Giannetsos, and Ali Marandi. ZEKRA: zero-knowledge control-flow attestation. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIACCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, pages 357–371. ACM, 2023.

[37] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, page 106. ACM, 2018.

[38] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *29th USENIX Security Symposium, USENIX Security 2020*, pages 451–468, 2020.

[39] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. LO-FAT: low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pages 24:1–24:6. ACM, 2017.

[40] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Penglai Enclave TVM. https://github.com/Penglai-Enclave/Penglai-Enclave-TVM, 2021.

[41] Abbas A. Fairouz, Monther Abusultan, Viacheslav V. Fedorov, and Sunil P. Khatri. Hardware acceleration of hash operations in modern microprocessors. *IEEE Trans. Computers*, 70(9):1412–1426, 2021.

[42] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[43] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1231–1242. ACM, 2022.

[44] Virgil D. Gligor. A note on the denial-of-service problem. In *IEEE Symposium on Security and Privacy, SP 1983, Oakland, California, USA, April 25-27, 1983*, pages 139–149. IEEE Computer Society, 1983.

[45] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[46] Shay Gueron and Vlad Krasnov. Parallelizing message schedules to accelerate the computations of hash functions. *Journal of Cryptographic Engineering*, 2(4):241–253, 2012.

[47] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: system programming guide, Part*, 2(11):0–40, 2011.

[48] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, 2017.

[49] Stefan Hermann. *Accelerating minimal perfect hash function construction using GPU parallelization*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2023.

[50] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 969–986. IEEE Computer Society, 2016.

[51] Jianxing Hu, Dongdong Huo, Meilin Wang, Yazhe Wang, Yan Zhang, and Yu Li. A probability prediction based mutable control-flow attestation scheme on embedded platforms. In *18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 13th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, August 5-8, 2019*, pages 530–537. IEEE, 2019.

[52] Intel. Building the GNU multiple precision arithmetic library for Intel software guard extensions. https://github.com/intel/sgx-gmp-demo.

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

[54] Intel. Asynchronous enclave exit notify and the EDECCSSA user leaf function. https://cdrdv2-public.intel.com/736463/aex-notify-white-paper-public.pdf, 2022.

[55] Intel. Performance considerations of hardware-isolated partitioned VMs with Intel Trust Domain Extensions (Intel TDX). https://www.intel.cn/content/www/cn/zh/developer/articles/technical/tdx-performance-isolated-partitioned-vms.html, 2023.

[56] Intel. Intel software guard extensions SSL (SGX SSL). https://github.com/intel/intel-sgx-ssl, 2025.

[57] Matthew A Johnson, Stavros Volos, Ken Gordon, Sean T Allen, Christoph M Wintersteiger, Sylvan Clebsch, John Starks, and Manuel Costa. Parma: Confidential containers via attested execution policies. *arXiv preprint arXiv:2302.03976*, 2023.

[58] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, 2016. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[59] David Kaplan, Jeremy Powell, and Tom Woller. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. *White paper, Advanced Micro Devices Inc*, 2020.

[60] Andrej Karpathy. Char-rnn project. https://github.com/karpathy/char-rnn, 2015.

[61] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.

[62] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

[63] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[64] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium, USENIX Security 2020*, 2020.

[65] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium, USENIX Security 2017*, pages 523–539, Vancouver, BC, August 2017. USENIX Association.

[67] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security 2017*, pages 557–574, 2017.

[68] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 465–484. USENIX Association, 2022.

[69] LimeChain. Wasm_injector: a rust-based command line utility to manipulate webassembly (wasm) modules. https://github.com/LimeChain/wasm-injector.

[70] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[71] Alan Litchfield and Weihua Du. Xfilter: An extension of the integrity measurement architecture based on fine-grained policies. *Applied Sciences*, 13(10):6046, 2023.

[72] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1466–1482. IEEE, 2020.

[73] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 771–788. USENIX Association, 2020.

[74] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. DIALED: data integrity attestation for low-end embedded devices. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 313–318. IEEE, 2021.

[75] NVIDIA. Cloud computing solutions accelerated computing in the cloud. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/, 2025.

[76] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[77] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 2012.

[78] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. CoVE: Towards confidential computing on RISC-V platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 315–321, 2023.

[79] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium, USENIX Security 2004, August 9-13, 2004, San Diego, CA, USA*, pages 223–238. USENIX, 2004.

[80] Vasily A. Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rüdiger Kapitza. Stanlite - A database engine for secure data processing at rack-scale level. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, pages 23–33. IEEE Computer Society, 2018.

[81] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[82] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. Wesee: Using malicious #VC interrupts to break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4220–4238. IEEE, 2024.

[83] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. HECKLER: breaking confidential VMs with malicious interrupts. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[84] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[85] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with Intel SGX. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 177–196. Springer, 2019.

[86] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[87] Shiuan-Tzuo Shen, Hsiao-Ying Lin, and Wen-Guey Tzeng. An effective integrity check scheme for secure erasure code-based storage systems. *IEEE Transactions on reliability*, 64(3):840–851, 2015.

[88] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328, 2016.

[89] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A. Erdogdu, and Ross J. Anderson. Manipulating SGD with data ordering attacks. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 18021–18032, 2021.

[90] Argirios Sideris, Theodora Sanida, and Minas Dasygenis. Hardware acceleration of SHA-256 algorithm using NIOS-II processor. In *8th International Conference on Modern Circuits and Systems Technologies, MOCAST 2019, Thessaloniki, Greece, May 13-15, 2019*, pages 1–4. IEEE, 2019.

[91] Jason (Jay) Smith. Unlock inference-as-a-service with cloud run and vertex AI. https://cloud.google.com/blog/products/ai-machine-learning/improve-your-gen-ai-app-velocity-with-inference-as-a-service, 2025.

[92] Liantao Song, Yan Ding, Pan Dong, Yong Guo, and Chuang Wang. TZ-IMA: supporting integrity measurement for applications with ARM trustzone. In *Information and Communications Security - 24th International Conference, ICICS 2022, Canterbury, UK, September 5-8, 2022, Proceedings*, volume 13407 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 2022.

[93] Liantao Song, Yan Ding, Yong Guo, Bao Li, and Bin Zhou. Dimac: Dynamic integrity measurement architecture for containers with ARM trustzone. In *IEEE International Conference on Web Services, ICWS 2024, Shenzhen, China, July 7-13, 2024*, pages 844–852. IEEE, 2024.

[94] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. OAT: attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1433–1449. IEEE, 2020.

[95] Apache Teaclave. Teaclave SGX SDK. https://github.com/apache/teaclave-sgx-sdk/tree/master/samplecode/tls/tlsclient.

[96] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium, USENIX Security 2018*, 2018.

[97] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium, USENIX Security 2017*, pages 1041–1056, 2017.

[98] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy, SP 2019*, pages 88–105. IEEE, 2019.

[99] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2333–2350. ACM, 2019.

[100] Jiang Wang, Kun Sun, and Angelos Stavrou. A dependability analysis of hardware-assisted polling integrity checking systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[101] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl 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*, pages 2421–2434, 2017.

[102] Wenhao Wang, Linke Song, Benshan Mei, Shuang Liu, Shijun Zhao, Shoumeng Yan, XiaoFeng Wang, Dan Meng, and Rui Hou. The road to trust: Building enclaves within confidential VMs. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[103] Yu Wang, Jinting Wu, Haodong Zheng, Zhenyu Ning, Boyuan He, and Fengwei Zhang. Raft: Hardware-assisted dynamic information flow tracking for runtime protection on RISC-V. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*, pages 595–608. ACM, 2023.

[104] Haoxuan Xu, Jia Xiang, Zhen Huang, Guoxing Chen, Yan Meng, and Haojin Zhu. Latte: Layered attestation for portable enclaved applications. In *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025, Venice, Italy, June 30 - July 4, 2025*, pages 339–354. IEEE, 2025.

[105] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy, SP 2015*, pages 640–656. IEEE, 2015.

[106] Nikita Yadav and Vinod Ganapathy. Whole-program control-flow path attestation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2680–2694. ACM, 2023.

[107] HanJae Yoon and ManHee Lee. SGXDump: a repeatable code-reuse attack for extracting SGX enclave memory. *Applied Sciences*, 12(15):7655, 2022.

[108] Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana. Plinius: Secure and persistent machine learning model training. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 52–62. IEEE, 2021.

[109] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 270–282. ACM, 2016.

[110] Ruide Zhang, Ning Zhang, Assad Moini, Wenjing Lou, and Y. Thomas Hou. Privacyscope: Automatic analysis of private data leakage in tee-protected applications. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 34–44. IEEE, 2020.

[111] Xin Zheng, Xianghong Hu, Jinglong Zhang, Jian Yang, Shuting Cai, and Xiaoming Xiong. An efficient and low-power design of the SM3 hash algorithm for IoT. *Electronics*, 8(9):1033, 2019.

Appendix

A. Interrupt Category

These interruptions can be broadly categorized as follows:

- *External Interrupts*: These are generated in response to external events outside the processor. Typically, they are managed transparently by the corresponding OS/CPU without requiring explicit intervention.
- *Software Interrupts*: These are intentionally triggered by software to handle synchronous events, such as system calls or task switching.
- *Exceptions*: These arise due to errors during software execution and can be further subdivided into:
  1. *Faults*: Occur when an error condition is detected about the current instruction, often allowing for recovery and continuation of execution.
  2. *Traps*: Deliberate events initiated by the processor, often used for debugging or system monitoring.
  3. *Aborts*: Severe errors that typically result in the termination of the process or even a complete shutdown of the processor.

B. Architecture Defined Interrupts

Table IV lists a detailed taxonomy of architecture-defined interrupts, based on the AMD [14] and Intel [47] manuals. Each interrupt is labeled as `TEx`, `NTEx`, or $-$ (indicating that the enclaved application will abort). `TEx` does not affect the enclaved application's integrity since the handling of `TEx` (such as page swapping or handling page faults) does not cause the enclaved application to transit to another state. For hardware exceptions with vectors 0-8 (such as #`DE` and #`DB`), this halt signals a critical failure, likely stemming from invalid inputs or program bugs. A hang also occurs for exceptions in vectors 9-31, as these events are not anticipated during secure enclaved application execution. Hence, the handling of `TEx` can be modeled as a self transition (self-loop) (s, ϵ, s).

C. Detailed Taxonomy of System Call

The system calls we support, listed in Table V, fall into three main categories: file system and I/O, memory management, and networking. We handle these calls as two distinct event types. Those designated as `NTEx` are treated as a user input in system call format. Conversely, system calls marked with / are not restricted from direct use, as these operations can expose user memory directly to the enclaved application without measurement or protection, creating vulnerabilities to security threats such as TOCTOU attacks. To enhance security for these system calls, the host program can implement a more strict, though less efficient, procedure. This involves first transferring the external inputs in system call format into the attesting environment to be measured, and then instructing the

TABLE IV: A detailed taxonomy of architecture defined interrupts.

Category	Trans-parency	Vec-tor	Mne-monic	Description
External Interrupts		2	NMI	Non-Maskable Interrupt
				User Defined Interrupts
Software Interrupts	TEx			Task Switching
	NTEx			System Calls
Exceptions (Faults)	-	0	#DE	Divide Error
		1	#DB	Instruction Fetch Breakpoint, General Detect Condition
		5	#BR	Range Exceed Caused by BOUND Inst
		6	#UD	Invalid Opcode
		10	#TS	Invalid TSS
		12	#SS	Stack-segment Fault
		13	#GP	General Protection
		16	#MF	x87 FPU Floating-point Error
		17	#AC	Alignment Check
		19	#XM	SIMD Floating-point Exception
	TEx	7	#NM	Device not Available
		14	#PF	Page Fault
Exceptions (Traps)	TEx	1	#DB	Data Read or Write Breakpoint, I/O Read or Write Breakpoint, Single-step, Task-switch
		3	#BP	Breakpoint
		4	#OF	Overflow Caused by INTO Inst
Exceptions (Abort)	-	8	#DF	Double Fault
		9	-	Coprocessor Segment Overrun
		18	#MC	Machine Check
		11	#NP	Segment not Present

TABLE V: A detailed taxonomy of Linux system calls.

Category	Trans-parency	System Calls Examples
File System and I/O	NTEx	open(), openat(), close(), read(), write(), lseek(), stat(), unlink()
Memory Management	/	brk(), mmap(), munmap(), mprotect()
Networking	NTEx	socket()

enclaved application to use that input rather than direct input usage.