Scalable and Secure Virtualization of HSM with ScaleTrust

Juhyeng Han, Insu Yun, Seongmin Kim, Taesoo Kim, *Member, IEEE*,
Sooel Son, *Member, IEEE*, and Dongsu Han, *Member, IEEE*

***Abstract*—Hardware security modules (HSMs) have been utilized as a trustworthy foundation for cloud services. Unfortunately, existing systems using HSMs fail to meet multi-tenant scalability arising from the emerging trends such as microservices, which utilize frequent cryptographic operations. As an alternative, cloud vendors provide HSMs as a service. However, such cloud-managed HSM usage models raise security concerns due to their untrusted and shared operating environment. We propose ScaleTrust, a scalable and secure system for key management. ScaleTrust allows us to scale the number of virtual HSM partitions, each of which is isolated with respect to each other and is robust against cloud insider attacks, while preserving physical isolation of the root of trust. To enable this, ScaleTrust uses Intel SGX and multiple HSM features, such as restricting key usage by controlling key attributes of in-HSM keys and establishing a secure channel using only HSM commands. Finally, we apply ScaleTrust to four real-world systems: Keyless SSL for TLS private key offloading, JSON Web Token authentication for microservices, key provisioning, and encryption in database systems. Our evaluation shows that ScaleTrust achieves multi-tenancy in a scalable way by providing multiple virtual HSMs with legacy HSM devices that are designed to support a single tenant. ScaleTrust provides security against insider threats while incurring 11.9% and 39.0% of end-to-end throughput and latency overhead for Keyless SSL compared to stand-alone HSMs.**

***Index Terms*—hardware security module (HSM), trusted execution environment (TEE), key management service (KMS), scalability, cloud computing security**

I. Introduction

HARDWARE security modules (HSMs) have served as a foundation of trust in the remote computation for cloud and edge services. Their applications span diverse domains, including certificate authorities (CAs) in public key infrastructure [7], e-commerce payment [28], and DNSSEC [21]. An HSM enables secure cryptographic operations (e.g., signing certificates) while providing physical isolation of cryptographic keys—plain-text keys never leave the HSM.

Unfortunately, scaling out on-premises HSMs involves significant capital investment and increases management complexity [15], [19]. The demand for service-level isolation has led to HSM partitioning, which virtualizes a physical HSM with isolated key protection [79]. However, due to limited hardware resources, state-of-the-art HSMs can only provide a limited number ($\sim$100) of HSM partitions [84]. Fundamentally, hardware-based isolation of in-HSM keys cannot meet the growing demands of emerging IT technologies, such as microservice architecture [20] and Keyless SSL [63], which require frequent inter-service cryptographic transactions [98].

To resolve the scalability issue, two approaches have been proposed. The first option is to entirely shift cryptographic operations to software-based key management service (KMS) modules and to leverage a commodity trusted execution environment (TEE) [19], [35]. Compared to HSMs, this achieves cost-effective horizontal scalability, as it utilizes commodity servers. However, this approach tampers with HSM-grade security properties such as physical separation and tamper-resistance that existing regulations [49] require, thus impeding its adoption in practice. For example, Canadian and U.S. governments require compliance with federal information processing standards (FIPS) 140-2 [1], which mandate physical separation for certification level 3 and above.

Another option is to utilize on-cloud key management services (KMS) [8], [58] that provide cloud-backed FIPS-validated HSMs in a virtualized form. However, this introduces fundamental security problems due to the untrusted nature of cloud platforms [13], [30]. Cloud KMSs are managed by untrusted cloud providers [78] and do not offer any protection against insider threats. For example, malicious software running in a cloud instance can obtain credentials to access HSMs and issue unauthorized commands to extract in-HSM keys Furthermore, a naïve approach to enable multi-tenancy by sharing an HSM (or an HSM partition) across multiple tenants causes security issues because it does not provide isolation across virtual partitions. For example, a malicious insider sharing an HSM with other tenants can execute malicious commands on the HSM to abuse the tenants' keys.

This paper presents ScaleTrust, which provides scalable and secure virtualization of legacy HSMs in a cloud environment. Specifically, ScaleTrust is designed to scale the number of virtual HSM partitions to accommodate multiple tenants while providing security against cloud insiders who control the HSM. ScaleTrust is also vendor-neutral—any legacy HSM can be used with ScaleTrust to scale the number of virtual HSM partitions for multi-tenancy.

To secure the root of trust, ScaleTrust preserves the physical separation property. It creates a logical partition (vHSM) inside the HSM and bootstraps a secure communication channel

J. Han, I. Yun, and D. Han are with the School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea. E-mail: sparkly9399@gmail.com; insuyun@kaist.ac.kr; dongsu.han@gmail.com.

S. Kim is with the Department of Convergence Security Engineering, Sungshin Women's University, Seoul 02844, South Korea. E-mail: dallas1004@gmail.com.

T. Kim is with the School of Cybersecurity and Privacy and the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA. E-mail: taesoo@gatech.edu.

S. Son is with the School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea. E-mail: sl.son@kaist.ac.kr.

between vHSM and client, which is secure against the malicious insider. This is enabled by utilizing a combination of PCKS#11 commands. For this, a trusted intermediary SGX enclave translates the user command to a series of HSM commands. However, because a user establishes a direct secure channel with vHSM, the intermediate enclave does not observe any plain-text keys, which makes ScaleTrust resilient to SGX side-channel attacks that try to leak HSM-generated keys. Finally, to ensure vHSM isolation, ScaleTrust detects any unauthorized key usage via log verification. ScaleTrust can unequivocally detect any key misuse attempts from a malicious insider.

We present a rigorous security analysis of ScaleTrust in § VIII. Our application case studies demonstrate that ScaleTrust is capable of efficiently scaling out virtual HSM partitions when applying ScaleTrust to four real-world systems: JSON Web Token authentication for microservices, Keyless SSL for TLS private key offloading, data encryption and key encryption in database systems. Our evaluation results show that ScaleTrust incurs performance and HSM key storage overhead for securing key management operations; ScaleTrust has 11.9% and 10.2% of end-to-end throughput overhead for Keyless SSL and JSON Web Token authentication service, respectively, and 8.14% of HSM key storage overhead compared to a stand-alone HSM, which does not provide protection against malicious insiders.

Our key contributions are as follows:

- A new HSM usage model that achieves both security and cost-effective multi-tenant scalability in key management.
- Design and implementation of secure key management operations for an HSM card with limited PKCS #11 APIs, which enables secure channel establishment and isolation across virtualized HSMs in a shared operating environment.
- Security analysis that demonstrates how ScaleTrust protects against attacks on each system component.
- Application case studies of ScaleTrust with performance evaluation that show practical benefits of our prototype.

II. Background on HSM and KMS

A Hardware Security Module (HSM) is a crypto-processing device that securely manages digital keys and performs cryptographic operations. To access an HSM device, a client typically uses public key cryptography standard (PKCS) #11 [52], which defines cryptographic operation APIs, such as key generation, encryption/decryption, and signing/verification. Using the PKCS #11 APIs, a client first establishes an HSM session, through which it issues commands. Note that HSMs support concurrent command execution across multiple sessions for high performance. The following summarizes four key security properties provided by HSMs widely used in cloud environments [7], [42], [57], [83].

FIPS-validated components. FIPS 140-2 regulations [1] strictly mandate that an HSM should provide hardware isolation, physical tamper-resistant protection, and self-destruction on hardware tamper events. The FIPS regulations also require HSMs to use FIPS-approved cryptographic algorithms with validated software implementations [86].

Key attributes. A client can set the access and usage policy of a key in an HSM by specifying key attributes when the key is generated or imported (unwrapped). For example, if the `extractable` attribute of a key is set to true, the key is extractable from the HSM, and if its `unwrap` attribute is set to false, the key cannot be used to perform unwrapping operations. In addition, an unwrapping key can have an `unwrap_template`, which consists of a set of key attributes and restricts all keys that are unwrapped by the unwrapping key to include the same key attribute values.

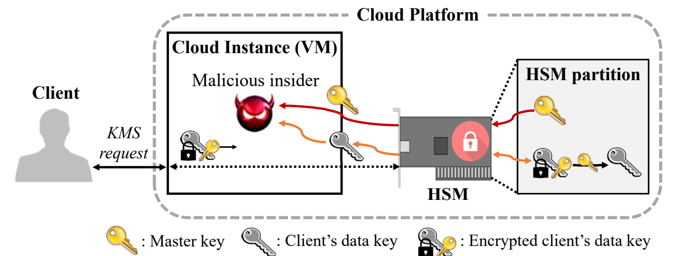


Figure 1: Security limitations of HSM-backed Cloud KMS.

Key authentication. HSMs support public key confirmation (PKC) [81] that provides an HSM-signed certificate for a non-extractable RSA key pair. In this certificate, the key pair is signed by an HSM-specific hardware key and an HSM manufacturer's key. Therefore, anyone with the HSM vendor's root certificate is able to confirm whether this RSA key pair originates from the specific HSM with the hardware key.

Log integrity. To support auditing of HSM events, an HSM generates log messages, each of which contains a communication session ID, an HSM command type, a stored key handle, and a chain of HMAC digests that ensures the message integrity. The HMAC key for the digests is kept in the HSM's protected memory, and the HSM encrypts this key when exporting it. Thus, HSM users cannot directly obtain the key in plain-text; however, only legitimate HSMs manufactured by *the same vendor* are able to import the HMAC key after decrypting the encrypted key. Given an HSM, HSM users are able to confirm whether generated log messages originate from the HSM and have not been changed by adversaries. HSM users verify the integrity of generated HSM logs by sending a log verification request to this HSM.

Key management services (KMS). A KMS typically builds the two-level key hierarchy [8]. A master key is used to encrypt/decrypt a client's data key as a key wrapping key or perform security-sensitive key operations, such as TLS private key operations. ScaleTrust follows the above usage model. A KMS typically uses the master key only within the HSM that is protected by a personal identification number (PIN) or password-based authentication [6], [41], [61]. Clients' data keys are used in the client system to perform client-side cryptographic operations (e.g., database table encryption). To store the data keys encrypted at rest, the client requests the KMS to encrypt the keys using the master key.

III. Motivation

Insider threat in cloud KMS. HSM-backed cloud KMS virtualizes HSMs [5], [8], [33], [58] by providing virtual partitions. In particular, it adds a level of indirection to isolate each tenant even within the same HSM partition, enabling

this approach to be more cost-effective than using dedicated HSMs [90].

However, this model suffers from a fundamental security limitation; it provides no protection against a malicious insider (i.e., a cloud provider) who has control over the HSM. This is a fundamental problem because the HSM credential is kept by the cloud provider and all HSM access, including access to the log, goes through a cloud intermediary. For example, in Cloud KMS, the KMS relays a client's request to the HSM instead of directly establishing a secure channel between the client and the HSM. This Man-in-the-Middle (MitM) environment allows the KMS to fully control communications between the client and the HSM, as shown in Figure 1. The malicious KMS can freely eavesdrop on their traffic and even send arbitrary commands to the HSM, thus leaking or misusing a client's secret key. Worse yet, this can go undetected because the insider can modify the KMS log, which is signed by the cloud KMS provider [11], [46].

Limitations of existing approach. To address this issue, a recent approach employs a hybrid solution [31] that utilizes SGX enclaves with an HSM to mitigate this insider threat. In particular, this solution replaces the KMS in the cloud environment with SGX, establishing a trusted computing environment that offers confidentiality and integrity. This hybrid approach indeed addresses security threats that this malicious insider poses but does not protect the client's secrets from SGX side-channel attacks. Recent studies have shown that SGX suffers from various types of side-channel attacks [34], [51], [73], [92], [93], [95], [97], which significantly undermine the secrecy of client's keys that the HSM generates. Unfortunately, existing solutions rely on the confidentiality guarantee from SGX, and their enclaves manage cryptographic keys in plain-text forms. Therefore, with SGX side channel attacks, these solutions can be easily broken, leaking the secrets. Even worse, existing techniques to prevent SGX side channels are either impractical [3], [4], [65], [71] (e.g., 50x slowdown [3]) or only covers specific types of attacks [36], [76].

IV. Design Goals

Motivated by the limitations of existing KMS, we propose ScaleTrust, designed to achieve four goals:

- **G1: Protection against insider threats.** ScaleTrust must ensure protection against malicious insiders in a cloud environment.
- **G2: Key usage isolation for multi-tenancy.** ScaleTrust must support multi-tenancy using legacy HSMs. In providing multi-tenancy, it must ensure isolation—a property in which only the tenant who created a master key in the HSM can access the key.
- **G3: FIPS-grade protection for the root of trust.** ScaleTrust utilizes FIPS-validated HSMs to ensure physical separation of master keys.
- **G4: SGX side-channel attack mitigation.** ScaleTrust must protect itself from attackers who try to achieve HSM-generated keys using SGX side-channel attacks.

Threat model and assumptions. We assume that a powerful adversary attempts to directly obtain or abuse cryptographic keys stored in an HSM or CPU-hardened trusted execution environments (TEEs). The adversary shares the same host or cloud platform with victims. He/she also has full control over the system software, such as the operating system and the hypervisor [12], [13], [69]. The adversary is thus able to forge the communication between the users and HSMs by manipulating the system memory. In addition, we assume that an attacker can even leak HSM-generated keys in enclaves using side-channel attacks [51], [97]. However, this adversary is not powerful enough to break an enclave's integrity nor capable of compromising any confidentiality requirements that are necessary to guarantee the integrity of the SGX ecosystem (e.g., an attestation key in the quoting enclave remains secure) [94], [96]. It is worth noting that such attacks are more difficult than attacking application enclaves because attacks against platform enclaves (e.g., quoting enclave) rely on specific types of micro-architectural vulnerabilities [92], [96]. Also, the security of quoting enclaves is essential for the SGX ecosystem; it is of utmost concern for SGX maintainers, including Intel and Microsoft. For quoting enclaves, they apply up-to-date compilation techniques, use small TCBs (the size of the Intel-signed quoting enclave prebuilt is 885KB), and promptly remediate potential threats. Unlike the quoting enclave, application enclaves tend to have a much larger attack surface and are more likely to have gadgets that can be exploited (e.g., page-table-based [97] and branch shadowing side-channel attacks [51]).

Finally, we assume HSMs are trustworthy. For instance, HSMs always perform reliable cryptographic operations as well as logging. Likewise, we assume that a PKC certificate from an HSM is trustworthy and that a client can verify its authenticity using the HSM vendor's public key. We also trust the SGX enclave's code integrity and a verification of (SGX-enabled) the platform's genuineness, supported by SGX remote attestation. Defense against denial of service [56] attacks and code vulnerabilities in HSM [14] or enclave [50] are outside the scope of this paper.

V. ScaleTrust Approach and Challenges

We envision a secure and cost-effective HSM virtualization built on top of minimal trusted components—an HSM and an SGX enclave—without trusting the cloud platform provider. ScaleTrust supports FIPS-grade protection, with HSMs serving as the root of trust, while enabling key isolation among different clients by co-utilizing SGX enclaves. However, this involves solving non-trivial challenges. We first describe a strawman design that highlights the challenge.

Strawman approach: Naïve TEE adoption. A strawman approach is to use an SGX enclave that serially relays each HSM command from clients to the HSM and verifies whether the HSM executed exactly the same command in the same order by inspecting the HSM log. However, this approach does not prevent attackers from issuing malicious HSM commands, such as exporting a secret from HSMs. Furthermore, attackers can observe the communication channel and/or even truncate HSM logs to hide their key usage logs.

Challenges in secure communication. To solve this problem, we first need to establish a secure channel between an HSM

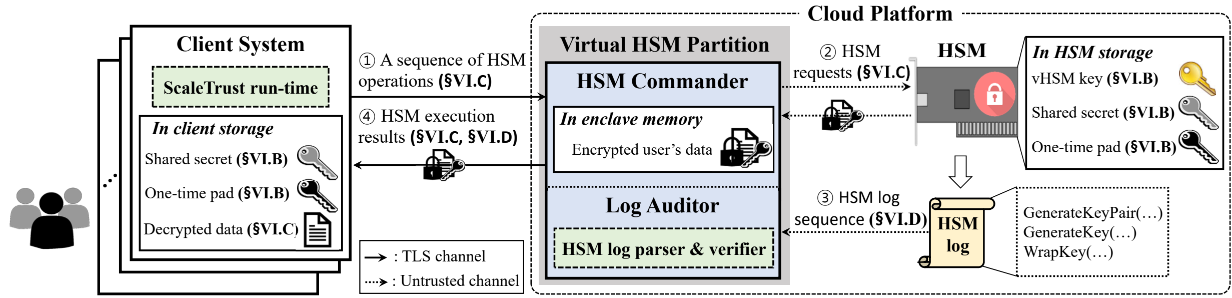


Figure 2: Deployment model of ScaleTrust (①-④: Workflow of ScaleTrust presented in § VI-A).

and enclaves without trusting the underlying cloud platform. However, this is not trivial in this context because a PCIe HSM supports limited cryptographic interfaces such as PKCS #11 APIs. In particular, these APIs do not support in-place signing operations (i.e., signing an in-HSM key with another in-HSM key), which make building a TLS-like authenticated channel challenging. For example, an HSM alone is not able to conduct a TLS handshake involving a Diffie-Hellman key exchange because the HSM is unable to sign the DH parameter by itself without other components that initiate the signing procedure. This dependency enables MitM attackers to easily intercept the encrypted communications or to impersonate each endpoint.

Protecting keys against malicious insiders. Because an HSM is a cryptographic black box, inspecting HSM logs is the only way to detect malicious HSM commands that the HSM performed. However, identifying malicious key usage from the log is not trivial for two reasons: 1) Different cloud tenants share the same cloud-managed HSMs; thus, HSM logs are multiplexed by numerous cloud clients. 2) The malicious insider can launch MitM attacks to truncate log messages because the logging system is only designed to ensure the integrity of consecutive log messages with a sequence of chained HMACs [80] and truncated log messages still hold valid cryptographic proofs. When a cloud KMS relays client requests to the HSM, a malicious insider can insert a *WrapKey* command, which extracts in-HSM keys. Then, the attacker can truncate the log sequence to hide the behavior, making it difficult to detect the ongoing attack.

VI. System Design

ScaleTrust achieves the aforementioned goals by co-utilizing HSMs and SGX enclaves, as shown in Figure 2. ScaleTrust utilizes HSMs to support physical separation for storing multiple clients' master keys (i.e., multi-tenancy). The enclaves in ScaleTrust work in tandem to 1) isolate the use of keys in the HSM for each client and 2) support secure communication between an HSM and KMS clients.

A. Workflow: Building a Chain-of-Trust

System overview. Figure 2 presents an overview of our system. The cloud side of ScaleTrust consists of an HSM and two types of enclave threads that share enclave memory:

- **HSM Commander Thread (Commander)** acts as a secure bridge between an HSM and a KMS client, isolating in-HSM key usages for each client.
- **Log Auditor Thread (Auditor)** detects any unauthorized access to in-HSM keys by verifying the HSM log.

Key type	Purpose	Key abuse protection
vHSM key	Identify vHSM, Authenticated root of trust, Unwrap secrets	Non-extractable, Unwrap-only, Unwrap template
Shared secret	Client ← HSM secure channel	Non-extractable, Wrap-only
One-time pad	Client → HSM secure channel	Erase after use

Table I: ScaleTrust keys used for secure communication.

We now describe step by step how ScaleTrust uses keys in the HSM.

① A ScaleTrust run-time at the client establishes a communication channel with the commander and authenticates the integrity of the commander enclave through remote attestation. If it succeeds, the run-time establishes a TLS channel with the commander. After that, when a client issues an HSM operation request, the run-time translates the request into a sequence of HSM commands (§ VI-C) and sends them to the commander through the TLS channel.

② Upon receiving the sequence of HSM operations, the commander issues the commands to the HSM through the PCKS#11 API. As part of the process, a virtual HSM partition is created and a secure channel between the client and the partition is established (§ VI-B).

③-④ The commander, with the help of a log auditor, verifies whether the client's request was executed in isolation. If the verification fails, the commander revokes the HSM execution results and informs the client (§ VI-D). Otherwise, the client receives the execution result from the virtual HSM partition through the secure channel.

B. Bootstrapping Secure Channel

ScaleTrust uses the PKCS #11 interface to bootstrap a virtual HSM partition (vHSM) and a secure communication channel between the client and vHSM, which preserves the confidentiality, integrity, and authenticity of communication. The use of a secure channel ensures that only the vHSM dedicated to a client is able to decrypt the HSM responses, while the commander enclave which relays the communication cannot access the responses in plain-text. Table I summarizes ScaleTrust's keys used for secure communication.

A virtual HSM partition provides an illusion that a client has exclusive access to the in-HSM keys when in reality, multiple

tenants share a single HSM. A vHSM is identified by the vHSM key (public) that a client holds.

vHSM creation. A vHSM is created by establishing a vHSM key that is a non-extractable RSA key pair stored in the HSM. The private key never leaves the HSM (non-extractable) because the key serves as the root of trust for other keys subsequently derived from the vHSM and a public key confirmation (PKC) certificate [81] signed by it that proves keys were generated by the vHSM.

Since the vHSM key is used to identify a vHSM, it must be handed over to the client. However, since the response of the HSM can be observed or manipulated by MitM attackers, ScaleTrust must authenticate the vHSM public key, without exporting the private key. For this, we issue a PKC certificate [81] that builds a chain of trust starting from the HSM vendor's root certificate, which is publicly available.

Establishing shared secret. After receiving the vHSM public key from the commander enclave, ScaleTrust establishes a shared secret with the vHSM for secure communication. To share a secret, the client sends an encrypted secret key using the public key. The vHSM then decrypts and stores the secret key in the HSM using the private key (`UnWrapKey`). This ensures that the plain-text secret key is accessible only to the client and the vHSM but is not visible to the commander.

In a normal environment, establishing a shared secret would be sufficient for bootstrapping a secure communication channel. However, this is not the case for ScaleTrust due to our unique threat model, which assumes the insider (and side-channel) threat. When the HSM sends an encrypted message using the secret key, an insider can observe the message and issue an HSM command to decrypt the message to obtain plain-text. To prevent this, we set the attribute of the secret key such that it can be used only for encryption but not for decryption. To automatically enforce such an attribute upon importing a secret key, ScaleTrust sets the unwrap template attribute of the vHSM key such that all keys unwrapped by vHSM keys can be used only for encryption. ScaleTrust verifies this property by issuing an unwrap request to HSM using the secret key upon creation of a vHSM. Verification succeeds when ScaleTrust detects a decrypt failure log message. Details of log verification are presented in § VI-D.

One-time pad. Since the shared secret key cannot be used to decrypt a message within the HSM, ScaleTrust utilizes a one-time pad (OTP), which is an AES key for building a secure channel between a client and the HSM. When a client requests a new OTP, the HSM generates an OTP (`GenerateKey`), encrypts it using the shared secret (`WrapKey`), and exports the encrypted OTP to the client. The client then obtains the OTP by decrypting it using the shared secret and uses the OTP to encrypt a client-side key or data before sending it to the HSM.

Note that a naïve design of using the vHSM key pair instead of an OTP is not secure, because malicious insiders can issue HSM commands to use the vHSM private key to recover the client's key/data after the client finishes key management operations. Thus, ScaleTrust uses the ephemeral OTP, which is erased from the HSM memory (`DestroyObjects`) at the end of a key management operation, thus mitigating insider threats. To prevent the recovery of the OTP after its destruction, ScaleTrust set key attributes of the shared secret to be non-extractable and wrap-only and the vHSM key to be non-extractable and unwrap-only, as shown in Table I, and disables the `modifiable` attribute of the keys, preventing the modification of key properties. ScaleTrust also detects key abuse during the key establishment phase by conducting log verification (explained in § VI-D).

PKCS #11 API	Additional protections	Restrictions
Key generation	Key encryption before store	None
Encrypt	Data encryption in request	Size $\leq$ 512 B
Decrypt	Data encryption in response	Size $\leq$ 512 B
Sign/Verify/(Un)Wrap	None	None

Table II: PKCS #11 APIs that ScaleTrust supports and additional security protections.

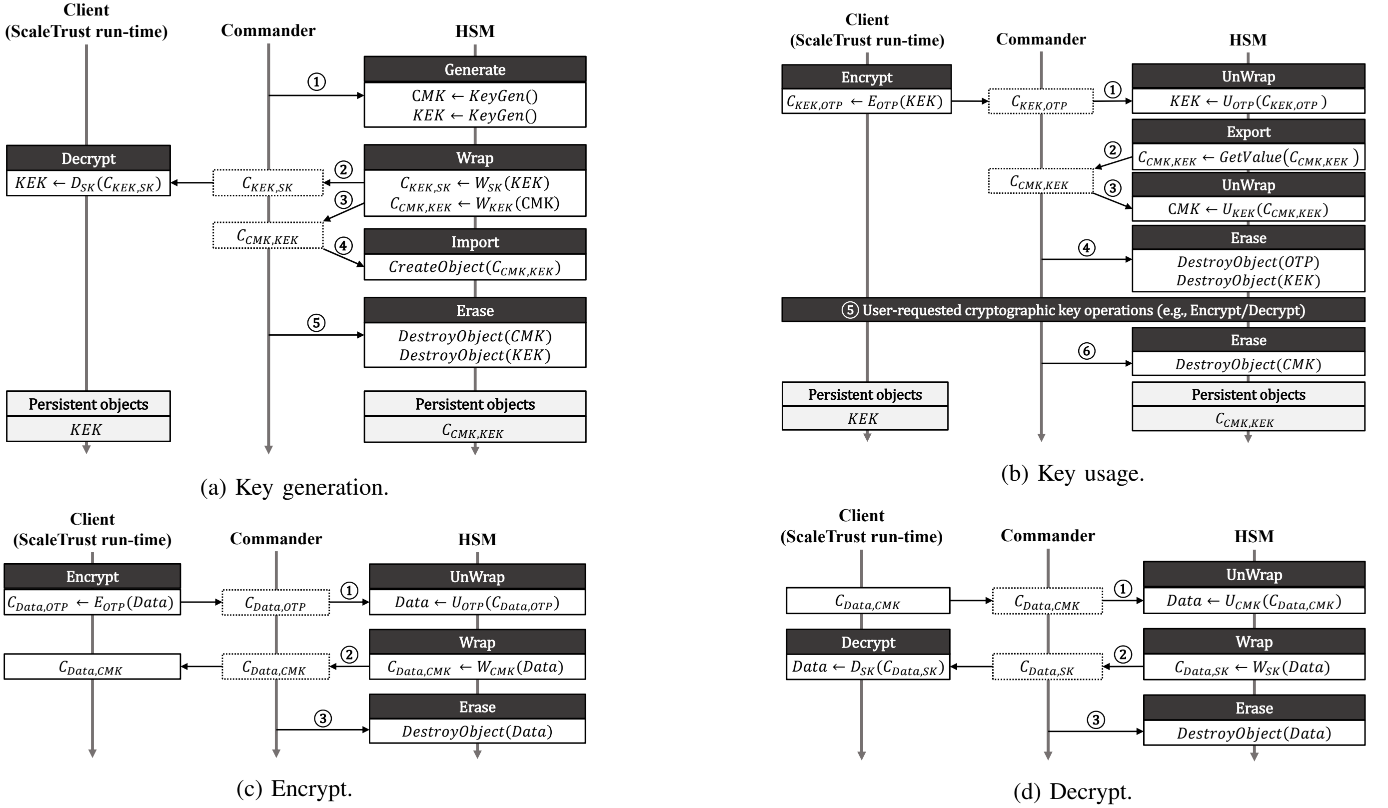
C. Key Management Operations

ScaleTrust exposes PKCS #11 APIs as existing KMSs; however, its underlying procedure is different. Unlike an existing KMS that simply relays a PKCS #11 request to an HSM, ScaleTrust additionally protects its data (e.g., keys or client data) from insider threats. To do this, its run-time library at a client communicates with the HSM commander and manages internal data such as additional keys for secure channels. Moreover, the library also encrypts and decrypts sensitive parameters for PKCS #11 APIs. In addition, the HSM commander translates a client's request into multiple HSM commands. Note, the commander can send commands only to the HSM, and the client has no direct access to the HSM (Figure 2).

ScaleTrust supports most PKCS #11 mechanisms with some restrictions, as shown in Table II. In particular, ScaleTrust encrypts and decrypts only 512 bytes data because ScaleTrust uses `WrapKey`/`UnwrapKey` for translation; these APIs deal with key data whose size can be 512-byte maximum. We believe that this restriction does not significantly limit the usability of ScaleTrust because existing KMS users rarely use KMSs for generic data encryption due to their prohibitive cost. They only utilize KMSs for important cryptographic operations such as signing or key encryption, which ScaleTrust supports.

We describe how ScaleTrust translates user-requested cryptographic operations into a series of PKCS#11 commands to HSMs.

Key generation. When a user makes a `GenerateKey` or `GenerateKeyPair` request, ScaleTrust performs the following steps, illustrated in Figure 3a. ① After receiving a request from the client, the commander calls PKCS #11 APIs to generate two keys in the HSM: a client master key (CMK) and an AES key encryption key (KEK). ② The commander shares KEK with the ScaleTrust run-time using a shared secret key (SK) from a secure channel in the bootstrapping step (§ VI-B). ③ Then, ScaleTrust makes HSM encrypt CMK using the KEK (i.e., `WrapKey`) and ④ imports this encrypted client master key back to the HSM (`CreateObject`). ⑤ Finally, the commander makes the HSM erase the plain-text client master key and the key encryption key (`DestroyObject`).



SK: Shared secret key (§ VI-B), OTP: One-time pad (§ VI-B), CMK: Client master key, KEK: Key encryption key, $C_{M,K}$: Cipher-text of message M encrypted using key K.

Figure 3: Key management operations in ScaleTrust.

After this key generation completes, the HSM stores the encrypted client master key ($C_{CMK,KEK}$), and the client stores the key encryption key (KEK). Thus, a malicious insider, who controls the HSM, is able to retrieve only the encrypted key, unless it compromises the client.

Key usage preparation. Unlike a vanilla HSM, in which a user can directly use plain-text keys, ScaleTrust requires additional steps to restore CMK from the encrypted keys of $C_{CMK,KEK}$ before using them, as shown in Figure 3b. ① When a user requests using a client master key in the HSM, the ScaleTrust run-time client sends the key encryption key (KEK)—which was received during key generation—to the HSM. Note, KEK is sent through the secure channel built with a one-time pad (OTP) (§ VI-B). ②-③ After sharing KEK, the commander exports $C_{CMK,KEK}$ from the HSM via `GetValue`, which was generated from the previous key generation procedure. It passes this $C_{CMK,KEK}$ back to the HSM to unwrap (`UnWrapKey`) it using KEK. This seemingly redundant procedure is necessary because PKCS #11 API does not support in-place decryption—decrypting an in-HSM key with another in-HSM key—so that the key to be decrypted must be exported first and re-entered with `UnwrapKey`. ④ Then, the commander sends requests to the HSM to erase OTP and KEK (`DestroyObject`) to prevent the adversary from obtaining the keys. ⑤ After that, ScaleTrust performs a user-requested cryptographic key operation such as encryption, decryption, or signing. Note that certain operations (e.g., encryption and decryption) may require additional procedures to support protection for their parameters such as encrypting data. Such additional procedures are discussed later. ⑥ After its use, ScaleTrust erases the plain-text client master key (CMK) from the HSM. CMK can be restored again when needed because $C_{CMK,KEK}$ remains in the HSM, and the client has KEK.

For optimization, ScaleTrust run-time allows its user to set a policy for batching. Based on this policy, ScaleTrust can perform multiple cryptographic operations at batch without erasing the keys in every operation. This allows ScaleTrust to avoid this series of key restore operations. Details of the optimization are explained in § VII.

Encryption. To support the `Encrypt` command in PKCS #11, ScaleTrust performs the following steps (Figure 3c), after the aforementioned key usage preparation. ① To make an encryption request, ScaleTrust's run-time encrypts its data using the one-time pad (`OTP`) and sends it to the commander. Then, the commander unwraps it into the HSM, which also has the same one-time pad from § VI-B. ② After that, the commander wraps data using a previously generated client master key (CMK) and then returns its results back to the run-time. ③ Finally, the commander erases its in-HSM data using `DestroyObject`, similar to other cryptographic operations. These steps ensure that the adversary is unable to observe any keys or plain-text data in ciphertexts. Thanks to the translation, ScaleTrust securely performs encryption while malicious insiders in the enclave never observe keys or data in plain-text.

Decryption. This decryption procedure (Figure 3d) is similar

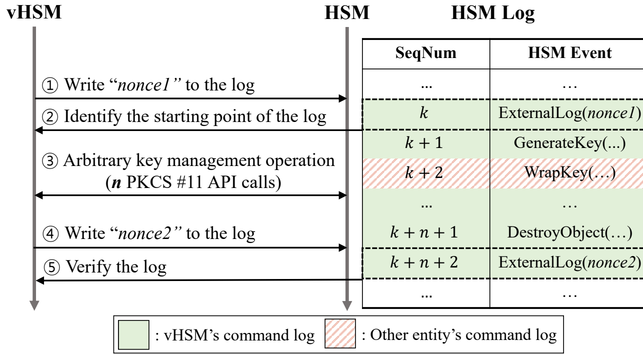


Figure 4: Log verification procedure.

to encryption. ① The run-time sends previously encrypted data with the sequence of commands to the commander. Then, the commander instructs the HSM to unwrap the data in plain-text using the client master key (`UnWrapKey`). ② The HSM executes `WrapKey` to encrypt the key using a shared secret key (§ VI-B), and the commander relays the encrypted key in the HSM response back to the run-time. ③ After that, the HSM erases the plain-text data (`DestroyObject`). Finally, the client run-time decrypts the encrypted key using the shared secret and returns it to the user. Unlike traditional KMSs, only the client and the HSM can see decrypted data, but not any intermediary, including the cloud platform.

Other cryptographic operations. PKCS #11 API also provides other cryptographic operations that are not covered in this paper. We categorize the types of cryptographic operations that ScaleTrust can/cannot support in an HSM. ScaleTrust can support cryptographic operations in the HSM if their input and output are either public data or key type data. For example, ScaleTrust can support both sign and verify operations that use only public data (e.g., certificates) and key type data (e.g., signing keys). However, ScaleTrust cannot support HSM operations that do not satisfy the previous condition. For example, ScaleTrust does not support `GenerateRandom` in an HSM because the HSM directly exports its result after processing. Therefore, a malicious insider can achieve this random value, which may result in serious security issues if this random value is used for cryptography. To remedy this, ScaleTrust delegates such operations to a client instead of using the HSM even though the HSM may provide better properties for randomness.

Note that both secure channel establishment (§ VI-B) and key management operations (§ VI-C) in ScaleTrust are vendor-neutral, because they only require PKCS #11 APIs support which is supported by most of commodity HSMs [29], [54], [83], [91], [99]. Moreover, SoftHSM [66]—a software implementation of an HSM device—can transparently support them without modification, as it also provides PKCS #11 interface [67].

D. Verifying vHSM Isolation

In the absence of an insider attacker that has access to HSM, all HSM access occurs through the commander, which provides effective isolation across vHSMs. However, in the presence of a malicious insider, ScaleTrust has to verify that the HSM did not execute any unauthorized commands that violates vHSM isolation.

Ensuring log freshness. To ensure the freshness of the log and thus prevent log replay, the ScaleTrust commander attaches nonce generated from the client at the beginning and end of each key management operation listed in § VI-C using the `ExternalLog` command. This also helps ScaleTrust to identify the beginning and end of a log sequence to be verified, as shown in Figure 4.

At the end of each client operation, the auditor retrieves the HSM log. Then, it matches the sequence of HSM commands—*PKCS #11 functions* and *key handle*—generated by the commander with the log. If any HSM command not issued by the commander is detected, the auditor deems the operation as unauthorized. If the auditor detects any unauthorized HSM commands, the auditor determines which keys are affected and reports the keys to the commander to revoke them. The auditor deems the keys used through unauthorized HSM commands and keys that are encrypted or decrypted by those as exposed, thereby revoking those keys.

Finally, HSM vendors differ in the log format. For each vendor, ScaleTrust requires a match function between the PKCS#11 commands and their log messages.

Log integrity verification. Finally, we must also verify the integrity of the HSM log. For this, when a log sequence passes the inspection, the auditor enclave takes the entire log and verifies the integrity of the batched log sequence.

Integrity verification requires an external HSM because a MitM attacker on the same platform can maliciously manipulate the verification results. However, HSMs do not allow users to export their log HMAC key in plain-text. Rather, it can only be shared with HSMs from the same vendor through encryption with a vendor-specific, non-extractable key. To overcome this limitation, ScaleTrust provides two options: 1) a public log verification service provided by the HSM vendor, or 2) another cloud platform using the same vendor HSMs. Note that some HSM vendors already provide on-demand HSM services [62], [82] that can be used as a remote log verification service, similar to Intel SGX's remote attestation [44]. The second option provides correct log verification results unless both cloud providers collude.

Either way, the auditor sends the log batch and encrypted log HMAC key to the external service and receives the result using TLS. If the result indicates that the log integrity was broken, the enclave cannot determine which in-HSM keys were abused, so it must revoke all HSM-derived keys.

Note that the log verification in ScaleTrust is also vendor-neutral, as it is compatible with other vendors' HSMs with only minor modifications. The FIPS 140-2 [1] certification of security level 2 or above mandates that cryptographic modules support auditing and provide protection against modification of the audited events. For example, HSMs implemented by Thales [83] and Yubico [99] provide a chained HMAC [80] and a chained hash digest [100] for each log line, respectively. ScaleTrust can easily support HSMs from any vendors by utilizing their log integrity verification mechanisms in the auditor's verification logic.

Threats on KMS	ScaleTrust	HSM	TEE	HSM-TEE
Attacks on Enclaves				
Side-channel attacks	M	N/A	✗	✗
Attacks on Cloud Instances (e.g., Enclaves) ↔ HSMs				
Eavesdropping	P	✗	N/A	P
Manipulating data messages	P	✗	N/A	P
Manipulating HSM commands	D	✗	N/A	✗
Attacks on Clients				
Client master key leakage	P	P	P	P

HSM: HSM-backed Cloud KMS, **TEE**: TEE-only Cloud KMS, **HSM-TEE**: HSM-TEE hybrid KMS, **P**: Prevention, **M**: Mitigation, **D**: Detection.

Table III: Comparison of attacks and defenses.

VII. Implementation and Optimization

We implement 7,148 lines of C++ code for ScaleTrust (4,899 lines for ScaleTrust's enclave implementation), including the HSM commander and log auditor logic, a TLS interface for clients, and SGX ocall wrappers. We use Intel SGX Linux SDK 2.8 [43] for SGX implementation and port OpenSSL 1.0.2l [68] for use in the SGX enclave. Also, we use the PKCS #11 API library provided by Thales [85]. Note that we implement the HSM commander thread to handle its cryptographic workloads exclusively in the enclave-protected region, and its TLS connections are terminated in the enclave.

Optimization for master key sign operations. Because a client master key restore requires heavy computation compared to a signing operation, restoring the master key for every signing operation incurs substantial latency. To avoid such overhead, ScaleTrust erases its plain-text signing key from the HSM only after performing signing operations up to the pre-determined threshold and then restores the key when the next signing request arrives. For example, ScaleTrust can be set to erase and restore its client master key for every 100 sign operations. Note that this does not make ScaleTrust vulnerable to insider threats because ScaleTrust verifies all uses of its signing key by performing log integrity verification in a batch each time it erases the key.

VIII. Security Analysis

We describe how ScaleTrust defeats the attacks described in our threat model § IV. To fully consider the security threats, we consider all threats in each entity (Clients, Enclaves, and HSMs) and communication channels between them (Clients ↔ Enclaves and Enclaves ↔ HSMs). Note that ScaleTrust is secure under attacks against HSMs and the channel between clients and enclaves because HSMs are trustworthy and reliable under our assumption, and ScaleTrust uses secure channels for communicating between clients and enclaves. Therefore, we discuss the remaining attacks in the following.

Table III compares the security of ScaleTrust with that of existing cloud KMSs. Existing KMSs can be categorized into HSM-backed cloud KMS, TEE-only KMS [19], [35], [53], and a hybrid approach [31] that utilizes a TEE-based KMS as a proxy of HSMs while allowing to migrate in-HSM keys to the KMS. These approaches suffer from serious security threats; HSM-based KMSs have no protection for insider threats, and TEE-based KMSs cannot mitigate side-channel attacks.

Attacks on Enclaves. ScaleTrust is secure under attacks that try to achieve client master keys stored in the cloud platform.

No.	HSM operation	Arguments (Key handles)
1	EXTERNAL_LOG	Nonce1
2	GENERATE_KEY_PAIR	h_CMK_private, h_CMK_public
3	GENERATE_KEY	h_KEK
4	WRAP_KEY	h_KEK ← h_SK
5	WRAP_KEY	h_CMK_private ← h_KEK
6	UNWRAP_KEY	h_SK2 ← h_vHSM_private
7	WRAP_KEY	h_CMK_private ← h_SK2
8	CREATE_OBJECT	-
9	DESTROY_OBJECT	h_CMK_private
10	DESTROY_OBJECT	h_KEK
11	EXTERNAL_LOG	Nonce2

Table IV: Simplified HSM logs when a key abuse occurs while establishing a client master key pair. Red logs represent the HSM commands requested by attackers. (The raw version of the logs is presented in Appendix Figure 12.)

For example, in our case, an attacker may try to leverage side-channel attacks to achieve the client master keys. However, ScaleTrust is secure against the attack because it deliberately stores its client master key only in encrypted forms. Even if attackers break the confidentiality of ScaleTrust enclaves, they can expose only the encrypted key. Without knowing the key encryption key stored in a client, the attacker never learns any sensitive data. In contrast, both TEE-only and HSM-TEE hybrid approaches rely on the confidentiality of SGX enclaves, which makes them vulnerable to side-channel attacks.

Attacks on Enclaves ↔ HSMs. We consider three types of attacks against the communication channel between the enclave and HSM: eavesdropping, manipulating data messages (key objects), and manipulating HSM control messages (HSM commands).

ScaleTrust is resistant to eavesdropping (i.e., passive attacks) because ScaleTrust ensures that all keys are encrypted between the enclave and the HSM. Note that traditional HSM-backed cloud KMS is vulnerable to a similar type of attack because attackers residing in cloud instances (VMs) can observe plain-text keys.

ScaleTrust also protects keys against active attacks that manipulate data messages, preventing sensitive data leakage. Except for the shared secret key in § VI-B, ScaleTrust encrypts all keys using authenticated encryption (AES-KWP [27]) during transmission. Therefore, if an attacker modifies this encrypted key, a decryption failure will occur at either the client or the HSM. ScaleTrust is also resilient to injecting a fake secret key. Note that attackers can set an arbitrary secret key using the RSA public key from the HSM. However, this incurs a discrepancy between keys in the client and ones in the HSM, which results in a decryption error in subsequent key management procedures. Note that decryption failures in ScaleTrust, at best, lead to denial-of-service for key management.

ScaleTrust also limits the impacts of active attacks that manipulate HSM commands to abuse in-HSM keys in the following way. ScaleTrust's HSM has permanent keys and ephemeral keys. Permanent keys are either protected by HSM key attributes (e.g., vHSM key) or encrypted (e.g., client master key) by the key encryption key (§ VI-C). Moreover, ScaleTrust can detect any abuse on ephemeral keys thanks to its log verification (§ VI-D). After their use, ephemeral keys are safely erased from the system.

Attack detection (working example).

Table IV shows a real working example in which ScaleTrust

detects a key abuse during the establishment of a client master key pair. Following the procedure described in § VI-C, ScaleTrust generates a client master key pair (log sequence number 2), stores the encrypted master private key in the HSM (8), and then erases the key from the HSM memory (9). However, in this example, an attacker intentionally imports her secret key to the HSM (6) before the master private key is erased from the HSM memory, and then exports the master private key using the key (7).

Our log verification detects the attack because it confirms whether the HSM logs match the HSM commands sent by the HSM commander enclave. Thus, it can detect unauthorized commands (6-7). Also, the attacker cannot modify the log messages because ScaleTrust verifies log integrity, as described in § VI-D. Finally, ScaleTrust revokes the affected keys when it detects an attack.

Worst-case security impact. Although ScaleTrust always detects attacks that manipulate HSM commands, it cannot prevent such attacks. However, ScaleTrust reduces their security impact compared to existing HSM-based KMSs because only plain-text client master keys that remain in the HSM memory at that moment (i.e., in-use keys) can be leaked. The rest of the client master keys are stored in an encrypted form. Potential vulnerability of in-use keys is a fundamental problem of HSMs because they must have plain-text of the keys when using them, which inevitably leaves a window of vulnerability. However, compared to HSM-backed cloud KMS or HSM-TEE hybrid KMS, ScaleTrust improves the security level because it eventually detects abuse and limits the security impact to currently restored keys. Note that ScaleTrust revokes all in-use keys when an attack is detected. We believe that, when clients adopt perfect forward secrecy, the security impact of its leakage is significantly reduced.

Attacks on clients. ScaleTrust also protects client master keys under data leakage in a client system the same as other KMS systems. ScaleTrust decrypts and uses the encrypted client master keys only in the HSM and never exports them to clients. The client master keys can be compromised only if the client itself is compromised *and* the attacker obtains the encrypted keys stored in the HSM.

IX. Applications of ScaleTrust

ScaleTrust can be used as a drop-in replacement for KMS for all services that require secure key management. We select four popular cloud services that require key management. Two of them, Keyless SSL and JSON Web token, originate from a Web service, and the other two involve data encryption. ScaleTrust provides PKCS#11 APIs similar to a standard HSM, and applications that already use the interface can use ScaleTrust without any code modification.

Keyless SSL. Keyless SSL [63], proposed by CloudFlare, offloads TLS session key generation to a cloud platform while keeping a TLS private key in a trusted key server. It splits the TLS handshake to be performed by two servers: a key server that signs handshake messages with a TLS private key and a session server that handles the rest of the handshake (e.g., key exchange with a client). CloudFlare recommends deploying key servers in the client's infrastructure with HSMs [64] to ensure that the TLS private keys are protected.

As noted before, a client requires deploying the key server in his local infrastructure because the security of Keyless SSL only depends on that of the key server. Unfortunately, it is costly and tricky to maintain such a service locally. ScaleTrust allows us to securely deploy the key server of Keyless SSL even in the cloud platform. We can use ScaleTrust as the key server in the cloud environment because it securely stores TLS private keys by splitting them into the client and the server. We still have one restriction though; the client and the server should use cloud services from different providers since ScaleTrust can be broken if both entities are compromised. Then, ScaleTrust can protect TLS keys unless two cloud providers collaborate for attacks.

Using ScaleTrust, we build an ECDHE-RSA based Keyless SSL key server. It receives ECDHE parameters from its KMS clients (e.g., session servers) and signs the parameters using its TLS private keys.

JSON Web Token (JWT) [47] is a widely used open standard for securely transmitting information. JWTs are typically signed using a secret or a public/private pair and are used to transmit a digitally signed token for Web applications. For example, Microsoft's identity platform [59] supports a JWT service that produces access tokens for client authentication.

We build a JWT authentication service that signs a JWT using private keys in ScaleTrust. Our prototype JWT service uses EC (Elliptic Curve) keys with ES256 (ECDSA using P-256 curve and SHA-256) [22] to generate signatures.

DB key provisioning service [10] is used to generate symmetric keys that encrypt data stored in a DB system (e.g., DB tables or columns). For this, a DB key server utilizes a KMS to generate a DB encryption key in an HSM and exports the key. When exporting the newly generated DB key, a KMS usually sends both keys in plain and encrypted forms. Then, the DB key server uses the plain-text DB key for encryption. After the key is used, it removes the key from its memory to prevent key leakage. If the server wants to restore the same DB key, it sends the encrypted DB key to the KMS to decrypt the key. We also implement the DB key provisioning service using ScaleTrust. Our prototype supports provisioning 256-bits AES keys as DB encryption keys for its clients.

Key encryption at rest service [10] provides a key encryption server, which encrypts client-side keys using master keys stored in HSMs. A client uses this service to protect its keys by encrypting them while they are not being used. When the encrypted keys are needed again, the client makes requests to the server to restore (i.e., decrypt) the keys.

We build a prototype for a key encryption server that takes 256-bit AES keys from a client and encrypts them through an AES-KWP algorithm. For this, we use a master key in ScaleTrust, which is also a 256-bit AES key. Similar to the previous application, ScaleTrust ensures that the cloud provider never achieves the client's encryption keys in plain-text.

X. Evaluation

We evaluate ScaleTrust to answer three questions:

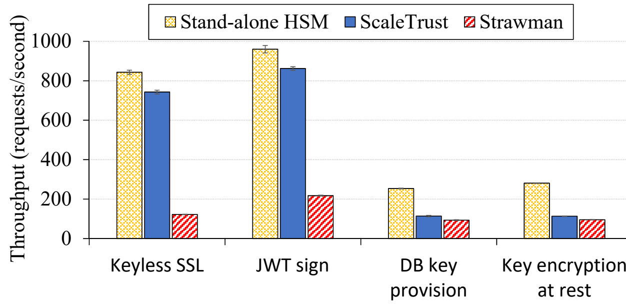


Figure 5: End-to-end throughput of KMS systems.

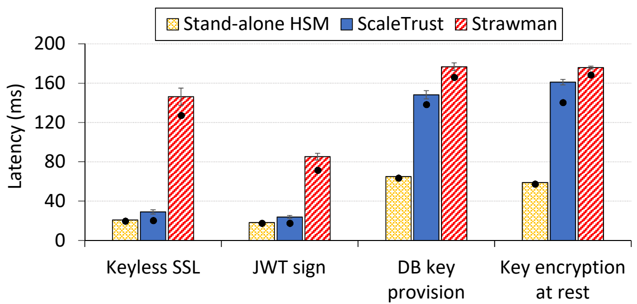


Figure 6: 95th-percentile latency of KMS systems. The circle mark on each bar denotes the median latency.

Operation	Stand-alone HSM # of commands G	E	S	O	ScaleTrust # of commands G	E	S	O
Master key signing	0	0	1	0	1*	3*	1	4*
Key provisioning	2	2	0	1	3	3	0	5
Key encryption at rest	1	2	0	0	2	4	0	5

G: Generate key (pair), E: Encrypt, decrypt, wrap, and unwrap, S: Sign, O: Other HSM commands (e.g., Destroy key), *: occurs only during master key restore.

Table V: The number of HSM commands required for each key management operation.

- RQ1: What is the performance overhead of ScaleTrust compared to a stand-alone HSM? (§ X-A)
- RQ2: Does ScaleTrust enable multi-tenancy? (§ X-B)
- RQ3: What is the performance impact of each ScaleTrust design component? (§ X-C)

Experiment setup. We evaluate KMS applications on three machines with Quad-core Intel Xeon E-2288 (3.70 GHz CPU, 8 physical cores) for a KMS server (HSM-equipped server), a KMS client (application server), and an end client (application client), respectively. The KMS server is equipped with a Thales Luna PCIe A700 HSM [85] whose firmware version is 7.4.0 and operates in FIPS-140-2 approved mode, and the KMS client machine is directly connected between the server and the end client through a 10 Gbps LAN cable. We set the end client to run ApacheBench [87] for generating back-to-back requests to the KMS client, which then sends HSM requests to the KMS server.

We compare ScaleTrust with two other systems: *stand-alone HSM (baseline)*, a KMS that uses a stand-alone HSM, and *strawman*, which provides the same key management APIs as ScaleTrust but serially relays HSM commands to serialize HSM logs, as described in § V.

We use the same PCIe HSM to perform log verification, instead of using an additional network HSM because we were not able to equip it in our testbed due to the high price. Instead, we emulate the network HSM with our PCIe HSM to include the log verification overhead in our measurement.

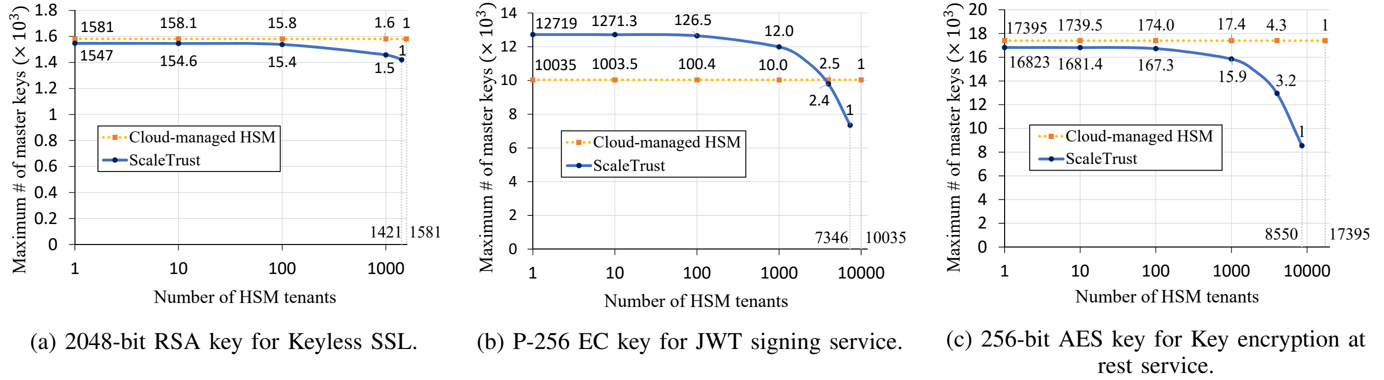
A. End-to-end Application Performance

We evaluate the end-to-end performance of each KMS system using four popular applications presented in § IX: Keyless SSL, JWT signing service, DB key provisioning, and key encryption at rest service. We measure throughput, 95th-percentile latency, and median latency at the end client (ApacheBench).

Figure 5 and Figure 6 respectively show throughput and latency (95th-percentile and median). For Keyless SSL and the JWT signing service, ScaleTrust incurs 11.9% and 10.2% of throughput degradation, and 39.0% and 31.1% of additional 95th-percentile latency, respectively, compared to the stand-alone HSM. For DB key provisioning and Key encryption at rest service, the overhead is relatively high; they respectively show 55.1% and 59.7% throughput degradation, and 128.2% and 173.3% additional latency. The reason for the larger overhead is that ScaleTrust executes more HSM commands to protect keys that are exposed to the untrusted cloud environment. Table V compares the type and number of HSM commands required for each key management operation for stand-alone HSM and ScaleTrust. Note that some operations (e.g., master key restore operations) are amortized over multiple sign operations performed in a batch. During our experiment, ScaleTrust executes 2.2 times more HSM commands than the baseline to perform key provisioning, while it requires 1.08 times more for a signing operation.

Compared to previous KMS studies and an existing cloud KMS, the performance overhead of ScaleTrust can be regarded as acceptable in practice. PALÆMON [35] took about 1200ms of processing time for protecting data and application execution, which was acceptable for a production application as it is less than 1500ms. Performance measurement results of SGX Barbican [19] showed that it took 1859ms for processing each request. Also, Microsoft Azure's Key Vault recommends setting an alert notification when latency exceeds 1000ms [60]. Our evaluation results show that ScaleTrust achieves end-to-end latency of less than 200ms which is acceptable for all the above cases.

The results also indicate that ScaleTrust outperforms strawman in all cases; from the leftmost application to the rightmost, ScaleTrust achieves 6.09x, 3.96x, 1.22x, and 1.18x more throughput and 0.20x, 0.28x, 0.84x, 0.92x of strawman's 95th-percentile latency, respectively. This is because Strawman cannot fully utilize the HSM through concurrent execution. ScaleTrust's relative improvement is higher with applications that require signing than with applications that mainly use key generation. This is because the signing is much cheaper than the key generation. In fact, signing throughput with a 2048-bit RSA key increases by up to 6.09x, whereas the throughput of 32-bytes AES key generation only increases by up to 1.16x.



(a) 2048-bit RSA key for Keyless SSL.

(b) P-256 EC key for JWT signing service.

(c) 256-bit AES key for Key encryption at rest service.

Figure 7: Maximum number of client master keys as the number of HSM tenants increases. The number above the marker represents the maximum number of keys divided by the number of HSM tenants.

Key stored in HSM	Key size (B)	Metadata size (B)	Total size (B)
Plain-text RSA key	1,224	96	1,320
Plain-text EC key	72	136	208
Plain-text AES key	32	88	120
Encrypted RSA key	1,232	116	1,348
Encrypted EC key	80	84	164
Encrypted AES key	40	84	124

Table VI: HSM key storage occupancy for each type of key.

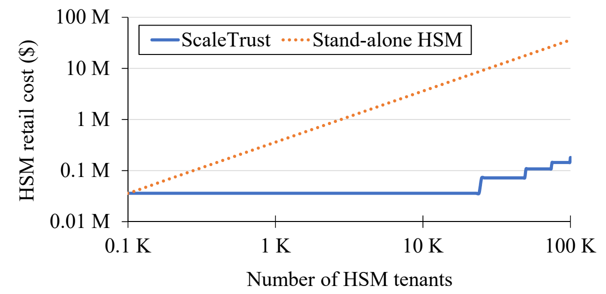


Figure 8: Estimated HSM retail cost as the number of HSM tenants increases. We assume that each tenant uses a 2048-bit RSA private key within an HSM.

B. Multi-tenant Scalability

Unlike our stand-alone HSM (Thales Luna A700) which provides only a single partition, ScaleTrust is no longer bound by the number of physical partitions in supporting multiple tenants. Instead, multi-tenancy is bound by the total size of HSM key storage (our HSM device provides about 1.99 MB). Although cloud-managed HSMs can also support multiple tenants up to the HSM key storage limit, they do not provide protection against insider threats, unlike ScaleTrust.

We evaluate the multi-tenant scalability of ScaleTrust by characterizing the key storage overhead. For comparison, we compare ScaleTrust with cloud-managed HSM that does not provide any isolation between tenants in the face of insider threats. To make the case realistic, we take the same applications from § IX, which use RSA, EC, and AES keys.

The key space that ScaleTrust occupies differs from that of the naïve cloud-managed HSM for two main reasons: 1) ScaleTrust introduces a per-HSM vHSM key and a per-tenant shared secret key (§ VI-B), and 2) in storing a private key, it stores an encrypted version instead of the plain-text key, unlike cloud-managed HSM.

When an HSM stores a key, it occupies the key size and the metadata size, which vary by key type. Table VI lists the size of keys used and the metadata they occupy in an HSM for ScaleTrust and Cloud-managed HSM.

Figure 7 shows the maximum number of client master keys each system can support as the number of tenants increases. The numbers above the markers on each curve denote the maximum number of keys for each HSM tenant. We observe that the overhead of ScaleTrust is moderate for two reasons: 1) The per-tenant shared secret key (256-bit AES key) occupies only 120 bytes. Even in the extreme case where each tenant uses one RSA private key in the HSM, secret keys occupy only about 8.14% of the HSM key storage. 2) The size of the encrypted master key is similar to that of plain-text keys. For applications that use RSA or AES keys with a single tenant, ScaleTrust respectively supports 2.15% and 3.29% fewer master keys than the cloud-managed HSMs. This is because when ScaleTrust encrypts master keys using the AES-KWP algorithm, the storage footprint of the key increases only by 8 bytes. Surprisingly, for applications that use EC keys, ScaleTrust provides more key space when the number of tenants is below 3,670. It turns out that the footprint of the encrypted EC key is actually smaller than its plain-text version. This is because plain-text EC keys have larger metadata (e.g., key attributes), as shown in Table VI.

Cost-effectiveness. Although we cannot make a fair comparison of costs between commercial cloud-managed HSMs and ScaleTrust, we provide an approximate comparison of costs between ScaleTrust and the legacy stand-alone HSM that isolates each client through HSM partitions. The legacy HSM limitation in multi-tenant scalability stems from the limited memory size of HSMs. Because each HSM partition divides the total HSM memory, the number of partitions is limited. For example, the most performant HSM from Thales has limited memory up to 32MB [84], which supports up to 100 partitions, with each partition assigned only 0.32MB of memory.

For this reason, a stand-alone HSM, which relies on HSM

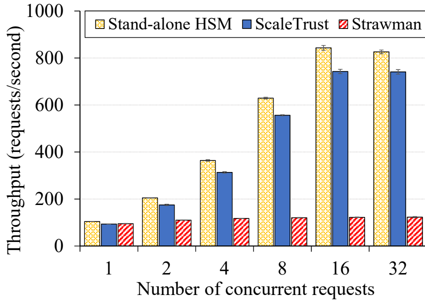


Figure 9: End-to-end throughput of Keyless SSL.

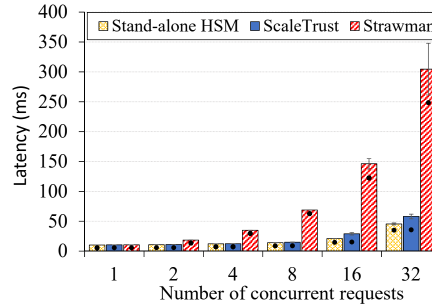


Figure 10: End-to-end latency of Keyless SSL. The circle mark denotes the median.

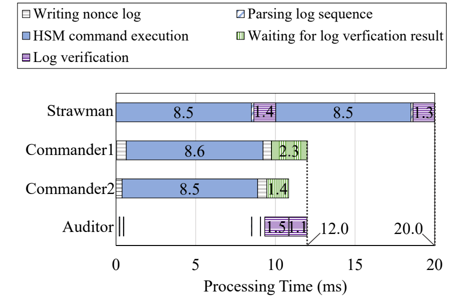


Figure 11: Comparison of processing time between ScaleTrust and Strawman.

partitions to provide isolation, requires significant capital investment to be scaled out for multi-tenancy. Note that HTTPS-based hosting service providers often manage tens of thousands of private keys for multiple clients [18]. With stand-alone HSMs, a service provider should deploy a new HSM for every 100 clients to isolate each client's private keys through HSM partitions. In contrast, ScaleTrust provides isolation that is not limited to the number of HSM partitions, and it can handle 24.9 K RSA private keys using a single HSM.

Figure 8 shows the comparison of estimated HSM cost between ScaleTrust and the stand-alone HSM as the number of HSM tenants increases to 100 K. We estimate the cost with the HSM which provides 100 partitions [84] and is priced at 36 K USD in retail cost [74]. The estimated result shows that the stand-alone HSM requires 1 K HSMs (36 M USD in retail cost) while ScaleTrust requires only 5 HSMs (0.18 M USD in retail cost) to support 100 K tenants.

C. Performance Breakdown

Benefit of concurrent execution. ScaleTrust leverages HSM's internal concurrency to achieve high throughput. To characterize this, we measure the throughput and latency (median and 95th-percentile) of each KMS system by increasing the number of end clients (i.e., HSM tenants) that concurrently make requests to the KMS. We take the Keyless SSL as a representative application because it is one of the most popular HSM use cases [2]. We vary the number of concurrent KMS requests from 1 to 32, as we observe that the HSM performance is saturated around the concurrency of 16.

Figure 9 and Figure 10 respectively show the throughput and latency when the number of concurrent requests is increased. Similar to the stand-alone HSM, ScaleTrust scales its throughput as the number of concurrent KMS requests increases from 1 to 16 (HSM saturated); the stand-alone HSM (baseline) and ScaleTrust throughput increase by up to 8.08x and 7.95x, respectively. ScaleTrust incurs throughput degradation up to 17.0% and increases the latency up to 39.0% against the stand-alone KMS. In contrast, strawman only scales its throughput by up to 1.29x and its latency increased almost proportionally to the number of concurrent requests because of its serialized execution.

Effect of concurrency and multi-threading. ScaleTrust utilizes multi-threading to take advantage of the HSM's concurrent execution. To demonstrate this, we instrument ScaleTrust and Strawman to provide timestamps for each operation during RSA signing with client master keys, and then we measure the processing time of each system when it receives two simultaneous signing requests. Figure 11 shows the measurement results. The result shows that the concurrent execution of commanders benefits from smaller tail latency. Compared to the strawman, which takes 19.95 ms to finish processing the two signing workloads, ScaleTrust takes 12.01 ms which is 60.12% of strawman's total processing time. We also note that the overhead of processing each log message is small enough to be handled by a single auditor thread, because parsing each log sequence takes only about 0.03 ms on average. The commander threads do not stall waiting to receive parsed log information from the auditor due to multi-threading.

XI. Discussion

We discuss how ScaleTrust can be FIPS compliant and which security properties ScaleTrust supports against TLS key leaks.

Complying with the FIPS 140-2 requirements. The actual FIPS certification requires accredited third-party testing [1]. However, we note that the ScaleTrust approach complies with the FIPS 140-2. ScaleTrust satisfies the FIPS 140-2 requirements [86], which are that 1) the system should only use FIPS-approved hardware and software and 2) master keys can only enter or leave the system in an encrypted form. In particular, ScaleTrust utilizes FIPS-validated HSMs that support physical security (e.g., tamper resistance) similar to the traditional HSM-backed cloud KMS [8], which already achieved FIPS 140-2 certification [9]. Also, ScaleTrust runs its HSMs in the FIPS-approved mode, which restricts the HSMs to use FIPS-approved cryptographic algorithms. For example, ScaleTrust uses the FIPS-approved `CKM_RSA_FIPS_186_3_PRIME_KEY_PAIR_GEN` function to generate its vHSM key (§ VI-B). Finally, ScaleTrust does not allow its plain-text client master keys to leave the HSM.

In addition, we believe that ScaleTrust can be applied to other types of third-party hardware devices other than HSMs if they support FIPS 140-2 compliant security properties such as secure cryptographic operations and event logging. Leveraging the security properties, ScaleTrust can enable isolated use of the hardware devices in an untrusted third-party environment.

Supporting security properties against TLS key leakage. To leak cryptographic keys stored in ScaleTrust's HSMs (e.g.,

	Compatible HSM	Approach to key protection
ScaleTrust	FIPS-compliant legacy	Isolate key usages in HSMs
SafetyPin	All legacy	Split keys over multiple HSMs
Fortanix	SGX-based HSM	Use security features of SGX
Myst	New HSM architecture	Enhance physical security

Table VII: Comparison of ScaleTrust with other HSM studies.

client master keys), the attacker must issue commands to the HSMs. However, since every HSM execution must leave an HSM log, such attacks are detectable through log verification (§ VI-D). If enclave's log verification result is forged due to the long-term TLS key leakage, it can be detected at the client by processing raw HSM logs periodically or confirming the genuine result directly from the public log verification service on which ScaleTrust depends. Note that the integrity of HSM logs is preserved by the chain of HMAC digests from HSMs regardless of TLS keys.

XII. Related Work

Our prior workshop publication [39] explores collaboration of HSMs and TEE-based KMSs, which reduces the burden of the HSMs by offloading frequent user-requested crypto operations to the KMS enclaves. In this extended version, we focus on a new problem, which is how we can scale multi-tenancy using HSM devices, with a completely new design that performs all crypto operations inside HSMs.

Hardware security module. SafetyPin [23] protects a key used for encrypting mobile backup data by splitting the key over multiple HSMs. Similar to ScaleTrust, it assumes malicious cloud providers can access the HSMs in a cloud platform. However, this study mitigates insider attacks by utilizing multiple HSMs to increase the attack cost, whereas ScaleTrust avoids insider threats by ensuring the isolation of key usages in HSMs through encryption. Fortanix's SGX-based HSMs [32] use Intel CPUs on HSMs to provide better scalability than legacy HSMs, and Myst [55] proposes a new architecture of cryptographic hardware that prevents some physical attacks such as hardware trojans. ScaleTrust, however, achieves the multi-tenant scalability using FIPS-compliant legacy HSMs that are already deployed or that are from preferred vendors by cloud providers [7], [42]. ScaleTrust also aims to be HSM vendor-neutral. Table VII summarizes the comparison of ScaleTrust with other HSM-related studies.

Attacks against SGX. Many research efforts study the security of SGX. For example, SGX has been attacked from various side-channel attacks by abusing the page fault [97], cache [34], branch target buffer (BTB) [51], and other micro-architectural vulnerabilities [73], [92], [93], [95]. Motivated by the side-channel attacks that reveal enclave content, ScaleTrust employs a design that does not rely on the confidentiality of enclaves. Unlike other generic yet expensive side-channel defenses using ORAM [37] or T-SGX [76], ScaleTrust employs dedicated design for the cloud KMS, incurring only modest performance overhead. Although, some attacks focus on undermining the integrity of SGX enclaves, our threat model considers them out of scope. For example, Lee et al. [50] and Biondo [17] have studied classical yet powerful memory corruption exploits, while SGX-Bomb [45] breaks SGX's integrity using the Rowhammer attack.

SGX applications. SGX has been widely used to protect security-sensitive applications. Haven [13], Graphene-SGX [88], SCONE [12], and Panoply [77] propose frameworks to support an unmodified application in SGX. VC3 [72], M2R [25], AirBox [16], and Ryoan [40] offload data processing to the cloud while ensuring user privacy. S-NFV [75] secures NFV states, and SGX-Tor [48] mitigates potential attacks on Tor network. SGX-Box [38], LightBox [26], and Safebricks [70] enable a secure middlebox on encrypted traffic. Civet [89] securely partitions Java applications within an enclave. OBLIVIATE [4], OBFUSCURO [3], ZeroTrace [71], Opaque [101], and OCQ [24] utilize SGX and accomplish performant oblivious execution to even hide the memory access patterns of a program. Unlike most prior work, we do not rely on the confidentiality of enclaves but address the problem of providing isolation of virtual HSMs.

XIII. Conclusion

ScaleTrust enables multi-tenant isolation of legacy HSMs within a cloud in the presence of a malicious insider. ScaleTrust supports most PKCS #11 operations for its users while defending against insider threats by translating the user's cryptographic requests into a series of HSM commands. This allows us to build virtual HSM partitions that support multiple tenants while providing isolation across tenants in a secure fashion. Our evaluation shows that ScaleTrust is a practical system that supports various real-world applications and achieves multi-tenancy scalability for key management; its performance is similar to that of a stand-alone HSM.

Appendix

Figure 12 shows the raw HSM logs of Table IV described in our security analysis (§ VIII). Each log contains a sequence number, a timestamp, a log message—an HSM operation and arguments (e.g., key handles)—, an HMAC, and an ASCII-HEX data record.

References

[1] FIPS PUB 140-2. *Security requirements for cryptographic modules*, 2001.

[2] 2021 Global encryption trends study. Technical report, Ponemon Institute, 2021.

[3] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. OBFUSCURO: A commodity obfuscation engine on Intel SGX. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[4] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[5] Alibaba Cloud. Alibaba Cloud Key Management Service. https://www.alibabacloud.com/product/kms.

[6] Amazon Web Services. Authenticating to PKCS #11. https://docs.aws.amazon.com/cloudhsm/latest/userguide/pkcs11-pin.html.

[7] Amazon Web Services. AWS CloudHSM. https://aws.amazon.com/cloudhsm.

[8] Amazon Web Services. AWS Key Management Service(KMS). https://aws.amazon.com/kms.

```
25102806,21/06/02 13:14:28,session 1868 Access 18084:0
        external message follows: 824BB9CC224D414E9CF5520DBF812E09F7935A0491B579C07933B432CEB40ED2,[HMAC],[Raw data record]
25102807,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_GENERATE_KEY_PAIR returned RC_OK(0x00000000) (generated private/public key handles=151205/151208),[HMAC],[Raw data record]
25102808,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_GENERATE_KEY returned RC_OK(0x00000000) (generated key handle=151197),[HMAC],[Raw data record]
25102809,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_WRAP_KEY returned RC_OK(0x00000000) (raw/wrapping key handles=151197/151215),[HMAC],[Raw data record]
25102810,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_WRAP_KEY returned RC_OK(0x00000000) (raw/wrapping key handles=151205/151197),[HMAC],[Raw data record]
25102811,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_UNWRAP_KEY returned RC_OK(0x00000000) (unwrapped/unwrapping key handles=151190/151265),[HMAC],[Raw data record]
25102812,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_WRAP_KEY returned RC_OK(0x00000000) (raw/wrapping key handles=151205/151190),[HMAC],[Raw data record]
25102813,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_CREATE_OBJECT returned RC_OK(0x00000000),[HMAC],[Raw data record]
25102814,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_DESTROY_OBJECT returned RC_OK(0x00000000) (object handle=151205),[HMAC],[Raw data record]
25102815,21/06/02 13:14:28,S/N 1431379664541 session 1868 Access 18084:0 operation
        LUNA_DESTROY_OBJECT returned RC_OK(0x00000000) (object handle=151197),[HMAC],[Raw data record]
25102816,21/06/02 13:14:28,session 1868 Access 18084:0
        external message follows: E409B3D20BF382576C41260754DC57C223B315189619C5D6EE37415559EA919A,[HMAC],[Raw data record]
```

*HSM log format: [Sequence number][Timestamp][Log message][HMAC][ASCII-HEX data record]

Figure 12: Raw HSM logs shown in Table IV. We omit an HMAC and an ASCII-HEX data record from each log for simplicity.

[9] Amazon Web Services. FIPS 140-2 Non-Proprietary Security Policy: AWS Key Management Service HSM. https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3617.pdf.

[10] Amazon Web Services. How Amazon DynamoDB uses AWS KMS. https://docs.aws.amazon.com/kms/latest/developerguide/services-dynamodb.html.

[11] Amazon Web Services. Working With Amazon CloudWatch Logs and AWS CloudHSM. https://docs.aws.amazon.com/cloudhsm/latest/userguide/get-hsm-audit-logs-using-cloudwatch.html.

[12] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[14] J.-B. Bédrune and G. Campana. Everybody be Cool, This is a Robbery! In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2019.

[15] J. G. Beekman and D. E. Porter. Challenges For Scaling Applications Across Enclaves. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)*, Shanghai, China, Oct. 2017.

[16] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan. Fast, scalable and secure onloading of edge functions using AirBox. In *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, pages 14–27, 2016.

[17] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[18] F. Cangialosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measurement and analysis of private key sharing in the https ecosystem. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[19] S. Chakrabarti, B. Baker, and M. Vij. Intel® SGX Enabled Key Manager Service with Openstack Barbican. *arXiv preprint arXiv:1712.07694*, 2017.

[20] L. Chen. Microservices: Architecting for continuous delivery and devops. In *Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[21] CloudFlare. The DNSSEC Root Signing Ceremony. https://www.cloudflare.com/dns/dnssec/root-signing-ceremony.

[22] Connect2id. ES256. https://www.javadoc.io/doc/com.nimbusds/nimbus-jose-jwt/6.0/com/nimbusds/jose/JWSAlgorithm.html#ES256.

[23] E. Dauterman, H. Corrigan-Gibbs, and D. Mazières. SafetyPin: Encrypted backups with human-memorable secrets. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2020.

[24] A. Dave, C. Leung, R. A. Popa, J. E. Gonzalez, and I. Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[25] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2R: Enabling stronger privacy in MapReduce computation. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[26] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren. LightBox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[27] M. Dworkin. Recommendation for block cipher modes of operation: methods for key wrapping. *NIST Special Publication*, 800:38F, 2012.

[28] EFTLAB. HSMs in a Payment Industry. https://www.eftlab.com/hsms-in-a-payment-industry.

[29] Entrust. Entrust nShield Connect HSMs. https://www.entrust.com/-/media/documentation/datasheets/entrust-nshield-connect-ds.pdf.

[30] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[31] Fortanix. Fortanix Delivers First Hybrid Cloud Data Security Solution that Integrates Cloud Native Applications with Legacy HSMs. https://fortanix.com/company/pr/2020/04/fortanix-delivers-first-hybrid-cloud-data-security-solution-that-integrates-cloud-native-applications-with-legacy-hsms/.

[32] Fortanix. Fortanix Hardware Security Module. https://resources.fortanix.com/fortanix-hardware-security-module-solution-brief.

[33] Google. Google Cloud HSM. https://cloud.google.com/hsm.

[34] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[35] F. Gregor, W. Ozga, S. Vaucher, R. Pires, D. L. Quoc, S. Arnautov, A. Martin, V. Schiavoni, P. Felber, and C. Fetzer. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 502–514. IEEE, 2020.

[36] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[37] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[38] J. Han, S. Kim, J. Ha, and D. Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet)*, pages 99–105, 2017.

[39] J. Han, S. Kim, T. Kim, and D. Han. Toward Scaling Hardware Security Module for Emerging Cloud Services. In *Proceedings of the*

4th Workshop on System Software for Trusted Execution (SysTEX), pages 1–6, 2019.

[40] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[41] IBM. Connecting to IBM Cloud HSM. https://cloud.ibm.com/docs/hardware-security-modules?topic=hardware-security-modules-connecting-to-ibm-cloud-hsm.

[42] IBM. IBM Cloud HSM. https://www.ibm.com/cloud/hardware-security-module.

[43] Intel. Intel® Software Guard Extensions SDK for Linux* OS. https://github.com/intel/linux-sgx.

[44] Intel. Intel® SGX Attestation Service Utilizing Enhanced Privacy ID (EPID). https://api.portal.trustedservices.intel.com/EPID-attestation.

[45] Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)*, Shanghai, China, Oct. 2017.

[46] Jeff Barr. AWS CloudTrail Update – SSE-KMS Encryption & Log File Integrity Verification. https://aws.amazon.com/blogs/aws/aws-cloudtrail-update-sse-kms-encryption-log-file-integrity-verification.

[47] M. Jones, J. Bradley, and N. Sakimura. JSON web token (JWT). https://tools.ietf.org/html/rfc7519.

[48] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Mar. 2017.

[49] F. R. Konkel. The Pentagon isn't ready yet for classified information to be stored off-premise in the cloud. https://www.nextgov.com/emerging-tech/2015/02/dod-wants-physical-separation-classified-data-cloud-now/105753.

[50] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[51] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[52] J. Leiseboer and R. Griffin. PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40. http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html.

[53] S. Luo, Z. Hua, and Y. Xia. TZ-KMS: A Secure Key Management Service for Joint Cloud Computing with ARM TrustZone. In *Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, Bamberg, Germany, Mar. 2018.

[54] MARVELL. LiquidSecurity and NITROX HSM Adapters. https://www.marvell.com/products/security-solutions/nitrox-hs-adapters.html.

[55] V. Mavroudis, A. Cerulli, P. Svenda, D. Cvrcek, D. Klinec, and G. Danezis. A touch of evil: High-assurance cryptographic hardware from untrusted components. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[56] 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 (HASP)*, Tel-Aviv, Israel, June 2013.

[57] Microsoft. Microsoft Azure Dedicated HSM. https://azure.microsoft.com/en-us/services/azure-dedicated-hsm/.

[58] Microsoft. Microsoft Azure Key Vault. https://azure.microsoft.com/en-us/services/key-vault/.

[59] Microsoft. Microsoft identity platform. https://developer.microsoft.com/en-us/identity.

[60] Microsoft. Monitoring Azure Key Vault - Alerts. https://docs.microsoft.com/en-us/azure/key-vault/general/monitor-key-vault#alerts.

[61] Microsoft Azure Dedicated HSM. Frequently asked questions (FAQ). https://docs.microsoft.com/en-us/azure/dedicated-hsm/faq#does-azure-dedicated-hsm-offer-password-based-and-ped-based-authentication-.

[62] Ncipher. Subscription-Based HSMs. https://www.ncipher.com/solutions/subscription-based-hsms.

[63] Nick Sullivan. Keyless SSL: The Nitty Gritty Technical Details. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/.

[64] Nick Sullivan and Chris Broglie . Going Keyless Everywhere. https://blog.cloudflare.com/going-keyless-everywhere/.

[65] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek. TRUSTORE: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2020.

[66] OpenDNSSEC. SoftHSM. https://www.opendnssec.org/softhsm.

[67] OpenDNSSEC. SoftHSMv2 GitHub. https://github.com/opendnssec/SoftHSMv2.

[68] OpenSSL Software Foundation. OpenSSL – Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[69] V. Phegade, J. Schrater, A. Kumar, and A. Kashyap. Self-Defending Key Management Service with Intel® Software Guard Extensions. https://www.intel.co.uk/content/dam/www/public/us/en/documents/technology-briefs/fortanix-sdkms-with-sgx-whitepaper.pdf.

[70] R. Poddar, C. Lan, R. Ada Popa, and S. Ratnasamy. SafeBricks: Securing Network Functions in the Cloud. In *Proc. NSDI*. USENIX, 2018.

[71] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[72] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[73] 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 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[74] SHI. SafeNet Luna Network Hardware Security Modules S790. https://www.shi.com/Products/ProductDetail.aspx?SHISystemID=ShiCommodity&ProductIdentity=35980058.

[75] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV*, New Orleans, LA, Mar. 2016.

[76] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[77] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB linux applications with SGX enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[78] B. Sumitra, C. Pethuru, and M. Misbahuddin. A Survey of Cloud Authentication Attacks and Solution Approaches. *International journal of innovative research in computer and communication engineering*, 2(10):6245–6253, 2014.

[79] Thales. Application Partitions. https://thalesdocs.com/gphsm/luna/7.4/docs/network/Content/administration/partitions/partitions.htm.

[80] Thales. Audit Logging. https://thalesdocs.com/gphsm/luna/7.1/docs/network/Content/administration/audit/audit_overview.htm.

[81] Thales. Confirm the HSM's Authenticity. https://thalesdocs.com/gphsm/luna/7.1/docs/network/Content/configuration/confirm/confirm_hsm.htm.

[82] Thales. Data Protection On Demand. https://cpl.thalesgroup.com/encryption/cloud-hsm-services-on-demand.

[83] Thales. Luna General Purpose HSMs. https://cpl.thalesgroup.com/encryption/hardware-security-modules/general-purpose-hsms.

[84] Thales. Thales Luna Network HSM. https://cpl.thalesgroup.com/encryption/hardware-security-modules/network-hsms.

[85] Thales. Thales Luna PCIe HSM. https://cpl.thalesgroup.com/encryption/hardware-security-modules/pcie-hsms.

[86] Thales. What is FIPS 140-2? https://www.thalesesecurity.com/faq/key-secrets-management/what-fips-140-2.

[87] The Apache Software Foundation. The Apache HTTP Server Project. http://httpd.apache.org/.

[88] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[89] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *Proc. Security*. USENIX, 2020.

[90] Utimaco. How HSMs support secure multi-tenancy? https://hsm.utimaco.com/blog/how-hsms-support-secure-multi-tenancy.

[91] Utimaco. SecurityServer Se Gen2. https://hsm.utimaco.com/products-hardware-security-modules/general-purpose-hsm/securityserver-se-gen2.

[92] 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 *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[93] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[94] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf, 2020.

[95] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[96] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *Proceedings of the 42th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2021.

[97] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[98] T. Yarygina and A. H. Bagge. Overcoming Security Challenges in Microservice Architectures. In *IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE, 2018.

[99] Yubico. Developer guide for YubiHSM2—GET LOG ENTRIES. https://developers.yubico.com/YubiHSM2/Commands/Get_Log_Entries.html.

[100] Yubico. YubiHSM 2. https://www.yubico.com/us/product/yubihsm-2.

[101] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. NSDI*. USENIX, 2017.

Juhyeng Han received his B.S. in the School of Computing from KAIST in 2016. He received his M.S. in 2018 and Ph.D. in 2022 in the School of Electrical Engineering from KAIST. His research interests are network systems and network security.

Insu Yun is an assistant professor at KAIST. He is interested in system security in general, especially, binary analysis, automatic vulnerability detection, and automatic exploit generation. His work has been published to the major computer conferences such as IEEE Security & Privacy, USENIX Security, and USENIX OSDI. Particularly, his research won the best paper award from USENIX Security and OSDI in 2018. In addition to research, he has been participating in several hacking competitions as a hacking expert. In particular, he won Pwn2Own 2020 by compromising Apple Safari and won DEFCON CTF in 2015 and 2018, which is the world hacking competition. Prior to joining KAIST, he received his Ph.D. degree in Computer Science from Georgia Tech in 2020.

Seongmin Kim is an assistant professor in the Department of Convergence Security Engineering at Sungshin women's university. He received his Ph.D. in the Graduate School of Information Security from KAIST in 2019. He received his B.S. in 2012 and M.S. in 2014 in the School of Electrical Engineering from KAIST. His research interests are system security (espeically, trusted computing and network security).

Taesoo Kim (Member, IEEE) is an Associate Professor in the School of Cybersecurity and Privacy and the School of Computer Science at Georgia Tech. He also serves as a director of the Georgia Tech Systems Software and Security Center (GTS3). Starting from his sabbatical year, he works as a VP at Samsung Research, leading the development of a Rust-based OS for a secure element. He is a recipient of various awards including NSF CAREER (2018), Internet Defense Prize (2015), and several best paper awards including USENIX Security'18 and EuroSys'17. He holds a BS from KAIST (2009), a SM (2011) and a Ph.D. (2014) from MIT.

Sooel Son (Member, IEEE) is an associate professor of School of Computing at KAIST. He received his Ph.D. in the department of computer science at the University of Texas at Austin. He is working on various topics regarding web security and privacy. He received a best student paper award at NDSS 2013.

Dongsu Han (Member, IEEE) received the B.S. degree in computer science from KAIST in 2003, and the Ph.D. degree in computer science in 2012 from Carnegie Mellon University. He is currently an associate professor at KAIST in the School of Electrical Engineering and Graduate School of Information Security. He is interested in networking, distributed systems, and network/system security. More details about his research can be found at http://ina.kaist.ac.kr. He is an associate editor of the IEEE/ACM Transactions on Networking.