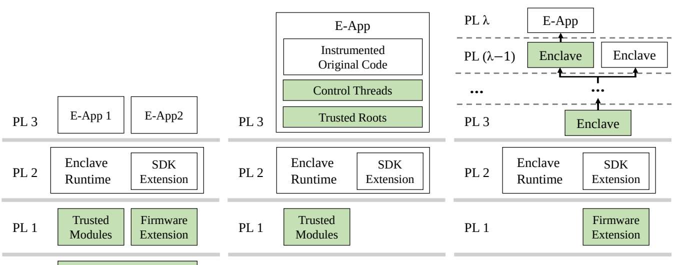
A Formal Approach to Multi-Layered Privileges for Enclaves

Ganxiang Yang, Chenyang Liu, Zhen Huang, Guoxing Chen*✉, Hongfei Fu, Yuanyuan Zhang, Haojin Zhu
Shanghai Jiao Tong University, China
{yangganxiang, zzshlcyy, xmhuangzhen, guoxingchen, jt002845, yyjess, zhu-hj}@sjtu.edu.cn

***Abstract*—Trusted Execution Environments (TEE) have been widely adopted as a protection approach for security-critical applications. Although feature extensions have been previously proposed to improve the usability of enclaves, their provision patterns are still confronted with security challenges. This paper presents PALANTÍR, a verifiable multi-layered inter-enclave privilege model for secure feature extensions to enclaves. Specifically, a parent-children inter-enclave relationship, with which a parent enclave is granted two privileged permissions, the Execution Control and Spatial Control, over its children enclaves to facilitate secure feature extensions, is introduced. Moreover, by enabling nesting parent-children relationships, PALANTÍR achieves multi-layered privileges (MLP) that allow feature extensions to be placed in various privilege layers following the Principle of Least Privilege. To prove the security of PALANTÍR, we verified that our privilege model does not break or weaken the security guarantees of enclaves by building and verifying a formal model named TAP$^{\infty}$. Furthermore, We implemented a prototype of PALANTÍR on PENGLAI, an open-sourced RISC-V TEE platform. The evaluation demonstrates the promising performance of PALANTÍR in runtime overhead $(< 5\%)$ and startup latencies.**



(a) Architecture-Level Extension (b) Intra-Enclave Compartmentalization (c) PALANTÍR Design

Fig. 1: Design patterns of feature extension to enclaves. Green-colored boxes indicate Trusted Computing Base (TCB) newly introduced by feature extensions for serving E-Apps. The term E-App stands for Enclave Application, and PL for Privilege Level. The symbol $\rightarrow$ indicates the parent-to-children relationship.

I. INTRODUCTION

As an evolving and promising security approach for protecting the users' sensitive data in programs, Trusted Execution Environment (TEE) [9], [17], [23], [32], [35], [37] has attracted numerous developers and researchers [8], [11], [47] to explore design patterns and applications on it [41], [42], [46], [50], [54], [58]. TEE introduces a secure area within the CPU, dubbed *enclave*, and protects the confidentiality and integrity of the code and data loaded within the enclave against even malicious privileged software like the operating system.

Excluding the privileged software, such as OS, from an enclave's TCB deprives the enclave of common features provided by privileged software, such as inter-process communication and virtual-machine-related operations, thus limiting the application of TEE. To address such deficiencies, plenty of diverse work has been proposed to bring various features back to TEE [26], [34], [61], [63], [64].

*Guoxing Chen is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241301
www.ndss-symposium.org

Due to the lack of a suitable privilege level to serve such extended features, existing work usually follows two design patterns, as shown in Fig. 1a and 1b. The first design pattern is *Architecture-Level Extension*. As shown in Fig. 1a, new features are enabled by modifying the architecture-level components, including machine mode and even the underlying hardware. For example, Cerberus [34] implemented its feature inside the secure monitor (located in the machine mode of the RISC-V architecture). SMILE [64] introduces a new trusted root located in x86 System Management Mode. However, this design monolithically integrated the feature into the privileged firmware, lacking fine-grained modularization, thereby violating the Principle of Least Privilege (PoLP) [48], which suggests that every module in a design should only access the resources and privileges that are necessary for its legitimate purpose.

The second design pattern is *Intra-Enclave Compartmentalization* (IEC). As shown in Fig. 1b, new features are supported via software-based compartmentalization within the enclave. For example, SGX-Migration [26] introduces a control thread inside an enclave for seamless live migration of the enclave. Zhao *et al.* [63] proposes multi-layer intra-enclave compartmentalization (MLIEC) to divide the enclave address space into multiple security layers to support enclave reuses. However, the security of this design pattern highly depends on correct and bug-free software implementation, *lacking architecture-level protection.* Recent work [5], [6], [7] has shown that code vulnerabilities in implementing software-based compartmentalization could be exploited to compromise the whole enclave.

Both of the feature extension patterns above exhibit security drawbacks, necessitating the proposal of a secure feature extension pattern. Ideally, from the perspective of the underlying platform, new features should be placed at a privilege level similar to existing enclaves, given that they solely serve these enclaves. From the perspective of existing enclaves, new features should be placed at a privilege level slightly higher than their privilege level to enjoy architecture-level protection. Furthermore, for complex features, multi-layered privileges are desired to better align with PoLP as different modules within a feature might require diverse accesses to resources and privileges. For example, the Reusable Enclave [63] feature proposed by Zhao *et al.* requires three levels of privileges, *i.e.*, a higher privilege level for nested attestation, a median privilege level for snapshot and rewinding, and a lower privilege level for executing user-provided applications.

A few studies have introduced inter-enclave and intra-enclave privilege-level separations. For example, Park *et al.* proposed Nested Enclave [44], which achieves hierarchical privilege isolation among enclaves. Specifically, it provides multiple inner enclaves sharing the same outer enclave through instruction extensions and meta-data modification, introducing two privilege levels to these enclaves. Melara *et al.* proposed EnclaveDom [38], which employs Memory Protection Keys (MPK) to implement intra-enclave memory tagging, realizing two privilege levels within an enclave. Although prior works demonstrate the feasibility and practicality of privilege separation, they do not support multi-layered privileges (more than two privilege levels) nor offer formal guarantees. Such limitations lead to the research problem: ***Can we design a verifiable multi-layered privileges model for enclave feature extensions following PoLP?***

In this paper, we propose PALANTÍR, a formally verified design of multi-layered privileges for enclaves. The key insight is that, given the existing TEE designs already ensure inter-enclave secure isolation, we can build multiple privilege layers among enclaves by introducing parent-children relationships. The parent enclave is authorized privileged permissions to access its children enclaves, such as read, pause, and resume. The software within the parent enclave could function as privileged features to its children enclaves. Meanwhile, from the point of view of the underlying platform and other privileged software, the parent enclave is no different from a legacy enclave (*legacy enclave*: an enclave having neither parent nor children enclaves). Notably, a children enclave can be further authorized privileged permissions and become the parent enclave of another enclave. Such nested privilege authorization facilitates multi-layered privileges among enclaves as shown in Fig. 1c.

PALANTÍR introduces two most common and critical privileged permissions needed for providing privileged features to enclaves, *i.e.*, *Spatial Control* and *Execution Control*, which seven primitives realize. Consequently, the PoLP is achieved in the sense that only seven primitives for supporting multi-layered privileges into the architecture level (to enjoy the architecture-level protection) while leaving the specific implementations of new features in necessary multiple privilege levels close to that of *legacy enclaves* .

We portray the security of PALANTÍR as follows: (1) the security of a *legacy enclave* or *root parent enclave* (an parent enclave that has no parent enclaves) will not be broken or weakened by any other enclaves, and (2) the security of a children enclave will not be broken or weakened by any other enclaves besides its parent enclave due to the two granted permissions.

We prove the security of PALANTÍR via formal verification. Based on an existing abstract formal model, TAP [52], we build TAP$^{\infty}$, an abstract formal model to formalize and integrate PALANTÍR on generic TEE platforms. Here λ denotes the number of privilege levels that can be formally verified and has an upper-bound $\lambda < \lambda_{\max}$. We introduced new security guarantee statements to retrofit our design on TAP$^{\infty}$, including the *Parent-Children Relationship Consistency* and *Exclusive Memory Consistency*. Note that the introduction of new privilege levels brings a significant model complexity increase from $\Omega(n^k)$ to $\Omega(n^{k^{\lambda_{\max}}})$, with n as the maximum process number the privileged software and parent enclaves can manage, and k as the solver-related parameter. To address the complexity explosion challenge, we utilize *Relevancy Propagation* and *Proposition De-skolemization* as optimization methods to accelerate formal reasoning of TAP$^{\infty}$. **By formal verification and inductive proofs, we guaranteed the security properties in TAP$^{\infty}$ with unlimited λ ($\lambda_{\max} = \infty$).**

Lastly, we implement a prototype of PALANTÍR based on a RISC-V TEE platform, PENGLAI [23], and provide two case studies, *i.e.*, *Reusable Enclave* and *Inter-Enclave Memory Sharing*, that fit cloud confidential computing scenarios. The evaluation includes the formal verification results and performance analysis.

In summary, this paper contributes as follows:

- It proposes PALANTÍR, a multi-layered privilege model on general enclave platforms that enables secure feature extensions with the least privilege.
- It builds TAP$^{\infty}$ to formalize the modified enclave platform model and to verify that it satisfies the *Secure Remote Execution* security property via automated formal verification.
- It implements the PALANTÍR prototype based on an open-sourced enclave platform and provides one case study, the *Hierarchical Deterministic Wallet*, for enclaves to demonstrate the framework usability.

II. BACKGROUND

In this paper, we concentrate on preserving a security property known as **Secure Remote Execution** (SRE), which encompasses the security aspects of remotely executing enclave programs, including previously mentioned Integrity and

Confidentiality. In the property's precondition assumptions, the remote platform provider is considered highly untrustworthy, capable of compromising privileged software and controlling the OS, hypervisor, or other vulnerable enclaves. By the Decomposition Theorem proved by Subramanyan *et al.* [52], an enclave platform that satisfies the triad of the properties below for any enclave program running on it also satisfies the SRE. So, we set our security guarantee target to prove the three properties.

- **Secure Measurement:** The *measurement* of an enclave should represent the enclave's initial state and can be used for enclave state integrity checking. Moreover, the measurement and the outside input sequence together uniquely determine the execution of the enclave.
- **Integrity:** Each enclave executing on a remote platform is tamper-resistant concerning program execution.
- **Confidentiality:** Each enclave's sensitive data will not be revealed to any untrusted entity, including malicious platform providers, untrusted privileged software, and other compromised enclaves or processes.

The SRE property above is satisfied by general enclave platforms, which have been proved in the Trusted Abstract Platform (TAP) [52]. Furthermore, multiple real-world TEE platform implementations have been refined to TAP model, including the Intel SGX [30] and the MIT Sanctum [17], which shows the comprehensiveness of TAP by proving the SRE property are satisfied on these distinct state-of-the-art TEE platforms. Recently, Lee *et al.* [34] also proposed TAP$_C$, a cross-platform memory-sharing primitive based on TAP, and implemented it on Keystone [35]. In Sec. IX, we perform a comparative analysis, elucidating the significant distinctions between our TAP$^\infty$ and previous works based on TAP.

This paper verifies the multi-layered privileges model across general enclave platforms. Indeed, we draw foundational inspiration from the TAP abstract formal model to develop our TAP$^\infty$. As such, verification at the binary- or instruction-level (*e.g.* Serval [40]) is not a primary objective of this study. However, these approaches remain valuable as they can verify that a given implementation accurately refines our model at the binary level.

III. Threat Model

Our design follows the typical TEE threat model, in which a user $\mathcal{U}$ deploys her enclaves onto an untrusted platform in the presence of adversarial $\mathcal{A}$. The adversary $\mathcal{A}$ has the software privileges and may launch various software attacks to exploit software vulnerabilities and destroy the communication between the user and her enclaves at will. The integrity and confidentiality of these enclaves are protected against any software adversary running in the remote enclave platform. And we do not consider these enclaves vulnerable or malicious by themselves.

With Palantír, $\mathcal{U}$ can firstly launch parent enclaves to set up desired features. Subsequently, her other enclaves are recursively governed by launched parent enclaves and become their children enclaves. As illustrated in Fig. 2, the features available to her enclave e equals all features provided by e's ancestor enclaves (*i.e.* Enclave 1 and Enclave 2). The remote parent enclaves could either be developed by the user $\mathcal{U}$ herself or developed and open-sourced by platform developers for public verifiability. Also, to minimize the TCB, the users are encouraged to launch only parent enclaves that provide necessary features. Notably, the Palantír framework does not introduce any additional entities to be trusted. Instead, each used parent enclave can be developed by the user herself as a feature provider and the coordinator for her enclaves. Regarding security guarantees, we formally verified the primitives of parent enclaves, ensuring it remains unaffected by any compromised enclaves, including its own children enclaves, other non-ancestor children enclaves and their children enclaves. We also verified that the security guarantees of any legacy enclave also remain unaffected by any other compromised parent enclave and their children.

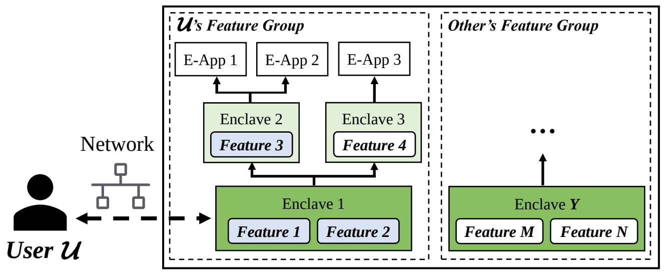
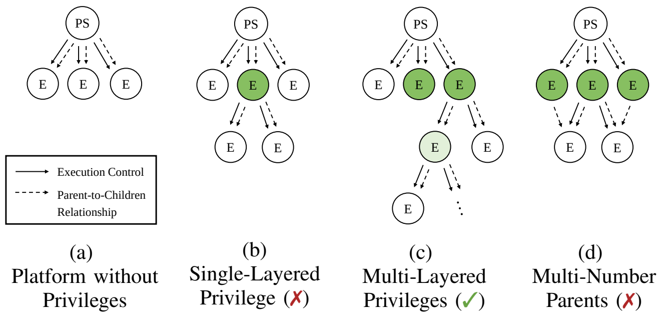


Fig. 2: Multi-layered privilege model. Green-colored boxes indicate privilege layers. Deeper shades represent higher privilege levels. Blue-colored boxes indicate features the E-App 1 can access. The symbol ($\rightarrow$) indicates the parent-to-children relationship.

Since our primary goal is to design a generic feature extension pattern, we do not consider any hardware-level side-channel attacks [56], [39], [49], [55] and leave side-channel resilient interface design as future work. Similar to previous work on formally modeling generic TEE platforms [34], [52], we consider program-specific or TEE-specific defenses [16], [24], [36], [57], [62] to be orthogonal to our work, as they could be ported to our platform-specific prototype implementation. Denial-of-service attacks are also kept out-of-scope to be consistent with the threat models for existing state-of-the-art enclave platforms.

IV. Design Choices

In Sec. III, we explain how features for children enclaves are facilitated by their parent enclave and ancestors. We make specific design choices to support parent-children enclave relationships to ensure verifiable multi-layered privileges over enclaves. This section delves into the details of our design decisions concerning the model and interfaces, which are crucial for modeling, verification, and implementation.



PS
E
Execution Control
Parent-to-Children Relationship
(a) Platform without Privileges
(b) Single-Layered Privilege (✗)
(c) Multi-Layered Privileges (✓)
(d) Multi-Number Parents (✗)

Fig. 3: Inter-enclave privilege models. Green-colored nodes indicate privilege layers. Deeper shades represent higher privilege levels. The term E stands for Enclave and PS for Privileged Software. Symbol ✓ indicates the design accepted by PALANTÍR while ✗ indicates those not accepted.

TABLE I: Spatial isolation permission.

Spatial Isolation Permissions	**Children Enclave Components**			
	Code	**Registers**	**Stack Frame**	**Others**
read-enabled (✓)	*R*	*R*	*R*	*R*
write-enabled (✗)	*R*	*RW*	*RW*	*RW*
execute-enabled (✗)	*RX*	*R*	*R*	*R*

[1] $R/W/X$ indicates *read/write/execution* permissions.
[2] The definition of symbols ✓ and ✗ is the same with that in Fig. 3.
[3] Other sections include the .bss and .data sections.

A. Key Privileges of Parent Enclave

We identified the two most critical inter-enclave privileges necessary for feature extension:

Execution Control. The Execution Control privilege enables a parent enclave to (1) arbitrarily launch a fresh enclave as its children enclave and (2) control its children enclave's life cycle, which includes starting and stopping its execution, attestation, *etc*. PALANTÍR is among the first few enclave models allow Execution Control among enclaves, specifically designed to enable parent enclave to establish an efficient communication and feature provision channel with its children enclaves.

Spatial Control. The Spatial Control privilege allows any contents of a children enclave to be unilaterally shared with its creator parent enclave at any moment. This Spatial Control scheme significantly expands the range and diversity of features that can be developed within parent enclaves by utilizing the feature extension channel.

Remark. With consideration of convenience, in our description below, we denote a parent enclave as PE. The children enclaves governed by a PE are denoted as CEs, as the feature recipients of the corresponding PE. Noting that a CE can be either privileged or non-privileged, indicating that a CE can also be a PE of other CEs.

B. Inter-Enclave Privilege Models

Upon granting Execution Control to PEs, we have effectively established one or more privilege layers between the privileged software (such as OS and Secure Monitor) and non-privileged enclaves. Comparative observation revealed that, unlike the *monolithic non-privileged* model in Fig. 3a and the *single-layered privilege separation* model in Fig. 3b, the *multi-layered privileges* (MLP) model in Fig. 3c brought significant security benefits to users.

First, the MLP adheres to the security principles of modularity and least privilege (PoLP) [48]. Specifically, the MLP design enables fine-grained privilege level separation among distinct modules inside one feature. For instance, in the Reusable Enclave and Live Memory Introspection feature extensions for enclaves [63], [64], the remote attestation modules for authentication should be assigned the highest privilege level. In contrast, other data marshaling functions do not require such privileges and can be designated to lower levels. By this privilege separation approach, each module will be positioned in its least privilege and spatially securely isolated. Second, the MLP reduces the Trusted Computing Base (TCB) through fine-grained feature positioning. As depicted in Fig. 2, the TCB burden for an enclave is confined to the firmware and features located in its ancestors. In contrast, the monolithic non-privileged model integrates all features within the firmware, leading to unnecessary TCB burdens from unused features.

C. Details of Privilege Permissions

In Sec. IV-A and Sec. IV-B, we identified and briefly introduced the distinct privileges of PEs that differentiate them from legacy enclaves. This section will delve into the detailed permissions of multi-layered privileges (MLP), inter-enclave parent relationships, and spatial control. Our discussion aims to construct a verifiable multi-layered privileges model.

1) Maximum Number of Privilege Layers: Although MLP offers both security and performance benefits, it also introduces considerable challenges in verification complexity. Denoting the number of privilege layers by λ, a seemingly simple approach to enable the MLP is to impose no restrictions on λ, allowing $\lambda \in \mathbb{N}_+$. However, this strategy could lead SMT solvers to produce inconclusive proofs. This issue arises due to excessive model complexity and the inability to complete inductions. Denote n as the maximum process number the privileged software and parent enclaves can manage and k as the solver-related parameter. The model complexity of TAP$^\infty$ experiences a double-exponential magnitude state explosion from that of TAP's $\Omega(n^k)$ to ours $\Omega(n^{k^{\lambda_{\max}}})$. We give detailed complexity analysis in Sec. VI-A and Appendix B.

Given the complexity explosion caused by λ, we tried to seek a reasonable layer range as $\lambda \in \mathbb{N}_\lambda = \{n \in \mathbb{N} \mid 1 \leq n < \lambda_{\max}\}$, to support the feature extension benefits discussed above and be verified successfully in an acceptable time. For additional information on determining a suitable $\lambda_{\max}$ and proving security properties under unlimited layer ($\lambda_{\max} = \infty$), refer to Sec. VIII-A.

2) Inter-Enclave Parent Relationship: In the inter-enclave privilege models previously discussed in Sec. IV-B, each enclave was configured to have only one unique parent enclave, namely its creator. However, the parent-children relationships can be expanded to a more general lattice model by incorporating the *multi-number parents* (MNP) design, as shown in Fig. 3d. This adaptation allows each enclave to recognize multiple privileged enclaves as its parents, thereby gaining access to all features provided by all of its parents.

MNP and MLP are both effective designs for delivering multiple features to an enclave. Notably, the capability to deliver multiple features of an MLP mode with a maximum of $\lambda_{\max} = N$ layers surpasses that of an MNP model with a maximum of $\mu_{\max} = N$ parents. Although both models can access up to N features, only MLP allows for inter-feature privilege separations, while MNP cannot. Moreover, the MNP design encounters a significant security challenge in maintaining *Confidentiality*. This issue arises from the Execution Control permission within the MNP model. In MNP, each enclave e can claim multiple parent enclaves as its parents, However, only one of these parents can be granted the Execution Control permission to start up e. In the proof of *Confidentiality*, consider two PEs ($\mathcal{P}_1[e]$ and $\mathcal{P}_2[e]$) both identified as the parents of an enclave e, where $\mathcal{P}_1[e]$ is permitted to boot e while $\mathcal{P}_2[e]$ is not. The adversary can easily distinguish the two enclaves by observing the differences in their state transition: the execution flow from the $\mathcal{P}_1[e]$ will involve a context switch to e, whereas the execution of the $\mathcal{P}_2[e]$ will not. However, proving *Confidentiality* requires that adversaries cannot distinguish between the trace of two PEs, which is impossible for the MNP model.

In conclusion, the multi-number parents (MNP) model offers feature delivery capabilities comparable to MLP but violates the *Confidentiality* requirement in SRE. Consequently, we exclude the MNP from our inter-enclave privilege model.

3) Spatial Control Permission: Table I shows a privilege permission list of Spatial Control. For the *read-enabled* privilege, the permission details are shown in the first row, where PE is allowed to load runtime contents from its CE. For the *write-enabled* privilege, the PE is also allowed to write data to CE execution states and data segments. Lastly, for the *execute-enabled* privileged enclave, it should also be permitted to execute a chunk of CE's code.

To maintain the enclave's security property and avoid significant conflicts between our prototype design and threat model, we restrict the Spatial Control permission in *read-enabled* grade. First, the *write-enabled* PE design directly conflicts with the *Confidentiality* proof requirement elaborated in Sec. VI, which caused a great challenge for us in finishing the formal verification part. In detail, given two PE traces with one writing data to its CE while another does not, the adversary can find the difference between them since the CE contents could be observed by the adversary, thus corrupting the *Confidentiality*. Second, the *execution-enabled* design directly corrupts the code integrity of PE. An adversary may inject attack code into a compromised CE. When the malicious CE code is executed by its victim PE, the whole PE might be hijacked and compromised. To solve this, we need a code verifier to verify any untrusted *to-be-executed* code in CE, like the eBPF [10] verifier in the Linux kernel. Taking a step back, although code verifiers are supported by mathematical proofs [21], [27], [45], their implementations are suffered from security vulnerabilities. For instance, CVE-2021-31440 [3] is a famous verifier-related vulnerability in which the attacker exploits the bound checking bug inside the original code of the eBPF verifier and bypasses it to inject malicious eBPF code inside kernel.

TABLE II: State Transition Set $\mathcal{G}$.

Operation	Description
`LAUNCH`	Create an enclave and becomes its parent enclave.
`ENTER`	Enter a children enclave from its entry point and execute.
`PAUSE`	Mark the current enclave as *stopped*. Back to its parent enclave.
`RESUME`	Resume execution of a children enclave from the stop point. Unmark *stopped* state of the enclave.
`EXIT`	Mark the current enclave as *exited*. Back to its parent enclave.
`DESTROY`	Destroy a children enclave.
`INSPECT`†	Do memory or context inspection on a children enclave.
`COMPUTE`	Do `LOAD`, `STORE`, and `EXECUTE` as a program.

[1] † symbol indicates newly introduced interfaces while other interfaces are modified to support PALANTÍR design.
[2] `COMPUTE` operation simulates the instruction execution in process executables.

Given the security concerns above, we adopt a conserved *read-enabled* parent enclave model as shown in the second row of Table I. To keep good isolation from a parent enclave and other enclaves that are not created by it, we further restrict the parent enclave's *read-only* range within its own children enclaves.

Remark. To sum up, we choose the *multi-layered privileges* (MLP) with a maximum of $< \lambda_{\max}$ privilege layers as our privilege separation model, with each layer having the *Execution Control* and *Spatial Control* privilege over its next-lower-layered enclaves.

D. Interface

Enclave programs require interfaces to support our inter-enclave privilege model discussed in the sections above. PALANTÍR utilizes explicit operations to facilitate the model, incorporating both existing primitives for general TEE platforms and newly introduced ones. Below, we detail the interface requirements for each privilege associated with parent enclave. For a comprehensive summary of these interfaces, refer to Table II.

Execution Control. Existing TEE platforms [17], [30], [52] allow privileged software to control the life cycles of enclaves through six operations: `LAUNCH`, `ENTER`, `PAUSE`, `RESUME`, `EXIT`, and `DESTROY`. In PALANTÍR, we grant PE these operations over its CE to support the Execution Control feature. Additionally, we modify the detailed semantics of these operations to align with parent-enclave relationship requirements in Sec. IV-C.

TABLE III: TAP$^\infty$ State Variables.

Symbols	Type	Description
pc	VA	The program counter.
$regs$	$\mathbb{N} \to$ W	General Purpose Registers.
Π	PA $\to$ W	The abstract physical memory region.
$curr$	ID	Current process eid.
own	PA $\to$ ID	Map from physical address to the enclave that owns it.
$\mathcal{M}$	(ID $\times$ VA) $\to$ (ACL $\times$ PA)	Page table mapping from process virtual address to permission bit and physical address.
$\mathcal{C}$	(SET $\times$ WAY) $\to$ (BOOL $\times$ TAG)	The abstract cache design with valid bit and tags.
$\mathcal{D}$†	ID $\to$ META	Map from process eid to metadata record.
$\mathcal{P}$†	ID $\to$ ID	Map from process eid to its parent's eid.

[1] † indicates newly proposed or updated state variables in TAP$^\infty$.

TABLE IV: Records of TAP$^\infty$ Process Metadata META.

Symbols	Type	Description
$\mathcal{D}^{EP}$	VA	The enclave's entrypoint.
$\mathcal{D}^{AM}$	VA $\to$ PA	The enclave's virtual address map.
$\mathcal{D}^{AP}$	VA $\to$ ACL	The enclave's virtual address permissions (*read/write/execute*).
$\mathcal{D}^{EV}$	VA $\to$ BOOL	Set of private virtual addresses.
$\mathcal{D}^{pc}$	VA	The enclave's saved program counter.
$\mathcal{D}^{regs}$	$\mathbb{N} \to$ W	The enclave's saved registers.
$\mathcal{D}^{paused}$	BOOL	Whether the enclave is paused.
$\mathcal{D}^{privil}$†	BOOL	Whether the enclave is privileged.
$\mathcal{D}^{parent}$†	ID	The enclave's parent enclave.

[1] † indicates additional records added to support multi-layered privileges (MLP).

Spatial Control. To enable the *read-enabled* permission outlined in Table I over CE runtime contents, we introduce one new enclave operation, INSPECT. The valid targets of INSPECT for a PE are limited to its children only.

Multi-Layered Privileges. MLP enables multiple layers of parent enclave, requiring semantic support at the enclave creation stage. Therefore, the only operation modified to support the MLP design is the LAUNCH, which is adjusted to allow a PE to create a new CE that could potentially serve as a PE in the future.

The subsequent sections discuss our formal model and interfaces.

V. Formal Model

We first give an overview of our TAP$^\infty$ formal model in Sec. V-A and then provide detailed specifications in Sec. V-B.

A. Formal Model Overview

1) Platform and Process State: An abstract enclave execution platform is one transition system $TS = \langle \mathrm{S}, I_0, \rightsquigarrow \rangle$ that represents the whole platform during the life of a user's enclave. S is the set of all TS' states σ. The platform always starts from an initial state $\sigma_0 \in I_0$, in which I_0 is the set of all legal initial states, and performs each state transition by applying a transition function $\rightsquigarrow \subset \mathrm{S} \times \mathrm{S}$. Furthermore, the set of all possible execution traces of the transition system TS in TAP$^\infty$ is denoted as $TRACE(TS) \subset \mathrm{S}^\omega$, which means all legal transition sequences. Almost all these symbols are also widely adopted by TAP [52] and TAP$_C$ [34].

Given the symbols above, we can derive the formal format of an enclave platform. For each possible execution trace π, we have $\forall \pi = \langle \sigma_0 ... \sigma_n ... \rangle \in TRACE(TS)$. $\forall i \in \mathbb{N}$. $\sigma_i \in \mathrm{S}$ and $\langle \sigma_i, \sigma_{i+1} \rangle \in \rightsquigarrow$. For each transition, the current state will take an operation from operation set $\mathcal{G}$ in Table II. Detailed state transition specification refers to Appendix A. For the initial state of an enclave e, we use $init_e(\sigma) : \mathrm{S} \to \mathrm{BOOL}$ to denote whether state σ includes the initial state of the launched e. Besides, we use several symbols interchangeably to simplify expressions. For process execution trace π, $\pi^{\langle i \rangle} \equiv \sigma^i$. Next, we describe state variables and process metadata fields for defining interfaces and proving security guarantees.

2) TAP$^\infty$ State Variables: For reference, all state variables in our TAP$^\infty$ model are listed in the Table III. For convenience, we adopt original symbols from TAP. Particularly, We use $v : T$ to denote that variable $v \in V$ has type T and $L \to R$ to denote the type of a map with index type L and value type R. The symbols pc : VA, $regs : \mathbb{N} \to$ W, and Π : PA $\to$ W are abstractions of program counters, registers, and memory with usual meanings. To be exact, VA and PA are fixed length bit vectors representing *virtual address* and *physical address* respectively, and W is *word*, which is also a fixed length bit vector as the minimal data unit. By maintaining a map from each physical address to a process id in the id set ID $= \{e_i \mid i \in \mathbb{N}\} \cup \{\mathbf{ps}\}$, the symbol own : PA $\to$ ID records each physical address p's owner process as $own[p]$. Notably, we use $\mathbf{ps}$ to represent the untrusted privileged software.

In our TAP$^\infty$, we propose two brand-new metadata fields to reprove SRE. First, to support the parent enclave and MLP, we add a new $\mathcal{P}$ map to record each enclave's parent relationship. $\mathcal{P}[e_2] = e_1$ means e_1 has successfully performed LAUNCH (e_2) once before and e_1 became the parent of e_2. Second, we introduce $\mathcal{D}$: ID $\to$ META as a map from each process id to its metadata typed META to describe its comprehensive runtime information. Notably, the $\mathcal{D}$ record can represent not only the state of enclaves but also that of the adversarial privileged software because the ID set includes privileged software ($\mathbf{ps}$) and other compromised enclaves. The detailed fields of record type META are listed in Table IV.

In the records of META, $\mathcal{D}_e^{AM}$ is frequently used in our proof to represent the abstract page table of e. And the $\mathcal{D}_e^{EV}$ is used to mark each virtual address v is exclusively owned by e or shared with others. Note that we use $\mathcal{D}_e^{EV}$ as an equivalent abbreviation of $\mathcal{D}[e].\mathcal{D}^{EV}$ here. For more abbreviation details, refer to Sec. V-A3.

3) Process Metadata Variables: To support MLP and parent enclave, we introduce two metadata fields in the process's record $\mathcal{D}$ metadata. The first is $\mathcal{D}^{privil}$: BOOL, which is **True** if the record $\mathcal{D}$'s owner process e has the privileges in Sec. IV-A and **False** otherwise. The second field is $\mathcal{D}^{parent}$: ID, whose value is the id of its parent enclave and should be synchronized with the result of global metadata $\mathcal{P}[e]$ for each enclave $e \in$ ID.

To simplify our equation, we ignore state symbols when we access TAP$^\infty$ state variables if only one state symbol (*e.g.* σ) occurs in an equation. That is

TABLE V: Assistant Notations and Functions in TAP$^\infty$.

Symbols	Type	Description
$shared$	PA $\to$ BOOL	Whether the physical address is shared.
$active$	ID $\to$ BOOL	Whether the enclave is executable.
$init$	S $\to$ BOOL	Whether the transition state represents the enclave's initial state.
obs	S $\to$ META	The process information that can be observed by the privileged software.
$ancestor$	$\big((\text{ID} \to \text{ID}) \times \text{ID} \times (\{0\} \cup \mathbb{N}_\lambda)\big) \to \text{ID}$	Calculate the k-distant ancestor of any enclave.
$valid\_memory\_layout$	ID $\to$ BOOL	Whether the enclave's memory layout is ready for `LAUNCH`.
$layer\_depth$	$\big((\text{ID} \to \text{ID}) \times \text{ID}\big) \to \mathbb{N}$	Calculate the number of privilege layers from an enclave to its root PE.

$$\forall \sigma.\ \mathbf{ps} = \sigma.curr \iff \forall \sigma.\ \mathbf{ps} = curr$$

Based on the abbreviation above, we next simplify *nested member accessing* operation on the process metadata $\mathcal{D}$. For example, in all equations with only one state symbol (*e.g.* σ), we have

$$\sigma.\mathcal{D}[e] \iff \mathcal{D}[e] \iff \mathcal{D}_e$$

and

$$\sigma.\mathcal{D}[e].\mathcal{D}^{foo} \iff \mathcal{D}_e.\mathcal{D}^{foo} \iff \mathcal{D}_e^{foo}$$

All equations follow these abbreviations in our paper.

4) Enclave Inputs and Outputs: For input and output for target enclave e, we follow TAP to use $I_e(\sigma)$ and $O_e(\sigma)$ to represent enclave e's input and output contents under current model state σ, respectively. I_e includes arguments of the transition operation $op \in \mathcal{G}$ and any memory region outside of e that is reachable for e excluding its own CE contents, if exist. And O_e includes the output of enclave e accessible for other enclaves and adversary OS under current state σ. Notably, we assume sanitized I_e with help from orthogonal toolchains [25], [62] to eliminate potential code-pattern vulnerability exploitations.

5) Formal Adversary Model: To model an adversary with privileged software in our threat model, we use $\mathbf{ps} \in$ ID to denote the privileged software, such as an OS or hypervisor, which the adversary directly controls. As a privileged process, it has the permission to `LAUNCH` any typed enclaves and `ENTER` them. Moreover, the untrusted adversaries have exclusive states that the enclave cannot access, such as other compromised enclaves or hypervisor private memory regions. For a target enclave e, the exclusive states of an adversary are denoted as $A_e(\sigma)$: S $\to$ META, which is a mapping from current transition system TS's state $\sigma \in$ S to the adversarial record $\mathcal{D} \in$ META. Similar with $A_e(\sigma)$, $E_e(\sigma)$: S $\to$ META is denoted as the enclave's private states $\mathcal{D} \in$ META under current TS state $\sigma \in$ S.

B. Interface Specifications

This section elaborates on the formal interface specifications of parent enclaves, as proposed in Sec. IV, distinguishing PALANTÍR and TAP$^\infty$ from previous work. It specifically describes the prerequisite conditions for each enclave operation listed in Table II, which must be met by the transition system TS before their execution. Additionally, we employ several assistant symbols and abstract functions to enhance the clarity of the specifications and theorems, detailed in Table V.

1) Execution Control and Multi-Layered Privileges: To achieve Execution Control and Multi-Layered Privileges (MLP), we revised the semantics of all existing enclave operations, as detailed in Sec. IV-D.

Initially, we modified the `LAUNCH` to allow any PE to create its own CEs. When an enclave calls an operation in $\mathcal{G}$, a list of parameters will be filled and passed to the platform, including enclave ID, address mapping, exclusive memory region, *etc.* Specifically, we added a δ parameter to denote whether the to-be-created enclave is privileged (PE) or not. When the transition system TS on state σ calls the $\texttt{LAUNCH}(e_{id}, \dots, \delta)$ with target created enclave id as e_{id} and privileged tag as δ, several pre-conditions must be held to prevent malicious usage of `LAUNCH`, as discussed below.

First, the target *to-be-created* enclave e_{id} should not be in-use, namely$\big(\neg active(e_{id})\big)$. Also, the target e_{id} must have an appropriate memory layout, with private data strictly protected from adversarial access, as depicted by $valid\_memory\_layout(e_{id})$. To retrofit Execution Control, we allowed not only adversary $\big(curr = \mathbf{ps}\big)$ but also privileged parent enclaves $\big(\mathcal{D}_{curr}^{privil}\big)$ to create enclaves. Following the MLP constraints in Sec. IV-C1, the maximum number of privilege layers cannot exceed the upper-bound $\lambda_{\max}$, which is also defined in Sec. IV-C1. Consequently, the MLP constraint of `LAUNCH` should be $\big((layer\_depth(\mathcal{P}, e_{id}) = \lambda_{\max}) \Rightarrow \neg\delta\big)$. Finally, the comprehensive form of the pre-condition for `LAUNCH` is formally presented as follows:

$$(\neg active(e_{id})) \wedge valid\_memory\_layout(e_{id}) \ \wedge \qquad (1)$$
$$\Big((curr = \mathbf{ps}) \vee (\mathcal{D}_{curr}^{privil} \Rightarrow (layer\_depth(\mathcal{P}, e_{id}) = \lambda_{\max}) \Rightarrow \neg\delta)\Big)$$

We update `DESTROY` semantics to prevent the adversary from directly destroying in-use PEs, thus avoiding triggering an inactive-parent situation in a running CE has a destroyed PE as its parent. In other words, a PE can be destroyed only if it all of its CEs has finished execution and exited normally, namely $\big(\forall e \in \text{ID}.\ active(e) \Rightarrow \mathcal{P}[e] \neq e_{id}\big)$. The rewritten pre-condition requirement for $\texttt{DESTROY}(e_{id})$ is below.

$$active(e_{id}) \wedge \big(\mathcal{P}[e_{id}] = curr\big) \ \wedge$$
$$\Big(\forall e \in \text{ID}.\ active(e) \implies \mathcal{P}[e] \neq e_{id}\Big)$$

Considering PE's features, we also put constraints on `ENTER` and `EXIT`. By our definition, after `EXIT`, the execution trace will return to $curr$'s parent enclave, while `EXIT` called by adversarial PS does nothing. The $\texttt{RESUME}(e_{id})$ follows similar semantics with `ENTER`: switches execution context to e_{id} and additionally marks e_{id} as *non-stopped*. Conversely, the $\texttt{PAUSE}(e_{id})$ operation mirrors `EXIT` by switching execution context out of e_{id} and additionally marking e_{id} as *stopped*. Given the semantic similarity above, we don't make duplicated elaborations on them.

$$\begin{tikzcd}
\pi_1^{\langle 0\rangle} \arrow[r, "eop_0"] \arrow[d, phantom, "\approx I \approx E"] & \cdots \pi_1^{\langle i\rangle} \arrow[r, "\mathcal{A}_1"] \arrow[d, phantom, "\approx I"] & \pi_1^{\langle i+1\rangle} \arrow[r, "eop_1"] \arrow[d, phantom, "\approx I \;\; \approx E, \approx O"] & \pi_1^{\langle i+2\rangle} \arrow[r, phantom, "\cdots"] \arrow[d, phantom, "\approx I \;\; \approx E, \approx O"] & \pi_1^{\langle j\rangle} \arrow[r, "\mathcal{A}_1"] \arrow[d, phantom, "\approx I \;\; \approx E, \approx O"] & \pi_1^{\langle j+1\rangle} \arrow[r, "eop_2"] \arrow[d, phantom, "\approx I"] & \cdots \\
\pi_2^{\langle 0\rangle} \arrow[r, "eop_0"] & \cdots \pi_2^{\langle i\rangle} \arrow[r, "\mathcal{A}_2"] & \pi_2^{\langle i+1\rangle} \arrow[r, "eop_1"] & \pi_2^{\langle i+2\rangle} \arrow[r, phantom, "\cdots"] & \pi_2^{\langle j\rangle} \arrow[r, "\mathcal{A}_2"] & \pi_2^{\langle j+1\rangle} \arrow[r, "eop_2"] & \cdots
\end{tikzcd}$$

Fig. 4: Proving Integrity in TAP$^\infty$. Here, $\approx X$ denotes that the symbol variable X of two state machines is equal after the same steps. For example, $\approx E$ on state σ_1, σ_2 means the enclave-related states are equivalent, denoted as $E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2)$. **Blue**-colored symbols indicate conditions that the proof provides to systems, like $\boldsymbol{I}$ for inputs. **Green**-colored symbols indicate constraints that the system must obligate, like $\boldsymbol{O}$ for adversary-observable outputs and $\boldsymbol{E}$ for enclave states. Adversarial operations $\boldsymbol{\mathcal{A}} \in \rightsquigarrow$ are marked **red**.

Following the design in Sec. IV-D, the constraint of ENTER(e_{id}) is shown below, which means any adversary or PE can only ENTER into its own CE.

$$active(e_{id}) \wedge \big(\mathcal{P}[e_{id}] = curr\big) \;\wedge\; \big(curr = \mathbf{ps} \vee \mathcal{D}^{privil}_{curr}\big)$$

2) Spatial Control: The Spatial Control feature is achieved by adding a new INSPECT operation to PE. We give the specification of INSPECT(e_{id}, v) as below, in which $e_{id} \in$ ID means the *to-be-inspected* CE and $v \in$ VA means the inspected virtual address.

$$\begin{aligned}
&active(e_{id}) \wedge \big(\mathcal{P}[e_{id}] = curr\big) \wedge \mathcal{D}^{privil}_{curr} \wedge \\
&\Big(\exists p \in \text{PA}. \big(\mathcal{D}^{AM}_{e_{id}}[v] = p\big) \wedge \big(own[p] = \mathbf{ps} \vee own(p) = e_{id}\big)\Big)
\end{aligned}$$

In the equation above, $\mathcal{D}^{AM}_{e_{id}}[v] = p$ ensures the v is properly mapped by e_{id}'s page table. And $\big(own(p) = \mathbf{ps} \vee own(p) = e_{id}\big)$ ensures the target inspection address is either a shared memory region represented by $own(p) = \mathbf{ps}$ or is an exclusive region for target enclave e_{id}, which is a must for preventing malicious PE's out-of-range INSPECT. In TAP$^\infty$, the thread context of e_{id} is also included in the INSPECT scope.

VI. Formal Guarantees

To recapitulate, one of our primary security goals is to prove our MLP that applied to an enclave platform still maintains the Secure-Remote-Execution (SRE) property. To achieve this, we reinterpret the SRE property on TAP$^\infty$ and describe the challenges of reproving the SRE property in Sec. VI-A. We contribute new theorem invariants in Sec. VI-B to address these challenges and complete the proof [1] .

A. Secure Remote Execution Guarantees

According to the Decomposition Lemma by Subramanyan *et al.* [52], it's sufficient to prove the triad of Secure Measurement, Integrity, and Confidentiality to hold the Secure Remote Execution (SRE) on TAP$^\infty$.

[1]Artifact of Formal Verification: https://github.com/arxgy/TAP-lambda

1) Secure Measurement: The enclave platforms must *measure* the target enclave to ensure its authenticity and state integrity before running it. We divide the measurement-related property into two different but related parts. The first part is stated as Eq. 2, meaning any two enclaves' initial states σ_1 and σ_2 are the same if and only if the enclaves' measurements are equal. The initial states are computed to the measurement μ when LAUNCH the enclave.

$$\begin{aligned}
&\forall \sigma_1, \sigma_2 \in \text{S}.\ \big(init(E_{e_1}(\sigma_1)) \wedge init(E_{e_2}(\sigma_2))\big) \implies \\
&\qquad \big(\mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2)\big)
\end{aligned} \tag{2}$$

The second part is that if two enclaves e_1 and e_2 have the same initial states from Eq. 2, they produce equivalent execution trace deterministically given equivalent input sequences. This property is also referred to as the *Execution Determinism* property, which is formalized as Eq. 3. It claims that given these assumptions, the target enclave will generate the same enclave private state E_e and the same output O_e after each atomic operation step in $\mathcal{G}$.

$$\begin{aligned}
&\forall \pi_1, \pi_2 \in TRACE(TS). \\
&\Big(E_{e_1}(\pi_1^{\langle 0\rangle}) = E_{e_2}(\pi_2^{\langle 0\rangle}) && \wedge \\
&\forall i \in \mathbb{N}.\ \pi_1^{\langle i\rangle}.curr = e_1 \iff \pi_2^{\langle i\rangle}.curr = e_2 && \wedge \\
&\forall i \in \mathbb{N}.\ \pi_1^{\langle i\rangle}.curr = e_1 \Rightarrow I_{e_1}(\pi_1^{\langle i\rangle}) = I_{e_2}(\pi_2^{\langle i\rangle})\Big) && \implies \\
&\Big(\forall i \in \mathbb{N}.\ E_{e_1}(\pi_1^{\langle i\rangle}) = E_{e_2}(\pi_2^{\langle i\rangle}) \wedge O_{e_1}(\pi_1^{\langle i\rangle}) = O_{e_2}(\pi_2^{\langle i\rangle})\Big)
\end{aligned} \tag{3}$$

For the secure boot stage of enclaves, the launch stage of root PEs is like legacy enclaves. During the launch stage of a CE e, its PE's measurement $\mu^{\mathcal{P}[e]}$ ought to be involved in μ^e such that the CE will only be launched and owned by the authenticated PE.

2) Integrity: As the second SRE property, *Integrity* ensures that the execution trace of the enclave in the presence of an adversary is entirely identical to the execution of the enclave program when the attacker is absent, which could be formalized by Eq. 4. In other words, the only approach to control the enclave's control flows and internal states is to control input sequences. A graphical demonstration of *Integrity* is presented in Fig. 4.

$$\begin{aligned}
&\forall \pi_1, \pi_2 \in TRACE(TS). \\
&\Big(E_{e}(\pi_1^{\langle 0\rangle}) = E_{e}(\pi_2^{\langle 0\rangle}) && \wedge \\
&\forall i \in \mathbb{N}.\ \pi_1^{\langle i\rangle}.curr = e \iff \pi_2^{\langle i\rangle}.curr = e && \wedge \\
&\forall i \in \mathbb{N}.\ \pi_1^{\langle i\rangle}.curr = e \Rightarrow I_{e}(\pi_1^{\langle i\rangle}) = I_{e}(\pi_2^{\langle i\rangle})\Big) && \implies \\
&\Big(\forall i \in \mathbb{N}.\ E_{e}(\pi_1^{\langle i\rangle}) = E_{e}(\pi_2^{\langle i\rangle}) \wedge O_{e}(\pi_1^{\langle i\rangle}) = O_{e}(\pi_2^{\langle i\rangle})\Big)
\end{aligned} \tag{4}$$

3) Confidentiality: As the last property, *Confidentiality* states that given the same enclave program holding different secrets, the adversary $\mathcal{A}$ cannot distinguish e_1 and e_2 and learn nothing but the information provided by the observation function *obs*. Thus for adversary $\mathcal{A}$ in step i, it receives same input $I_{e_1}(\sigma_i) = I_{e_2}(\sigma_i)$ and generates same states $A_{e_1}(\pi_{e_1}^{\langle i+1\rangle}) = A_{e_2}(\pi_{e_1}^{\langle i+1\rangle})$. Same with the assumption in

Integrity, the adversary can combine any possible machine instructions to perform arbitrary attacks. The formal format of *Confidentiality* is presented in Eq. 5 below.

$$\begin{aligned}
&\forall \pi_1, \pi_2 \in TRACE(TS). && (5)\\
&\Big(A_{e_1}(\pi_1^{\langle 0\rangle}) = A_{e_2}(\pi_2^{\langle 0\rangle}) && \wedge\\
&\forall i \in \mathbb{N}.\ \pi_1^{\langle i\rangle}.curr = \pi_2^{\langle i\rangle}.curr \wedge I^P(\pi_1^{\langle i\rangle}) = I^P(\pi_2^{\langle i\rangle}) && \wedge\\
&\forall i \in \mathbb{N}.\ \pi_1^{\langle i\rangle}.curr = e \ \Rightarrow\ obs(\pi_1^{\langle i\rangle}) = obs(\pi_2^{\langle i\rangle})\Big) && \Longrightarrow\\
&\Big(\forall i \in \mathbb{N}.\ A_{e_1}(\pi_1^{\langle i\rangle}) = A_{e_2}(\pi_2^{\langle i\rangle})\Big)
\end{aligned}$$

The challenges in TAP$^\infty$. The challenge of proving SRE in TAP$^\infty$ resides in the model complexity. As shown in Sec. IV-C1, the TAP$^\infty$ model complexity, $O(n^{k^{\lambda_{\max}}})$, has a double-exponential explosion over that of the original TAP model, $\Omega(n^k)$, in which n stands for the maximum process amount the privileged software and parent enclaves can have, and k for the solver-specific parameter.

This challenge arises due to the introduction of the parent-children relationships and multi-layered privileges, significantly increasing the number of valid operations and reachable transition states. An illustrative comparison of the model complexity between TAP$^\infty$ and previous works resides in Fig. 3c. Compared with the non-privileged TAP model in Fig. 3a, the execution flows of TS in TAP$^\infty$ not only include legacy PS-E but also PE-CEs. For example, in proving the Integrity of TAP$^\infty$ (Fig. 4), the *to-be-verified* PE, e_{id}, can create CEs and utilize `ENTER` to switch the execution contexts into theirs. Similarly, a malicious PE can enter its CEs, which are not trusted by our threat model and are also permitted to perform any malicious operations. Consequently, the maximum MLP layer $\lambda_{\max}$ significantly affects the model complexity. The mathematical deduction for the TAP$^\infty$ model complexity refers to Appendix B.

We optimize the SMT solver and formal propositions to solve the challenges above. Notably, we adopted *Relevancy Propagation* [18] in Z3 to accelerate the formal reasoning and utilized *unskolemized* [51], yet logically equivalent, propositions to partially prove the SRE property of TAP$^\infty$ with unlimited MLP layers ($\forall\lambda \in \mathbb{N}_+$). For more details, refer to Sec. VIII-A.

B. Theorem Invariants

This section describes the necessary core theorems proposed to proceed with the formal verification of TAP$^\infty$. These theorems are presented as invariants in the proof procedures of SRE. By these consistency properties below, we rigorously follow our PE and MLP prototypes designed in Sec. IV.

1) Parent-Children Relationship Consistency: We first begin with invariants related to newly introduced metadata fields. Firstly, the adversarial privileged attacker's parent is itself, since the OS or hypervisor is launched at boot time, as the Eq. 6 shows.

$$\forall \pi \in TRACE(TS).\ \forall i \in \mathbb{N}.\big(active(\mathbf{ps}) \wedge \mathcal{P}[\mathbf{ps}] = \mathbf{ps}\big) \quad (6)$$

Also, we put constraints on the identities of PEs, that is, any enclave's parent must be privileged or be the adversary (**ps**), which is illustrated in Eq. 7.

$$\begin{aligned}
&\forall \pi \in TRACE(TS).\ \forall i \in \mathbb{N}.\ \forall e \in \textrm{ID}. && (7)\\
&\qquad active(e) \Longrightarrow \mathcal{D}^{privil}_{\mathcal{P}[e]} \vee \big(\mathcal{P}[e] = \mathbf{ps}\big)
\end{aligned}$$

Given the system design choices outlined in Sec. IV, we defined the constraint on $\mathcal{P}$ concerning MLP. To recapitulate, the number of the multi-layered privileges is capped at $\lambda_{\max}$, as formalized in Eq. 8. To show this constraint, we must introduce a scaffold function, $ancestor(\mathcal{P} : \textrm{ID} \rightarrow \textrm{ID}, e : \textrm{ID}, k : \{0\} \cup \mathbb{N}_\lambda) : \textrm{ID}$, to help us calculate the k-distant ancestor of any enclave e under any parent-children relationship mapping $\mathcal{P}$. For example, the 1-distant ancestor of e is its parent, namely $e_1 = \mathcal{D}^{parent}_e$ and the 2-distant ancestor of e is its grandparent, namely $e_2 = \mathcal{D}^{parent}_{e_1}$. This constraint is

$$\begin{aligned}
&\forall \pi \in TRACE(TS).\ \forall i \in \mathbb{N}.\ \forall e \in \textrm{ID}.\ active(e) \Rightarrow && (8)\\
&\qquad (ancestor(\mathcal{P}, e, \lambda_{\max} + 1) = \mathbf{ps}) \wedge\\
&\qquad (D^{privil}_e \Rightarrow ancestor(\mathcal{P}, e, \lambda_{\max}) = \mathbf{ps})
\end{aligned}$$

Notably, this constraint can also be deduced from the semantics of `LAUNCH` in Eq. 1 and Eq. 7.

2) Exclusive Memory Consistency: To support PE's `INSPECT` operation, we provide a consistency property in TAP$^\infty$ stronger than the TAP$_C$ and TAP model proposed earlier, named after *Exclusive Memory Consistency*. This consistency property claims the equivalence of own and $\mathcal{D}^{EV}$ should be maintained in all saved metadata records and the currently running process. It means any physical address p that is exclusively owned by an enclave e_i should be mapped by virtual address regions of e_i only, and any exclusively protected virtual address v's mapped physical address p should be viewed as exclusive that belonging to e_i only, $\big(\forall v \in \textrm{VA}.\ \mathcal{D}^{EV}_e[v] \Leftrightarrow own[\mathcal{D}_e{}^{AM}[v]] = e\big)$.

However, in the earlier formal models, TAP and TAP$_C$ support only a one-way satisfaction relationship for each enclave e: $\mathcal{D}^{EV}_e[v] \Rightarrow own(\mathcal{D}^{AM}_e[v]) = e$. Moreover, this incomplete consistency between own and $\mathcal{D}^{EV}$ was maintained on target enclave e in the former models but for e only. In our TAP$^\infty$ model, we promoted this property to each valid PE and its valid CEs (if it has any), namely $\big(\forall e.\ \mathcal{D}^{privil}_e \vee \mathcal{D}^{privil}_{\mathcal{P}[e]}\big)$.

Finally, we give the comprehensive form of the *Exclusive Memory Consistency* as Eq. 9 below. Although PE must copy data to feature recipients, its CE, to ensure *Exclusive Memory Consistency*, the copy overhead of it is negligible.

$$\begin{aligned}
&\forall \pi \in TRACE(TS).\ \forall i \in \mathbb{N}.\ \forall e \in \textrm{ID}. && (9)\\
&\quad \Big(active(e) \wedge \big(\mathcal{D}^{privil}_e \vee \mathcal{D}^{privil}_{\mathcal{P}[e]}\big)\Big) \Longrightarrow\\
&\quad \Big(\forall v \in \textrm{VA}.\ \mathcal{D}^{EV}_e[v] \Longleftrightarrow own[\mathcal{D}^{AM}_e[v]] = e\Big)
\end{aligned}$$

Since other CEs might be compromised, the adversary also indirectly controls them. So the memory consistency also satisfies

$$\forall p \in \text{PA}.\ \forall v \in \text{VA}.\ \forall e \in \text{ID}.\ shared(p) \implies$$
$$\big(own[p] = \mathbf{ps}\ \wedge (\mathcal{D}_e^{AM}[v] = p) \Longleftrightarrow \neg\mathcal{D}_e^{EV}[v]\big)$$

In brief, the owner of a shared memory (*i.e.* $shared(p), p \in$ PA) should be the privileged adversary **ps**. If a virtual address $v \in$ VA of enclave e is mapped to p, then v should not be considered to be exclusively owned by e.

VII. Implementation in RISC-V Penglai

To demonstrate the feasibility of Palantír, we implemented it[2] on the Penglai open-source enclave platform [23], which is based on RISC-V processors. In Penglai, all enclave-related operations are managed by a high-privileged, lightweight firmware known as *secure monitor*, ensuring effective enclave management and robust security guarantees. The Palantír implementation complies with the security guarantees in Sec. VI in both interface-level (Sec. V-B) and theorem invariants (Sec. VI-B). For the interface-level alignments, we have updated the interfaces of parent enclaves in Penglai, revising operations such as `LAUNCH`, `ENTER`, `EXIT`, and `DESTROY`, by the specifications detailed in Table II and Sec. V-B. Additionally, we introduced extra operations, `PAUSE`, `RESUME`, and `INSPECT`, following our outlined specifications in the same section.

To uphold the theorem invariants in Sec. VI-B, we first ensure the *Parent-Children Relationship Consistency* through a management module within the secure monitor. This module governs the inter-enclave parent-children relationships according to the specifications outlined in Sec. VI-B1. Secondly, *Exclusive Memory Consistency* is naturally achieved in our implementation without any code modification due to Penglai's use of the RISC-V Trapped Virtual Memory (TVM) feature. TVM enables the secure monitor to unmap a contiguous physical memory region from the guest OS, allowing modifications to the OS's page table. This prevents the OS from accessing the memory regions secure monitor seeks to protect. Consequently, Penglai does not allocate a single exclusive physical memory region to two distinct active enclaves simultaneously. Further discussion on the security guarantees refers to Sec. X.

Implementation Breakdown in LoC. Although Palantír introduces privilege layers to deploy features, there is still a need to drop to privileged software to support essential primitives for PE and MLP. We implemented the foundational functions of Palantír in the secure monitor of Penglai platform. From the platform perspective, the TCB growth introduced by Palantír is 3,372 LoC only. In the users' view, the TCB growth size equals the 3,372 LoC above plus the runtime TCB and features PEs actually in use. Table VII gives a detailed implementation breakdown.

[2]Artifact of Implementation: https://github.com/arxgy/Penglai-Enclave-Privileged

TABLE VI: Formal verification complexity of TAP$^\infty$ model. Here #pn denotes the number of to-be-verified procedures, #fn denotes the number of functions, #an denotes the number of annotations, and #ln denotes the lines of codes (LoC).

Model/Proof	Model Details #pr	#fn	#an	#ln	Verification Time (s)	
TAP					$(\lambda_{\max} \equiv 0)$	
Integrity†	12	13	145	985	15	
Secure Measurement	6	3	100	800	6	
Confidentiality†	8	0	200	1388	17	
TAP$^\infty$					$\lambda_{\max} = \infty$	$\lambda_{\max} = 8$
Integrity†	15	14	146	1403	40610	83655
Secure Measurement	9	7	261	1271	○	445
Confidentiality†	8	0	378	2352	6842	57042

[1] ○ indicates the proof is inconclusive and dropped by the SMT solver. Appendix C provides a proof by induction as a supplement.
[2] † indicates the SMT solver adopts *Relevancy Propagation*.

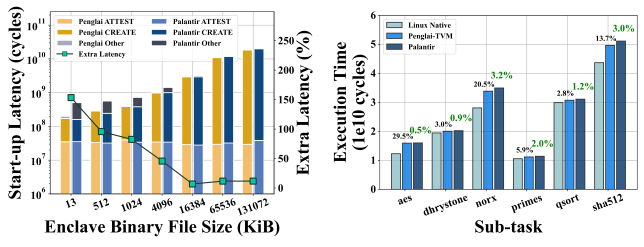
TABLE VII: Palantír implementation breakdown in LoC. The miscellaneous section encompasses comments and logs.

Function	Palantír Privileged TCB	Enclave Runtime TCB	Penglai Driver
Parent Enclave Primitives (Sec. V-B)	1,803	0	991
MLP Management (Sec. VI-B1)	256	0	0
Wrappers for Interfaces (Sec. V-B)	0	718	0
Miscellaneous	1,313	67	60
Total Line of Codes (LoC)	3,372	785	1,051

VIII. Evaluation

Benchmarks. We evaluate Palantír on four aspects:

- **Verification Results:** Verification costs of TAP$^\infty$ caused by the newly-introduced MLP primitives. (Sec. VIII-A)
- **Start-up Latency:** Introducing PE incurs negligible overhead to the start-up latency of CEs and legacy enclaves. Also, MLP can significantly reduce start-up latency in inter-enclave feature sharing. (Sec. VIII-B)
- **Computation Overhead:** Our MLP design incurs negligible computation overhead. (Sec. VIII-C)
- **Case Study:** We provide an end-to-end case study, *Hierarchical Deterministic Wallet* to show the benefits of Palantír in porting applications. (Sec. VIII-D)



(a) Start-up Latency: Micro-benchmark Breakdown

(b) Computation Overhead: RV8 Benchmark

Fig. 5: Evaluation on Palantír implementation. In Fig. 5a, the Other cost is caused by inter-enclave communications. In Fig. 5b, compared to the original Penglai platform, the average overhead of Palantír is 1.8%.

Experimental Setup. All evaluations are performed on a DELL PowerEdge T550 Tower Server with two Intel® Xeon® Gold 5318Y CPUs each with 48 cores at 2.10 GHz and 512 GB memory. For the verification of TAP$^{\infty}$, we use Boogie 2.16.0 and Z3 SMT Prover 4.8.7. We developed and evaluated our implementation on the PENGLAI Enclave Platform based on commit *3c0c81f*.

A. Verification Results

This section discusses our models and machine-checked proofs. We adopt BoogieIVL [14], [20] as our formal model-building language, which is designed to be a middle part of the program verifier and usually performs as an intermediate verification language between high-level languages like C or C#, and low-level SMT logic clauses. Boogie utilizes ACSL-style annotations [15] to specify verification conditions in the formal model, including pre-/post-conditions, loop invariants, *etc*. Then, the formal model and verification conditions will be compiled into SMTv2 commands and solved by the Z3 SMT solver [19]. For more details in the Boogie toolchain and TAP$^{\infty}$ formal system, refer to Appendix A.

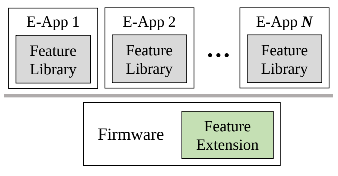
Table VI shows the verification result of TAP$^{\infty}$. **We successfully proved the SRE property of TAP$^{\infty}$ with unlimited privileged layers $\lambda_{\max} = \infty$.** It required 2,711 lines of code (LoC) in BoogieIVL to incrementally construct the TAP$^{\infty}$ model, support PE and MLP, and complete proofs for all three SRE properties: *Integrity*, *Secure Measurement*, and *Confidentiality*. This effort took three person-months working approximately 30 hours a week to finish. In detail, our formal verification work can be illustrated by the great verification time gap between our TAP$^{\infty}$ and the origin TAP in Table VI. Compared with TAP, our verification time bloated to 3$\sim$4 orders of magnitude across all properties. The significant discrepancy arises from increasing model complexity caused by the MLP, as previously discussed in Sec. VI.

As depicted in Table VI, we successfully verified the Confidentiality and Integrity of any enclave model with unlimited privilege layers ($\lambda_{\max} = \infty$). To solve the proof complexity challenge, We first employed an optimization methodology called ***Relevancy Propagation*** [18], which significantly expedited our proof by a factor of at least 200$\sim$300x. This performance improvement is because the Z3 SMT Solver navigates a vast set of constraints during verification, constantly maintaining and updating this set through new assignments. Each new assignment can trigger further constraint propagation. With *Relevancy Propagation* activated, the theorem prover identifies and tracks only the assignments critical to the proof, thereby significantly reducing unnecessary propagation.

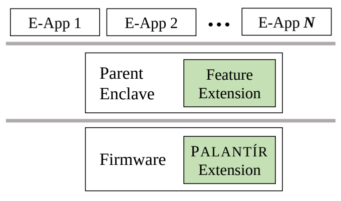
We also utilized de-skolemized [51], yet logically equivalent, substitutes of specification in Sec. V-B to solve the state explosion in TAP$^{\infty}$ caused by the MLP design. In mathematical logic, skolemized propositions mean propositions with uniformed nested quantifiers (*e.g.* $\forall x.\forall y.\phi(x, y)$). Symmetrically, de-skolemized ones are propositions with alternating nested quantifiers (*e.g.* $\forall x.\exists y.\phi(x, y)$). Notably, the SMT solver utilizes heuristics instead of constructive logic to check satisfiability, leading to an inconclusive proof of *Secure Measurement* when λ becomes large. Consequently, we give a manual proof of it in Appendix C. In brief, the proof is constructed and finished by inducting the execution state of transition systems.

TABLE VIII: Runtime latency of MLP. Each sub-task in the RV8 benchmark suite are run inside λ-layered PEs, and the overhead is compared with the $\lambda = 0$ case in percentage (%).

Sub-task \ Privilege Level (λ)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Avg.**
`AES`	0.407	0.892	0.842	1.472	1.308	1.338	1.042	0.090
`dhrystone`	0.561	0.184	-0.070	0.190	0.051	0.878	0.326	0.860
`norx`	1.085	1.240	0.863	0.831	0.425	1.622	1.064	1.544
`primes`	1.349	1.496	1.351	1.558	1.362	1.954	2.007	1.752
`qsort`	0.468	0.613	0.452	0.808	0.826	1.110	0.875	0.736
`sha512`	0.118	0.279	0.644	2.887	3.627	1.178	0.206	1.276
Avg.	0.406	0.892	0.842	1.472	1.308	1.338	1.042	**1.043**



(a) Monolithic Sharing Pattern



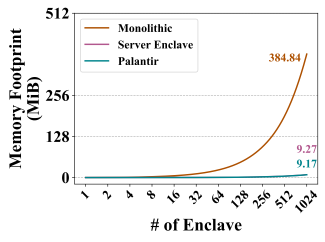
(b) PALANTÍR Sharing Pattern

Fig. 6: Real-world feature sharing. **Gray**-colored boxes indicate duplicated feature runtime during feature sharing. **Green**-colored boxes indicate newly introduced TCB for E-Apps.

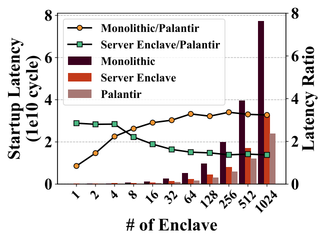
B. Start-up Latency

We measure the most crucial and costly part of the enclave usage scenario in the cloud: the creation and attestation stage.

Micro-benchmark. We first simulate a scenario in which a remote user $\mathcal{U}$ deploys its enclave e to a remote platform. Then we compare the time cost between deploying the non-privileged e in the original PENGLAI platform or by our PALANTÍR, in which we launch a PE first and let the PE create e then. We evaluate enclaves with different binary file sizes and analyze the inner overhead proportions inside the procedure.



(a) Memory Footprint



(b) Start-up Latency

Fig. 7: Evaluation of the real-world feature sharing. The PALANTÍR-based feature sharing pattern enables nearly constant-size memory footprint regarding CE number and 1.2$\sim$3.5x start-up acceleration for large-scale enclave scenarios, compared with the native feature extension pattern.

Given our prototype implementation, the cost of launching e on PALANTÍR will be more significant since we have a 2-stage start-up procedure.

Observation: According to the result in Fig. 5a, when the CE's binary size $<$ 1MiB, the overhead of launching CE by PALANTÍR is several times higher than on original platforms. However, when the binary size increases, the overhead gap tends to be negligible. This is mainly caused by the extra creating time of PE, as we must launch PE first to further launch e as a CE in PALANTÍR. Although the binary size of e varies, the agent PE program is fixed. So when e size surges, the cost proportion of creating the PE shrinks and becomes negligible ($< 10\%$). Besides, in PENGLAI platform, the critical overhead lies in LAUNCH since the secure monitor will compute the measurement of e in creation instead of the attestation stage, which also fits the proportion shown in Fig. 5a.

Real-World Feature Sharing. The multi-layered privileges design in PALANTÍR effectively reduces start-up latency for multiple enclave applications that utilize common services. As shown in Fig. 6a, the traditional monolithic feature extension patterns, such as *Architecture-Level Extension* and *Intra-Enclave Compartmentalization*, require library loading for each enclave application, thereby introducing unnecessary duplication of libraries. In contrast, PALANTÍR eliminates this redundancy by placing features in a PE. Then enclave applications, as its CEs, can share the features, as illustrated in Fig. 6b. Our evaluation confirms that such feature sharing can reduce enclave applications' memory footprint and start-up overhead.

Fig. 7 shows the overhead comparison of PALANTÍR with other feature extension patterns. We support the application enclaves with the Platform Security Architecture (PSA) storage feature by importing WolfSSL [2] and mbedtls [1] libraries into PALANTÍR and parent enclaves. The combined memory footprint of the WolfSSL and Mbed TLS libraries is approximately 1 MiB, whereas the application code accounts for about 100 KiB. In the baseline PENGLAI configuration, we demonstrate two setups. The first run LAUNCH enclaves containing the PSA feature and application codes are denoted as the *monolithic pattern*. The second configuration employs *server enclaves*, as proposed by PENGLAI, which houses the PSA feature. The legacy enclaves can send IPC requests to registered server enclaves to acquire services, denoted as *server enclave pattern*. On the other hand, the PALANTÍR setup always creates a PE to house the PSA service, and the PE governs multiple CEs for sharing.

Observation: Fig. 7a illustrates that, unlike the monolithic pattern's linearly increasing memory footprint, our PALANTÍR configuration achieves a nearly constant memory footprint overhead as the number of CEs increases. This efficiency stems from eliminating redundant feature codes, which produce a relatively small code size compared to the monolithic pattern. Fig. 7b demonstrates the runtime effectiveness of PALANTÍR by evaluating the start-up latency of application CEs. The speedup over the monolithic pattern (3∼4x) is caused by the large gap between enclave binary file sizes. The speedup over the server enclave pattern (1.5∼3x) can be attributed to the performance gap between the details in implementing inter-enclave communications.

TABLE IX: One-time setup overhead of Hierarchical Deterministic Wallet on PALANTÍR.

Hierarchy Component	Root Parent Enclave	Children Enclave
Overhead (cycles)	4.9×10^8	2.7×10^8

C. Computation Overhead

The Execution Control privilege granted to the parent enclave allows it to manage the scheduling of its CEs, thereby introducing inevitable context switch overheads to the original PENGLAI platform. To quantify the computational overhead imposed by the Execution Delegation relative to the native PENGLAI TEE platform, we utilized the CPU-intensive RV8 benchmark suite [12] to assess the overall execution overhead. We ported RV8 to PENGLAI and executed the RV8 applications as native Linux programs, as legacy PENGLAI enclaves, and as CEs controlled by a 1-layered PE, respectively. We then compared the total runtime costs across these different execution environments. As shown in Fig. 5b, the runtime overhead caused by our PALANTÍR implementation is $< 3.3\%$ in all cases and 1.8% on average, nearly negligible.

Multi-Layered Privileges Latency. PALANTÍR introduces multi-layered privileges (MLP) in Sec. IV-B and Sec. VII. Nested PEs will introduce extra control delegations since the modified secure monitor in PALANTÍR will delegate software interruptions or requests from a lower-layered PE to the higher-layered PE. To exclude irrelative interference from other enclaves, we allow each PE to possess only one CE to form an ownership chain, with the lowest-layered non-privileged CE c housing computations. We evaluated the overhead of MLP based on the RV8 above benchmark, as shown in Table VIII. Given results in Sec. VIII-A, we only consider machine-checkable layers ($\lambda_{\max} = 8$).

As depicted in Table VIII, MLP does not impose significant overheads to PENGLAI. With 16 repetitions of each sub-task, the MLP introduces always $\approx 1\%$ latencies. This negligible latency is primarily attributed to the execution flow predominantly switching between the lowest-layered CE c, which executes each sub-task, and its PE $\mathcal{P}[c]$, resulting in negligible differences compared to the optimal $\lambda_{\max} = 1$ case.

D. Case Study: Hierarchical Deterministic Wallet

To underscore the necessity of introducing multi-layered privilege separation in TEEs and to demonstrate the feasibility of our implementation in delivering features, we present an end-to-end example, *Hierarchical Deterministic Wallet*.

The Hierarchical Deterministic Wallet (HDW) is a cryptographic wallet that leverages a hierarchical structure to derive public-private key pairs from an initial master key

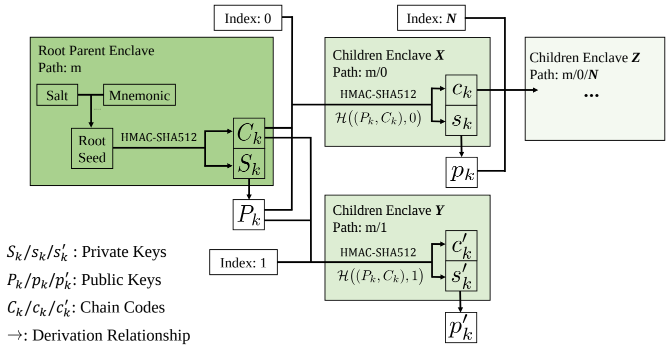


Fig. 8: Case study: Hierarchical Deterministic Wallet. Green-colored boxes indicate privilege layers. Deeper shades represent higher privilege levels.

TABLE X: Runtime performance of Hierarchical Deterministic Wallet case. The overhead is compared with the wallet implementation on a PENGLAI legacy enclave in percentage (%). The performance on the legacy enclave is taken on average for the path length.

Scheme \ #Derivation Path Length	**1**	**2**	**4**	**8**	**16**	**PALANTÍR Avg. (%)**	**Legacy Enclave Avg. (cycles)**
Master Key Generation	0.57	0.33	0.44	1.57	0.97	**0.78**	1.96×10^7
Child Key Derivation	7.03	6.34	9.22	8.43	9.68	**8.14**	2.01×10^7
Child Key Sign	0.84	1.28	3.21	1.70	1.76	**1.76**	2.14×10^7
Child Key Verify	1.14	2.11	3.55	4.98	3.60	**3.08**	1.25×10^8

pair for each cryptocurrency transaction. Initially proposed in BIP32 (Bitcoin Improvement Proposal) [59], HDW has since been adopted as a standard within the Bitcoin community. HDW is characterized by three primary features: (1) *deterministic generation*: all keys in a wallet are deterministically generated, ensuring a consistent and reproducible key generation process. (2) *master private key*: a wallet owner can derive public-private key pairs from a master key pair. The master key pair should be generated from the seed input. (3) *hierarchy*: the derived key pairs can function as master key pairs, enabling the recursive generation of further derived keys.

To support HDW, we integrated open-sourced libraries and cryptographic algorithms into PALANTÍR, including libbase58, secp256k1 elliptic curve, AES/Rijndael algorithm, *etc*. Additionally, we implemented the BIP39 [43] specification to facilitate the master seed generation process. The seed is generated using HMAC-SHA512 from a mnemonic and a passphrase salt. With this setup, HDW functionality can be seamlessly deployed onto PALANTÍR by utilizing existing cryptography library APIs and incorporating `PAUSE` and `RESUME` for inter-enclave communications.

Security Analysis. The multi-layered isolation of PALANTÍR meets the stringent security demand of HDW. As shown in Fig. 8, the hierarchical key pairs in HDW are distributed across the enclave layers in PALANTÍR, with each key pair housed within a distinct enclave. Indeed, the multi-layered privileges of enclave provide robust protection, safeguarding each master private key against privacy leakage and attacks from its potentially compromised child key pairs. In contrast, in a monolithic design that computes and places the master keys and child keys in the same process, the lack of hardware-aided hierarchical isolation leads to potential master private key leakage [22] once children's (derived) private keys are compromised. Moreover, the isomorphism between the parent-children key pair derivation hierarchy in HDW and the parent-children enclave hierarchy in PALANTÍR ensures both the correctness and determinism feature of generated keys.

Overhead. We utilized the existing testbench in the open-sourced trezor-crypto [4] library to evaluate the performance of the HDW on PALANTÍR. As mentioned, the profiling program can be seamlessly integrated into PALANTÍR without cryptographic API modifications. The overhead comprises a one-time setup phase and periodic key derivation/signing/verifying. The results in Table IX and X are averaged over 512 runs.

The setup overhead resides in launching multiple hierarchical enclaves, including one necessary root PE and multiple CEs as needed. Although the setup overhead scales linearly with the number of CEs, launching each enclave (Table IX) is incurred only once.

For runtime performance, we separately evaluated the crypto schemes of HDW, as depicted in Table X. The performance of PALANTÍR-based HDW was compared against the monolithic HDW design on a PENGLAI legacy enclave. To illustrate the scalability of PALANTÍR hierarchical enclaves, we accessed runtime performance across multiple levels of key derivation. In this case, the generated child key pairs (p_k, s_k) recursively generate child key pairs (p'_k, s'_k), forming a derivation path. The result indicates that PALANTÍR efficiently supports HDW without introducing significant overheads, regardless of the length of derivation paths. Previous evaluation in Table VIII also corroborated the scalability of hierarchical enclaves.

Furthermore, we observed that the Child Key Derivation scheme exposes a slightly higher overhead over others, primarily due to inter-enclave communications triggered by `PAUSE` and `RESUME`. Despite this, the overall impact on performance remains relatively low ($< 5\%$). This is because the derivation scheme ($\approx 10^7$ cycles) costs an order magnitude smaller than the sign and verify scheme ($\approx 10^8$ cycles). Moreover, each used key will only be derived exactly once but will be used for signing and verifying at least once, resulting in the impact of derivation on the overall performance of less than 10%.

Other Evaluations. The entire HDW library in PALANTÍR includes $27,184$ LoC in C imported from trezor-crypto [4]. It only takes 24 LoC modifications to be integrated into PALANTÍR SDK as a static library.

Given the programmability of hierarchical PE and MLP, the benefits of PALANTÍR extend beyond that. Its real-world advantages lie in the possibilities of what can be implemented within PEs, giving enclave authors more power.

IX. RELATED WORKS

Formal verification on TEE features. Among works on feature extension for TEE platforms, Lee *et al.* [34] introduces

Cerberus, an architectural-level memory-sharing extension incorporating formal verification to ensure its security.

The PALANTÍR and Cerberus have significant differences between them. Cerberus primarily focuses on providing a verified feature, the inter-enclave memory sharing. In comparison, PALANTÍR operates at a broader scale, offering a verifiable feature extension framework through multi-layered privilege separation. This allows users to develop customizable features tailored to their specific needs rather than limiting them to a predefined feature extension like Cerberus. For the design part, PALANTÍR introduces numerous new challenges and techniques. It significantly increases the verification complexity from TAP's and TAP$_C$'s $\Omega(n^k)$ to our TAP$^\infty$'s $\Omega(n^{k^{\lambda_{\max}}})$. Unlike Cerberus, which only offers static sharing interfaces to an enclave, PALANTÍR introduces not only *read-enabled Spatial Control* but also *Execution Control* over enclaves, as elaborated in Sec. V-B. From the functionality aspect, the formal model of Cerberus is considered a subset of our TAP$^\infty$. Concerning security, PALANTÍR is not securing the functionality of a specific enclave, but rather any functionality it provides, including other functionality to be explored.

Privilege separation within TEE. Prior work has been studied to support privilege protection levels within TEEs. Nested Enclave [44] provides privilege separation between the privileged *inner-enclave*s and non-privileged *outer-enclave*s to isolate the untrusted third-party libraries. vTZ [28] supports multiple secure VMs using the secure world primitive of ARM TrustZone. AEGIS [53] and CURE [13] offer distinct security types on the code and data and support multiple secure execution modes on an application.

Compared with existing works, our PALANTÍR advances beyond in three key aspects: (1) generality, (2) availability, and (3) security guarantees. First, the privilege separation model of PALANTÍR is designed for general enclave platforms, ensuring it is not constrained by any specific TEE design. At the same time, all of the existing work is tailored to a particular enclave platform. Second, PALANTÍR is the first to extend the inter-enclave privilege separation to a λ-layered model ($\lambda \geq 2$), marking a significant expansion from the typical single- or double-layered privilege models prevalent in prior works. Lastly, PALANTÍR distinguishes itself as the pioneer in formally verifying the multi-layered inter-enclave privilege separation model, TAP$^\infty$, on generic enclave platforms.

Privilege separation through micro-kernel architecture. Similar to PALANTÍR, the micro-kernel architecture offers performant separation between security-critical and less-privileged system components. Prior research [33] has formally verified a microkernel's functional correctness and security properties. However, the key distinction between PALANTÍR and microkernel architecture lies in their security models. In the threat model of PALANTÍR, the privileged software is untrusted; thereby, the security of PEs is still ensured by hardware isolation and formal verification, even under a compromised kernel. In other words, the nested enclaves in PALANTÍR fulfill the data confidentiality and integrity requirements of services better than micro-kernels in scenarios of outsourcing storage and computation to untrusted clouds.

X. DISCUSSION

Verification of implementation. As discussed in Sec. II and Sec. VII, our work proposes a multi-layered privileges model for general enclave platforms. Consequently, this paper does not cover code verification of our implementation.

Indeed, any discrepancies between the formal model specifications and the actual implementation can significantly increase real-world vulnerabilities. As declared in Sec. VII, our implementation adheres to the security specifications established in our TAP$^\infty$. Additionally, crucial invariant assumptions such as *Exclusive Memory Consistency* are supported by the inherent design of the PENGLAI platform. However, these implementation guarantees alone do not fully ensure the security principles are flawlessly executed. Binary-level verification is still required to assert that the actual implementation can be effectively refined to stringently meet the proven SRE security, which could be another line of research.

Capability of feature extension. To migrate existing applications to PALANTÍR, the developer needs to understand the semantics of PE primitives listed in Table II. The primitives are essential when implementing privilege-related features for inter-enclave management and communications. We have provided function interfaces as part of the SDK runtime to facilitate this process, as detailed in Table VII. Developers can utilize the interfaces to call the PE primitives as e-calls with structured parameters.

From another perspective, our *read-enabled* privilege design, as detailed in Section IV-C3, inherently limits the scope of future features that our PE framework can support. For instance, we are currently unable to support write-dependent features, such as *write-enabled* shared memory. However, this extension may conflict with the *Confidentiality* property. Consequently, further exploration of spatial isolation permissions is designated as future work.

PALANTÍR on other TEE platforms. As discussed in Sec. II, PALANTÍR focuses on designing a generic multi-layered enclave model independent of platform-specific features. The generality of PALANTÍR is based on the PE operations in Table II. For process-based TEEs, deployment can be achieved by introducing the PE primitives as firmware-level extensions. For instance, deploying PALANTÍR onto Keystone [35] follows a similar approach as with PENGLAI: (1) rewriting the secure monitor and kernel driver to support syscall-level primitives, and (2) enabling enclave runtime SDK to server as interfaces. For VM-based TEEs, deployment involves introducing primitives as ISA/hypervisor instructions, with additional modifications on MMU and para-virtualized I/O. This architecture-based extension approach has been previously adopted to support single-layered VM-based TEE partitioning [31], [60] and nested virtualizations [29].

XI. Conclusion

In conclusion, we proposed Palantír, a verified multi-layered privilege model on general enclave platforms. We introduced the parent-children inter-enclave relationship and multi-layered privileges (MLP) for secure feature extension for enclaves. We further provided detailed specifications for parent enclaves to evaluate the security properties within our TAP$^{\infty}$ model. We verified that the SRE security guarantee is rigorously upheld through automated formal verification. We also implemented the Palantír prototype on the Penglai open-source enclave platform. Lastly, we conducted a case study to show the security and efficiency benefits of our inter-enclave privilege layer design.

Acknowledgment

We thank our shepherd as well as the 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, 62102254, 62325207, 62132013, and 62172271.

References

[1] "Mbed TLS." https://www.trustedfirmware.org/projects/mbed-tls/, 2020.

[2] "Wolfssl embedded SSL/TLS library." https://www.wolfssl.com/, 2020.

[3] "An incorrect bounds calculation in the linux kernel ebpf verifier," *https://github.com/bsauce/kernel-exploit-factory/blob/main/CVE-2021-31440/exp/CVE-2021-31440.c*, 2021.

[4] "Trezor-crypto library." https://github.com/trezor/trezor-crypto/, 2021.

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

[6] "CVE-2023-5165." Available from MITRE, CVE-ID CVE-2023-5165 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-5165, 2023.

[7] "CVE-2023-6345." Available from MITRE, CVE-ID CVE-2023-6345 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-6345, 2023.

[8] *https://www.aliyun.com/*, Aliyun.

[9] *https://developer.arm.com/ip-products/security-ip/trustzone*, Arm TrustZone technology.

[10] *https://docs.kernel.org/bpf/verifier.html*, eBPF verifier. Linux.

[11] *https://cloud.google.com/*, Google Cloud.

[12] *https://github.com/michaeljclark/rv8-bench.*, RV8 Benchmark. 2017.

[13] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1073–1090. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani

[14] M. Barnett, B.-Y. E. Chang, R. DeLIne, B. Jacobs, and R. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO 2005*. Springer Berlin Heidelberg, November 2005.

[15] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C Specification Language," 2008.

[16] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Stealing Intel Secrets from SGX Enclaves via Speculative Execution," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, June 2019.

[17] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[18] L. de Moura and N. Bjørner, "Relevancy propagation," *Technical Report MSR-TR-2007-140, Microsoft Research, Tech. Rep.*, 2007.

[19] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[20] R. DeLIne and R. Leino, "Boogiepl: A typed procedural language for checking object-oriented programs," Tech. Rep. MSR-TR-2005-70, March 2005.

[21] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," in *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*. Springer, 2011, pp. 351–368.

[22] C.-I. Fan, Y.-F. Tseng, H.-P. Su, R.-H. Hsu, and H. Kikuchi, "Secure hierarchical bitcoin wallet scheme against privilege escalation attacks," *International Journal of Information Security*, vol. 19, pp. 245–255, 2020.

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

[24] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults," in *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*. Springer, 2017, pp. 357–380.

[25] A. U. S. Gopal, R. Soori, M. Ferdman, and D. Lee, "TAILCHECK: A lightweight heap overflow detection mechanism with page protection and tagged pointers," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 535–552.

[26] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure Live Migration of SGX Enclaves on Untrusted Cloud," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 225–236.

[27] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969. [Online]. Available: https://doi.org/10.1145/363235.363259

[28] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 541–556. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua

[29] Intel, "4th generation intel core vpro processors with intel vmcs shadowing," Tech. Rep., 2013, white Paper. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf

[30] Intel, "Intel SGX SDK," https://github.com/intel/linux-sgx, 2019.

[31] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, "H-SVM: Hardware-assisted secure virtual machines under a vulnerable hypervisor," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2833–2846, 2015.

[32] D. Kaplan, J. Powell, and T. Woller, "AMD SEV-SNP: Strengthening VM Isolationwith Integrity Protection and More," White paper, Tech. Rep., 2020.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220.

[34] D. Lee, K. Cheang, A. Thomas, C. Lu, P. Gaddamadugu, A. Vahldiek-Oberwagner, M. Vij, D. Song, S. A. Seshia, and K. Asanovic, "Cerberus: A formal approach to secure and efficient enclave memory sharing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1871–1885.

[35] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[36] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "CAMP: Compiler and allocator-based heap memory protection," in *33rd*

USENIX Security Symposium (USENIX Security 24). Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4015–4032. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng

[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013.

[38] M. S. Melara, M. J. Freedman, and M. Bowman, "Enclavedom: Privilege separation for large-TCB applications in trusted execution environments," *ArXiv*, vol. abs/1907.13245, 2019.

[39] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1466–1482.

[40] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–242.

[41] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party machine learning on trusted processors," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 619–636. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

[42] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.

[43] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "BIP 39: Mnemonic code for generating deterministic keys," https://github.com/bitcoin/bips/wiki/Comments:BIP-0039, September 2013, layer: Applications, Comments-Summary: Unanimously Discourage for implementation, Status: Proposed, Type: Standards Track.

[44] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with sgx," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 776–789.

[45] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, "Software foundations," *Webpage: http://www.cis.upenn.edu/bcpierce/sf/current/index.html*, 2010.

[46] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave," in *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*. Springer, 2018, pp. 451–470.

[47] M. Russinovich, "Introducing Azure confidential computing," https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017.

[48] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[49] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 753–768. [Online]. Available: https://doi.org/10.1145/3319535.3354252

[50] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB linux applications with SGX enclaves," in *The Network and Distributed System Security Symposium*, 2017.

[51] T. Skolem, "Logico-combinatorial investigations in the satisfiability or provability of mathematical propositions: a simplified proof of a theorem by l. löwenheim and generalizations of the theorem," *From Frege to Gödel. A Source Book in Mathematical Logic*, vol. 1931, pp. 252–263, 1879.

[52] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2435–2450.

[53] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *IEEE Des. Test*, vol. 24, no. 6, p. 570–580, nov 2007.

[54] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX Annual Technical Conference*, 2017, pp. 645–658.

[55] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[56] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 54–72.

[57] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[58] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for Intel SGX enclaves," in *Proceedings of the 19th International Middleware Conference*, ser. Middleware '18, 2018.

[59] P. Wuille, "BIP 32: Hierarchical Deterministic Wallets," https://github.com/bitcoin/bips/wiki/Comments:BIP-0032, February 2012, layer: Applications, Comments-Summary: No comments yet, Status: Final, Type: Informational, License: BSD-2-Clause.

[60] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 246–257.

[61] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4111–4128.

[62] Z. Yu, G. Yang, and X. Xing, "ShadowBound: Efficient heap memory protection through advanced metadata management and customized compiler optimization," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 7177–7193. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng

[63] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin, "Reusable enclaves for confidential serverless computing," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4015–4032. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-shixuan

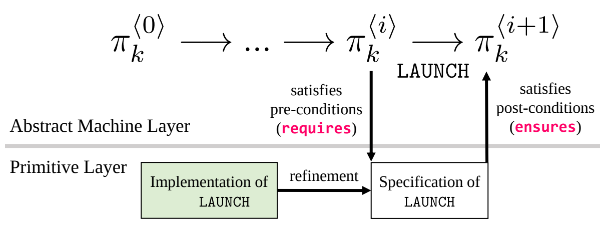
[64] L. Zhou, X. Ding, and F. Zhang, "Smile: Secure memory introspection for live enclave," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 386–401.

Appendix

A. Formal System Details

We formalized TAP$^\infty$ as a transition system TS written in Boogie [14]. To verify TAP$^\infty$, the verifier begins with a process state and, at each step, performs a state transition using an operation $op \in \mathcal{G}$ (Table II), as shown in Fig. 9.

Using ACSL-style [15] annotations, we specified how the state transits after each primitive is triggered. As shown in Fig. 10a, the `requires` statement will perform an identity sanity check over the current running process on the state machine, provided as pre-conditions. The `ensures` statement will confirm the model state details after the state transition, provided as post-conditions. Specifically, it will ensure that (1) if `LAUNCH` succeeds, the new system state is the same as the expected one, and (2) if `LAUNCH` fails, the next step state is the same as the previous one. The pre-/post-conditions



$\pi_k^{\langle 0 \rangle} \longrightarrow \dots \longrightarrow \pi_k^{\langle i \rangle} \xrightarrow{\texttt{LAUNCH}} \pi_k^{\langle i+1 \rangle}$
Abstract Machine Layer
satisfies pre-conditions (requires)
satisfies post-conditions (ensures)
Primitive Layer
Implementation of LAUNCH
refinement
Specification of LAUNCH

Fig. 9: State transition in TAP$^\infty$. Green-colored boxes indicate detailed implementation verified by Boogie and SMT solvers.

```
procedure launch(
    /* eid */ eid : tap_enclave_id_t,
    /* PT */ addr_map : addr_map_t,
    ...
    /* entrypoint. */ entrypoint: vaddr_t,
    /* privileged */ privileged : bool
)
    returns (status: enclave_op_result_t);
    requires (forall e : eid, n : int ::
     valid[e] &&
     is_valid_depth(n) &&
     is_valid_depth(n+1));
    ...
    ensures (status == op_success) ==>
        (valid[eid]);
    ensures (status == op_success) ==>
        (pc[eid] == entrypoint);
    ...
```

(a) LAUNCH Specification

```
implementation launch(
    /* eid */ eid : tap_enclave_id_t,
    /* PT */ addr_map : addr_map_t,
    ...
    /* entrypoint. */ entrypoint: vaddr_t,
    /* privileged */ privileged : bool
)
    returns (status: enclave_op_result_t);
{
     ...
     metadata_valid[eid]           := true;
     metadata_addr_map[eid]        := addr_map;
     ...
     metadata_pc[eid]              := entrypoint;
     metadata_privileged[eid]      := privileged;
     metadata_owner_map[eid]       := cpu_enclave_id;
     status := enclave_op_success;
}
```

(b) LAUNCH Implementation

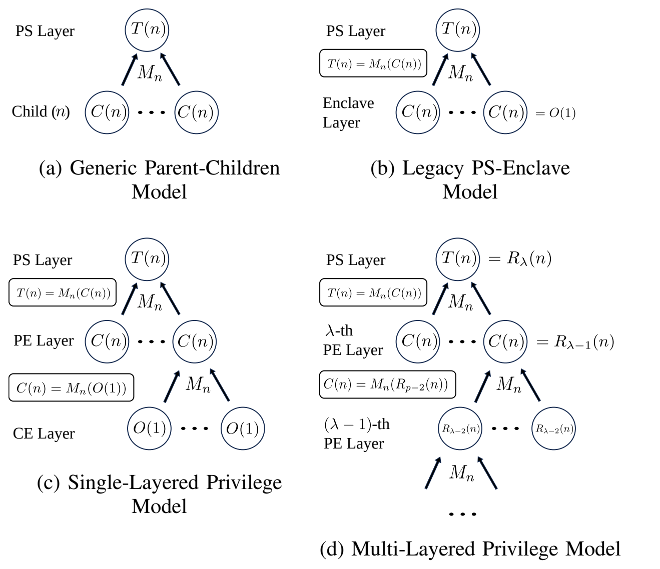
Fig. 10: TAP$^\infty$ formal model in Boogie. Please refer to the source code for the full version of the specification.

generate a complete verification condition set for Boogie to verify the TAP$^\infty$ model. Furthermore, although the higher-level SRE only invokes the specification-level primitives, the primitives themselves must be formally verified by providing a concrete implementation, as depicted in Fig. 10b.

B. TAP$^\infty$ Model Complexity Analysis

This section discusses the formal model complexity concerning $\lambda_{\max}$ in PALANTÍR. To recap, the $\lambda_{\max}$ represents the maximal number of privilege layers in the *multi-layered privileges* (MLP) design.

Overview: Proof Mechanization via $\lambda_{\max}$. To verify the model, a verifier must check each machine model's state and any processes that may be created. Consequently, the complexity of the proof directly depends on the number of possible processes involved, denoted as δ.



(a) Generic Parent-Children Model

(b) Legacy PS-Enclave Model

(c) Single-Layered Privilege Model

(d) Multi-Layered Privilege Model

Fig. 11: Model complexity analysis. The λ-*layered* model means the maximal number of privilege layers is λ. The *PS* term indicates privileged software. The black arrow $\rightarrow$ indicates the children-to-parent relationship.

Another observable fact is that the number of possible processes δ directly correlates with the number of maximal process layers $\lambda_{\max}$ and the number of maximal children for each privileged parent enclave n. Specifically, δ can be bounded by $\lambda_{\max}$ and n together as follows,

$$\delta \leq \sum_{\lambda=1}^{\lambda_{\max}} n^{\lambda-1} \tag{10}$$

This bound is because the parent-children process relationship can be presented as a tree, with the λ-th layer having at most $n^{\lambda-1}$ process(es). Notice that Eq. 10 establishes for any $\lambda_{\max}$. Consequently, an intuitive approach is to analyze the model's complexity through induction over $\lambda_{\max}$ with keeping n fixed.

By the observation above, we define constant time $\Theta(1)$ as the cost of verifying a normal process, such as a non-privileged CE or a non-enclave process. This is because these processes lack the permission to perform the CREATE and ENTER operation, thereby, cannot create new processes or privilege layers. In the following paragraphs, we give a generic parent-children model, analyze its complexity, and then fit our TAP$^\infty$ model designs into it.

Generic Parent-Children Model. First, we consider a parent-children model with a top parent node and n children nodes as shown in Fig. 11a, which can be applied to all of our formal models below. We denote $C(n)$ as the cost of verifying a children, and $T(n)$ as the cost of verifying the whole model. Since $T(n)$ also depends on the cost of children complexity $C(n)$, it can be re-written as a compositional function of $C(n)$:

$$T(n) = M_n(C(n)) = M_n \circ C(n) \tag{11}$$

in which M_n is a function only related to the proof algorithm. Since the formal verification is based on SAT solvers and the SAT problem is NP-complete, M_n is at least polynomial.

Legacy PS-enclave Model. As shown in Fig. 11b, the legacy PS-enclave model without PE fits the parent-children model above, in which we have $C(n) = \Theta(1)$ since each child is a legacy process. This model could be applied to any original TEE platform. By Eq. 11, its complexity $R(n)$ satisfies

$$R(n) = M_n \circ C(n) = M_n \circ \Theta(1) = M_n(\Theta(1)) \tag{12}$$

λ-Layered Models. After introducing MLP, we denote the complexity of verifying a λ-layered privilege model with at most n children as $R_\lambda(n)$.

Case 1. Single-Layered Privilege Model. For a single-layered model ($\lambda = 1$), the top layer fits the parent-children model as shown in Fig. 11c. Since each child could be considered a traditional PS-enclave model, its complexity satisfies $C(n) = R(n)$. By Eq. 11 and Eq. 12, the proof complexity $R_1(n)$ has

$$\begin{aligned} R_1(n) = T(n) &= M_n \circ R(n) \\ &= M_n \circ M_n \circ \Theta(1) \\ &= M_n^{(2)}(\Theta(1)) \end{aligned} \tag{13}$$

in which $M_n^{(\lambda)}$ means a compositional function mapping ($M_n \circ ... \circ M_n$) for λ times.

Case 2. Multi-Layered Privilege Model. For the multi-layered model with the maximal layer number λ ($\lambda \in \mathbb{N}_+$), we can also apply the parent-children model to the top layer, with each child as a $(\lambda - 1)$-layered model as shown in Fig. 11d. So by Eq. 11 and Eq. 12 we have such a recurrence equation:

$$\begin{aligned} &C(n) = R_{\lambda-1}(n) \\ &R_\lambda(n) = T(n) = M_n \circ C(n) = M_n \circ R_{\lambda-1}(n) \end{aligned}$$

by induction we know

$$\begin{aligned} R_\lambda(n) &= M_n^{(\lambda-1)} \circ R_1(n) \\ &= M_n^{(\lambda+1)} \circ \Theta(1) \\ &= \Theta(M_n^{(\lambda+1)}(1)) \end{aligned} \tag{14}$$

Since M_n is at least polynomial, it has $M_n(x) = \Omega(\sum_{i=0}^{q} m_i \cdot x^i)$ in which each parameter m_i should be a polynomial function $m_i(n)$ related to n.
To solve the $M_n^{(\lambda)}(1)$, let

$$k = \max\{deg(m_i(n)) | i \in [0, q]\}$$

then we have

$$M_n(1) = \Omega(\sum_{i=0}^{q} m_i(n)) = \Omega(n^k) \tag{15}$$

and

$$M_n^{(\lambda)}(1) = \Omega(\sum_{i=0}^{q} m_i^{(\lambda)}(n)) = \sum_{i=0}^{q} \Omega(n^{deg^\lambda(m_i(n))}) = \Omega(n^{k^\lambda}) \tag{16}$$

in which m_i^n also means a compositional function mapping $(m_i \circ ... \circ m_i)$ for n times. From the definition above, we know given the fixed verification algorithm in the SMT solver, the degree k of the polynomial function M_n will be fixed.

Conclusion 1. Complexity in General PE Model. By Eq. 14 and Eq. 16, the proof complexity of the λ-layered model is

$$R_\lambda(n) = \Omega(M_n^{(\lambda+1)}(1)) = \Omega(n^{k^{\lambda+1}}) \ (\forall p \in \mathbb{N})$$

which is a double-exponential function $R_\lambda(n) = E_n(\lambda)$ about λ given a fixed polynomial formal verification algorithm.

Conclusion 2. TAP Complexity. By Eq. 12 and Eq. 15, the original TAP model fits the PS-enclave model.

$$R^{\text{TAP}}(n) = M_n(O(1)) = \Omega(n^k)$$

Conclusion 3. TAP$^\infty$ Complexity. In TAP$^\infty$, we confine the upper bound of the number of privilege layers with range $\lambda < \lambda_{\max}$. By Eq. 13 and Eq. 16, our TAP$^\infty$ model fits the $(\lambda_{\max} - 1)$-layered model ($\lambda = \lambda_{\max} - 1$).

$$R^{\text{TAP}^\infty}(n) = \Omega(M_n^{((\lambda_{\max}-1)+1)}(1)) = \Omega(n^{k^{\lambda_{\max}}})$$

We can see TAP$^\infty$ has an double-exponential-$\lambda_{\max}$ complexity explosion from original TAP complexity $\Omega(n^k)$. That is also why we named our model TAP$^\infty$.

C. Proof Sketch for Unlimited TAP$^\infty$

This section provides a proof sketch by hand that inductively proves the Secure Remote Execution (SRE) property in TAP$^\infty$ with unlimited privilege levels λ, namely $\lambda_{\max} = \infty$. Since the *Integrity* and *Confidentiality* of TAP$^\infty$ has been formally verified by us in Table VI, we only focus on the *Secure Measurement* property. Take a step further, since the first part of *Secure Measurement* (Eq. 2) is independent with privilege level λ, we only pay attention to its second part, the *Execution Determinism* property (Eq. 3).

Inductive Proof of Execution Determinism.

To recap, the *Execution Determinism* states that if two enclaves e_1 and e_2 have the same initial states from Eq. 2, they produce equivalent execution trace deterministically given equivalent input sequences.

The initial $\lambda = 0$ case has been formally proven in TAP$^\infty$ (Table VI). By induction, suppose the *Execution Determinism* is established for any $\lambda < n$ with some $n \in \mathbb{N}$. Then for an n-length parent-children ownership chain $\{e_0 \to e_1 \to ... \to e_n\}$ with e_0 as the root parent enclave, our goal is to prove that the execution of e_n is also deterministic. By induction, the execution of $e_0, e_1, ..., e_{n-1}$ are deterministic. Then the states of e_{n-1} after each small step are deterministic. Consequently, the startup timing and initial states of e_n are deterministic.

Since each PE can not change the inner state of its CE, the execution of e_n could be considered as a non-privileged enclave just same as $\lambda = 0$ case of TAP$^\infty$. The only difference is that the enclave ids $e_0, e_1, ..., e_{n-1}$ are no longer available. From previous verification, we have known that the *Execution Determinism* works for the $\lambda = 0$ case. Given the definite initial state of the to-be-created enclave e_n and a definite input stream, the execution of e_n is also deterministic. □