Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

SRX – Secure Data Backup and Recovery for SGX Applications

DANIEL ANDRADE[1], JOÃO SILVA[2] (MEMBER, IEEE), AND MIGUEL CORREIA[3] (Senior Member, IEEE)

[1]INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal (e-mail: daniel.andrade@tecnico.ulisboa.pt)
[2]INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal (e-mail: joao.n.silva@inesc-id.pt)
[3]INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal (e-mail: miguel.p.correia@tecnico.ulisboa.pt)

Corresponding author: Miguel Correia (e-mail: miguel.p.correia@tecnico.ulisboa.pt).

This work was supported by the European Commission through contract 870635 (DE4A); by national funds through Fundação para a Ciência e Tecnologia (FCT) grant number UIDB/50021/2020; and by Universidade de Lisboa (ULisboa), Instituto Superior Técnico (IST), and Instituto de Engenharia de Sistemas e Computadores – Investigação e Desenvolvimento (INESC-ID).

ABSTRACT Intel SGX improves the security of applications by shielding code and data from untrusted software in enclaves. Since enclaves lose their state when closed, that state has to be sealed, i.e., cryptographically protected with a secret key, and stored outside the enclave boundary. In SGX, the used key is bound to both the enclave and the processor that sealed the data, so it is unfeasible for any enclave in another computer to derive the same secret key to unseal such data. This offers security to the data, but also makes it impossible to recover that data if the original computer is damaged or stolen. In order to support backup and recovery of data sealed by enclaves, we propose SRX, a solution for sharing sealed data amongst a restricted set of SGX-enabled computers executing the same enclave code. Enclaves using SRX have access to common keys to seal and unseal enclave data, allowing the sharing of sealed data among the trusted domain. SRX guarantees that these secret keys are never exposed outside the trusted domain. SRX was implemented and evaluated with two applications: a bitcoin wallet and a password manager.

INDEX TERMS Intel SGX, sealing, backup, recovery, TEE

I. INTRODUCTION

The Intel Software Guard Extensions (SGX) technology [1]–[3] enables user-level applications to allocate protected regions of memory to run sensitive code [4]–[6]. These regions of memory are called *enclaves* and ensure the confidentiality and integrity of the enclosed code and data, even from privileged software. A measurement of the initial enclave code and data is taken during its building process, and is used by verifiers during local or remote *attestation* [7] to prove that: (i) the code is executing inside an enclave, (ii) the platform (SGX-enabled computer) running the enclave has not been revoked, and (iii) that neither code nor data have been tampered with. A *platform* is an SGX-enabled *computer*, that is, a computer containing and using a *processor* that supports SGX.

Enclaves are stateless in the sense that they lose their state when closed, e.g., when the platform hibernates or shuts down. Their state can only be preserved if stored outside the enclave. For that purpose, the state is cryptographically protected before being exported, i.e., encrypted and added an integrity code. This encryption process is referred to as *sealing* and the secret key is called *sealing key* [2]. The sealing key is unique and bound to a specific enclave in a particular platform (i.e., bound to the processor), meaning that the sealed data can only be decrypted in the same enclave–platform pair.

The standard sealing procedure is adequate when only a single platform is supposed to access the enclave data, and there is no need to share data among multiple platforms. Since each processor has a unique sealing key per enclave, only the processor and enclave pair that sealed the data can unseal it.

This scenario can be relaxed to allow the data sealed by an enclave–platform pair to be unsealed by a different enclave signed by the same vendor and running on the same platform. This is useful when an application receives an update and needs to unseal data from an older version of itself. However, even in this case a processor is still unable to unseal data sealed by those same enclaves on other platforms because the sealing key continues to be bound to the processor. The intrinsic security characteristics of SGX become a shortcoming for data backups. If the platform that seals the data is lost then the user permanently loses access to the data even if there is a backup of the sealed data. A possible solution is to export private keys, but this is far from ideal, since it impairs the strong feature of SGX of preventing secrets from being accessible outside the enclave.

To allow data backup and recovery across platforms, we propose the *SGX Recovery Extension* (SRX) library for sharing sealed data between platforms of a *group*. In SRX, sealed data is data of an application enclave (e.g., the passwords' file of a password manager) that is stored outside the enclave boundary and only accessible by a configurable group of platforms. SRX allows the creation of groups G_i of platforms in such a way that an enclave e can access data sealed by itself or by members of its group of platforms. In other words, e can seal data in platform $p \in G_i$ and unseal that sealed data in platform $q \in G_i$ even if $p \neq q$. Isolation amongst enclaves remains in place and enclaves from other groups do not have access to data sealed by e. To accommodate this, SRX uses deterministic key pairs for an enclave–platform pair created by the SGX EGETKEY instruction and uses those key pairs in a key-exchange protocol between the platform and a common entity representing the group. To add legitimate platforms to the group, the SGX attestation mechanism guarantees the authenticity of the platform, along with a user authorization via a trusted user interface. The end user (i) is the single user of its own group, (ii) is dealing with a small group of computers it has access to, and (iii) manages backups and recovery of their own data. This is similar to how an end user would handle their backups.

SRX supports shareability and recovery of data. We use the term *shareability* to designate the ability of all platforms of a certain group to unseal data sealed by any other platform of the same group. Shareability allows applications to recover sealed data. *Recovery* means data sealed by one platform is accessible to all other platforms members of the same group, even after the platform that sealed the data becomes permanently unavailable.

SRX uses standard SGX mechanisms and has been implemented as a library, and integrated in two applications, to show the feasibility and applicability of our proposal. These test applications are a *bitcoin wallet* built on top of *libbtc* and a *password manager* based on *Titan*, and we use SRX to allow enclaves in the same group to unseal each other's sealed data. We evaluated SRX experimentally with the two applications. The results show an overhead under 30 ms to create the enclave, when comparing the SRX version with the SGX-only version of an application, and under 1 ms to subsequently retrieve a secret key accessible only to members of a particular group to unseal data. These two costs, enclave setup and deriving a secret key shared by group members, are paid only once and are independent of the size of the sealed data.

This works' contributions are: (i) a novel mechanism that enables the recovery of sealed data on an SGX-enabled platform different from the one that sealed the data, supporting securely moving data between platforms and data backup; and (ii) a prototype of SRX, its integration with two relevant applications we adapted to work within an enclave, and its experimental evaluation.

Section II introduces our approach, and presents the threat model and security properties of SRX. Section III details our sealing mechanism for SRX data and the protocols of SRX. Section IV describes the prototype and the two applications. Section V and Section VI evaluate the prototype. Section VII overviews related work. Section VIII presents our conclusions. The appendix presents a proof sketch of the correctness of the solution and a figure with the keying material derivation.

II. A RECOVERY EXTENSION

SRX provides new functionality not available in SGX. This section describes its application context, threat model, architecture, interface, and security properties.

A. APPLICATION CONTEXT

One possible use for SRX is a software house providing an enclaved password manager application to end users. Backups are an essential feature of a password manager, but sealed data cannot be decrypted by a processor different than the one that sealed that data. Having the enclaved password manager export the secret encryption key outside the enclave would expose that key to potential attackers who, additionally, could develop malware to exploit the export/import interface of such application, weakening the isolation benefits of SGX. Transferring the secret encryption key via a trusted service of the vendor would expose the keying material to the vendor and turn such service into a target for attackers.

SRX is a trusted library imported into enclave-based applications developed by an Independent Software Vendor (ISV) [8]. The ISV provides these enclaved applications to its employees and/or third parties as a service. These enclaved applications run *locally* on the end users' computers, but the ISV has access to a computational environment to run application support services that may be required.

For the example of the password manager, SRX supports backing up these passwords without the two mentioned problems, i.e., exporting the secret key outside the enclave and in the process exposing it to potential attackers, and trusting the ISV with the secret key. The ISV can build backup/recovery functionality on top of SRX for the benefit of end users, without exposing the secret encryption key outside the enclave, because SRX gives a secure mechanism to the programmer and user to share secret keys across SGX-enabled computers.

Fig. 1 provides a high-level view of two groups of platforms using SRX in their applications. Group 1 is composed of platforms p and q that share SRX sealed data $s_{x.1}$ with one another, and similarly, Group 2 is composed of platforms r, s and t that share SRX sealed data $s_{x.2}$ with each other. The secret keys that encrypt $s_{x.1}$ and $s_{x.2}$ are never exposed outside the respective enclaves, and this data is always encrypted before being transferred from one enclave to another or stored outside the enclave.

The transfer of encrypted data from one enclave to another is the responsibility of the application enclave software, developed by the ISV. SRX only provides the keying material to encrypt the enclave data.

B. THREAT MODEL

The trusted computing base (TCB) consists of the platforms' processors and enclaves. The SRX trusted library is statically linked with the application enclave and therefore implicitly part of the TCB. In addition, the Intel Attestation Service (IAS), controlled by Intel, is part of the TCB since it certifies enclaves as legitimate and up to date. The Remote Attestation Proxy (RAP), controlled by the ISV, is outside of the TCB. We trust the platforms' processors and assume all hardware is implemented correctly and that any guarantees provided by SGX cannot be circumvented. Side-channel attacks [9]–[12] are beyond the scope of this paper.

The adversary controls the network, all unprivileged and privileged software on the platforms, can initiate multiple instances of the same enclave, restart those instances, and restart the platforms.

In SRX, each group of platforms is composed of a small number of computers and controlled by one end user that manages the backup and recovery of their own data.

We assume the existence of an Authenticated Encryption with Associated Data (AEAD) scheme [13] that is both IND-CPA and INT-CTXT. This form of encryption provides confidentiality, integrity, and authenticity to the input plaintext, and integrity and authenticity of some Associated Data (AD), which is not encrypted. We also assume the existence of a Key Derivation Function (KDF) using a pseudorandom function and a 2-party key exchange protocol, Elliptic Curve Diffie-Hellman (ECDH) [14], that is secure against eavesdroppers.

AEAD [15] has functions `Encrypt(K,N,P,A)` $\rightarrow$ `C,T` for authenticated encryption and `Decrypt(K,N,C,A,T)` $\rightarrow$ `P` or `FAIL` for authenticated decryption, where `K` is a secret key, `N` is a nonce, `P` is the data to be encrypted and authenticated, `A` is the data to be authenticated but not encrypted, `C` is the ciphertext which is of the same length as the plaintext `P`, `T` is the authentication tag, and the `FAIL` error code means the input is not authentic. KDF has one function `KDF(salt,IKM,info,L)` $\rightarrow$ `OKM` where `salt` is an optional non-secret random value, `IKM` is the input keying material, and `info` is an optional string binding the derived output keying material, `OKM` of length `L`, to context-specific information. ECDH has one function `ECDH(KR,KU)` $\rightarrow$ `K` where `KR` is the private key of one entity A, `KU` is the public key of a distinct entity B, and `K` is a secret shared between A and B.

C. SRX OVERVIEW

This section presents an overview of the architecture and functioning of SRX.

1) Components

Fig. 2 shows the key components of the system and their logical connections. The architecture is composed of an SGX application containing an enclave using SRX, a trusted user interface, and a remote attestation proxy that communicates with Intel during attestation.

a: SGX applications

SGX applications, also known as *enclaved applications*, use Intel SGX to secure their secrets and are divided into two logical components, the trusted component (application enclave) and the untrusted component. The SRX trusted library is added to the trusted component by the ISV responsible for development of the application enclave. Requests made by the SGX application are understood to have originated in untrusted code and requests made by the application enclave are understood to have originated in trusted code (from within the enclave).

b: SRX

SRX is a trusted library, imported by the application enclave, that offers developers mechanisms to share secret keys across SGX-enabled computers. This enables a platform to unseal data that was previously sealed by a different platform without exposing the secret key outside the enclave and without requiring the two platform to establish a live communication channel.

In SGX, sealed data is bound to a single platform. Transferring data to a different platform, without exposing the secret key outside the enclave, can be achieved by using an encrypted communication channel between two mutually-attested enclaves.

In SRX, sealed data is bound to a group of platforms. Any platform in the group can unseal data sealed by other platforms in the same group by using a shared secret key provided by SRX. In comparison with SGX, this task can be achieved in SRX without having a live communication channel. This is useful for backups because data sealed by a platform using SRX can later be retrieved by a different platform even when the platform that sealed the data is unavailable because it is offline, or because, for example, it was damaged or stolen.

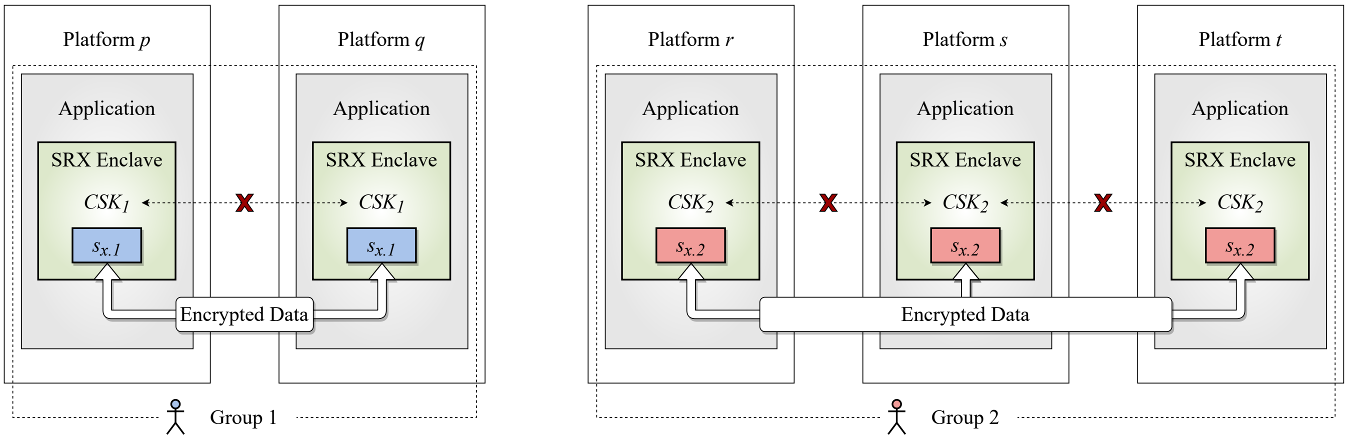

FIGURE 1. High-level view of SRX depicting two distinct groups of platforms. Platforms in Group 1 share SRX sealed data with one another, without exposing their Common Sealing Key (CSK_1) outside the respective enclaves, but not with platforms in Group 2 which have their own Common Sealing Key (CSK_2).

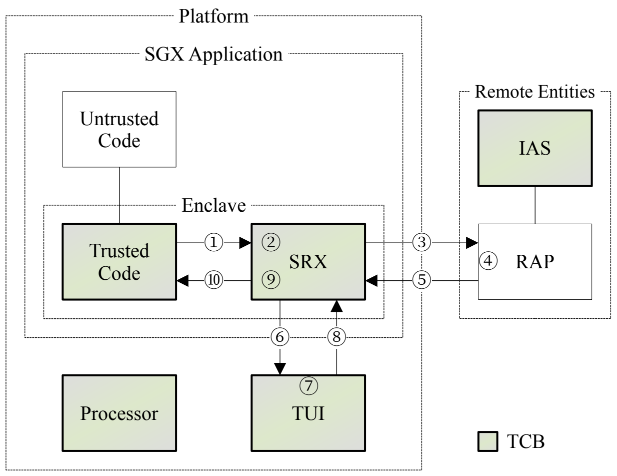

FIGURE 2. Typical architecture of a system using the SRX trusted library. The circled step numbers are used in the protocols' description (§ III).

SRX provides the building block to share data across SGX-enabled computers. Issues such as managing data backups, building a trusted path between application enclave and user, concurrency, and consistency are the responsibility of the ISV and should be implemented at the application level. The SRX trusted library is part of the trusted code and thus implicitly attested when the application enclave attests itself.

c: RAP

IAS attests the integrity and authenticity of the trusted code of the SGX application and is provided by, and under the control of Intel [16]. The RAP proxy communicates with IAS during remote attestation (§ III-G) forwarding enclave quotes from application to IAS and attestation verification reports from IAS to application. The ISV provides and controls RAP.

We assume there is a single RAP per ISV, serving all of their enclaved applications. For example, a company providing enclaved applications to its employees could use SRX to enable those applications to backup their sealed data and recover it in different platforms (in the same group), and all of these platforms would attest to the RAP deployed by the ISV.

Recall from Section II-B that RAP is outside of the TCB. If compromised it can block some SRX operations while not recovered (create group, add platform to group), but not steal any secrets.

d: TUI

SRX assumes the existence of a *Trusted User Interface* (TUI) that provides a trusted path from the ISV application enclave to the user. SRX uses the TUI for actions deemed sensitive, such as adding a new platform to the group (§ III-D) and requesting secret keys when policy requires it (§ III-F). Work in this area applied to SGX [17], [18] and other environments [19], [20] could be reused by ISV developers.

2) Functionalities

During typical operation, the application enclave provides SRX with an identifier and asks for the corresponding secret key. The secret key returned by SRX is then used by the application enclave to seal its application data. The application enclave can ask for multiple secret keys by providing SRX with different identifiers. SRX ensures all enclave–platform pairs in the same group can access the same set of secret keys.

Recovering from the latest sealed data may happen when the original processor that sealed that data becomes unavailable, for example, due to damage. Recovering from old sealed data is useful to rollback state to a previous version due to, for example, accidental deletion of data. Usually, accessing older data could be considered a rollback attack [21], [22] but in SRX rollback protection would hinder data recovery at the application level.

In SRX a group is composed of a few platforms and managed by a single end user who also handles their own backups, not multiple users on a distributed backup system or cloud environment. SRX stores a timestamp in s_x when it is modified, and this timestamp is displayed to the end user via TUI. SRX does not store a timestamp in application enclave sealed data, but it can be added by the ISV developer. The timestamp is not a security feature but it works as a memory aid for the end user. When the timestamp comes from the computer executing the enclave it is untrusted but this could be updated to use a trusted time source [23], [24].

Trusted timestamps can mitigate rollback attacks but proper rollback protection requires storing the persistent state, or integrity information pertaining to that state, of enclaves in non-volatile trusted memory or on a trusted server. Intel supports monotonic counters for rollback protection using non-volatile replay-protected storage on the platform [25]. However, monotonic counters are not a solution for SRX which works across multiple platforms. Using a trusted server available to all group platforms also has its drawbacks because it requires placing trust in yet another entity, namely that this trusted server will always provide its clients with the latest data and not serve obsolete data. This could be mitigated if Intel itself, which is already trusted and part of the TCB, complemented IAS with a centralized monotonic counter service accessible to all SGX-enabled computers.

Data synchronization and consistency is an orthogonal problem and out of scope for SRX. This should be solved by the ISV at the application-enclave level when invoking SRX functionality. For example, when needed the ISV should provide mechanisms for: copying data between platforms, solving inconsistencies namely merging data, and handling different versions of the data. SRX supports multiple groups with overlapping membership (each s_x represents a group with its own Group ID), however, modifying two copies of the same s_x (same Group ID) in different platforms in parallel is not supported by SRX and may cause the group to fork. For instance, updating the TCB (§ III-B) or removing a platform (§ III-E) triggers a seed update which leads to a fork when done over different copies of s_x. When that occurs, both instances of s_x retain access to past application enclave sealed data but have diverging future application enclave sealed data because the seed used for generating shared secret keys is different.

D. SRX PROGRAMMING INTERFACE

In the SGX programming model, an enclave call (ecall) is the invocation of a function located within the enclave from untrusted code, and an outside call (ocall) is the invocation of a function located outside the enclave made from within the enclave. These functions are specified by the ISV in an Enclave Definition Language [26] file containing the function prototypes of all trusted and untrusted functions. In SRX, the programmer continues to use this programming model to develop the trusted components of the application.

The application enclave can access SRX functionality using an interface called the *SRX Trusted API*. This API defines a set of functions that are implemented by the SRX trusted library and three functions that are implemented by the ISV (`auth`, `attest`, and `time`). The API does not provide encryption functions since SRX only provides the shared keying material, via `get_sk`, to the application enclave. The application enclave then uses this keying material to encrypt its data using encryption functions offered by the Intel SGX SDK or other third-party libraries. Table 1 provides an overview of the ecalls of this API and Table 2 lists its symbols.

The meaning of the symbols is the following: s_x represents SRX sealed data (d_x is the unsealed version of that data); d_p represents the details of a platform (pid, nonce, and public key); *pid* is the unique platform identifier; *gid* is the group identifier; n is a nonce; L is the length of the output secret key K; p is a policy (§ III-F); *spos* is the seed position; m represents a message; d_{in} is user data added to the quote sent from application enclave to IAS during remote attestation and d_{out} is the certified report replied by IAS during remote attestation.

Functions are grouped in three categories: group management, key management, and external interactions. Group management is composed of functions `setup` that initializes the SRX state essentially creating the group with the first platform (the one currently executing the enclave); `init_p` that initializes the metadata of some platform to be added to the group; `add_p` that adds a platform, previously initialized with `init_p`, to the group; `remove_p` that removes a platform from the group; and `list` that lists the platforms currently in the group. Key management is composed of functions `get_sk` that retrieves a secret K shared by all platforms in the group; and `update` that refreshes the keying material of the group, which is useful after a security update. External interactions is composed of functions `auth` which asks the user for authorization, via TUI, before proceeding with an operation described in m, for example, before adding a new platform to the group; `attest` that invokes the remote attestation protocol, for example, during group creation with `setup` and when initializing a platform with `init_p` to ensure the platforms being initialized (and then added) to the group are legitimate and up to date; and `time` that returns a timestamp that is added to s_x when it changes, for example, during security updates. Functions `auth`, `attest` and `time` are invoked internally by SRX during its operation, although the application enclave can also invoke these three functions, and are implemented by the ISV.

E. SECURITY PROPERTIES

The following security properties arise from the use of SRX:

1) *S1 Authorization:* Consider the set of functions $F = \{add\_p, remove\_p, get\_sk\}$ of the SRX API. For any function $f \in F$, f is only executed for a group G_i

TABLE 1. The Trusted API of SRX

Category	Function (ECALL)	Description
Group management	setup() $\rightarrow s_x$	Creates the group
	init_p() $\rightarrow d_p$	Initializes a new platform
	add_p(s_x, d_p) $\rightarrow s_x'$	Adds a new platform to the group
	remove_p(s_x, pid) $\rightarrow s_x'$	Removes a platform from the group
	list(s_x) $\rightarrow \{gid, pid_0, \ldots\}$	Lists group members
Key management	get_sk($s_x, n, L, p, spos$) $\rightarrow K$	Retrieves shared secret keys
	update(s_x) $\rightarrow s_x', spos$	Updates group keying material
External Interactions	auth(m, gid) $\rightarrow bool$	Requests user authorization via TUI *
	attest(d_{in}) $\rightarrow d_{out}$	Requests attestation with IAS via RAP *
	time() $\rightarrow timestamp$	Requests a timestamp *

* Implemented by the ISV.

TABLE 2. Symbols of the Trusted API of SRX

Symbol	Description
s_x	SRX sealed data
d_x	Unsealed version of s_x
d_p	Platform's details (pid, nonce, and public key)
pid	Unique platform identifier
gid	Unique group identifier
n	Nonce
K	Shared secret key
L	Length of output secret key K
p	Policy
$spos$	Seed position
m	Message displayed to user via TUI
d_{in}	User data added to quote during remote attestation
d_{out}	Certified report replied by IAS

without returning an error if the execution of f is requested by a platform $p \in G_i$ and the execution is authorized by the user.

2) *S2 Shareability:* If platform $p \in G_i$ seals data a obtaining s_a, then any other platform $q \in G_i$ with access to s_x and s_a can disclose the content of a.
3) *S3 Confidentiality:* No platform $q \notin G_i$ can disclose the content of a from s_x and s_a sealed by platform $p \in G_i$.

Appendix A presents an argument that SRX satisfies these security properties.

III. SRX PROTOCOLS

Section III-A details how SRX obtains the keying material that seals SRX data, d_x, and unseals SRX sealed data, s_x. Section III-B explains the mechanism for generating deterministic keys used in the previous section. The remaining sections describe the initialization protocol that creates the group (§ III-C), the group management protocols for adding platforms to the group (§ III-D) and removing platforms from the group (§ III-E), how the application enclave obtains secret keys, from the SRX component, for sealing its data, d_a, and unsealing its data, s_a (§ III-F), and how remote attestation works in SGX and SRX (§ III-G). The protocols' description follows the steps in Fig. 2 (signalled with ◯), and uses the ecalls from Table 1.

A. SEALING IN SRX

SRX implements a custom-built sealing mechanism based on an AEAD scheme: all keying material (nonces, public keys, and secret keys) is encrypted except if needed to derive the SRX sealed data decryption key, in which case it is only authenticated.

The intuition behind this custom-built sealing mechanism is that a combination of the ECDH key agreement [27] scheme and the SGX EGETKEY instruction [28] can be used to obtain the same secret key and initialization vector (provided as input to the AEAD algorithm) in all platforms of a group but not in any other platform.

Each platform in a group G_i has a Platform Sealing Key Pair (PSKP) which is used in unsealing s_x. A set of platforms, forming a group G_i, establish a secret among themselves using ECDH with their PSKP as input. These key pairs need to be derived deterministically, not randomly, by each platform because they are used in reconstructing the secret keying material that unseals s_x and are lost when the enclave is destroyed. For this purpose each platform retrieves a secret (to use as private key) from an SGX-based processor-specific key hierarchy via EGETKEY. The key pair of a platform is therefore bound to a specific enclave–platform pair much like the original SGX sealing key.

The ECDH public keys are static [29, § 6.3] and are validated by placing their hash in the user data field of the quote sent to IAS during remote attestation. IAS validates the platform, ensuring it is legitimate, and signs the quote implicitly certifying the keying material as belonging to a legitimate platform. The user authorizes the addition of a new platform to the group via TUI, further controlling which platforms can join the group (from all those that are legitimate to only those that are legitimate and authorized by the end user).

a: Sealed data

We consider two kinds of sealed data: SRX sealed data and application enclave sealed data.

- SRX sealed data (s_x) is group data (d_x when unsealed) accessible only to SRX and cryptographically protected using the AEAD scheme. This is where keying material is stored.
- Application enclave sealed data ($s_a, s_b, \ldots$) is application data ($d_a, d_b, \ldots$ when unsealed) accessible only to the application enclave and cryptographically protected using keying material derived from d_x.

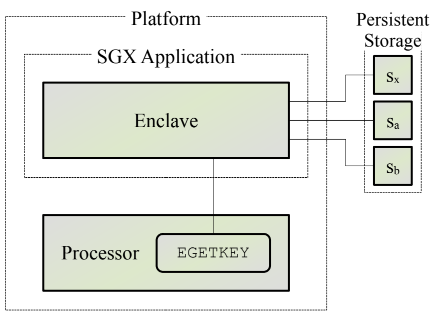

FIGURE 3. Storage of sealed data s_a of the application enclave and s_x of SRX.

Fig. 3 shows how the application enclave sealed data is stored. SRX and the application enclave seal and store their data independently from one another, that is, data is not sealed and stored in a single common bundle as one might expect. This is done to increase the flexibility: although SRX requires only one sealed data bundle, s_x, the application enclave can have many, $s_a, \ldots, s_n$. The secret shared by members of group G_i is used to unseal s_x, and within s_x we find the keying material to unseal $s_a, \ldots, s_n$. Fig. 4 shows the contents of s_x and s_a.

b: Derivation

Table 3 summarizes the keying material used in SRX where column *Entity* shows the owner and the symbol in the last column indicates whether the entry is stored in s_x encrypted and authenticated (🔒), is stored in s_x only authenticated (🔓), or is derived on demand from the keying material stored in s_x (🔁).

Fig. 10 in Appendix B depicts the derivation of the keying material for unsealing s_x and subsequently s_a; and the relationship between the public, private, and secret keys of SRX.

TABLE 3. Keying material

Entity	Name	Acronym	
Group	*seed*		🔒
	Base Key	BK	🔁
	Common Sealing Key	CSK	🔁
	Common Initialization Vector	CIV	🔁
	Group Sealing Public Key	GSKU	🔓
	Group Sealing Private Key	GSKR	🔒
	Encryption Nonce	EN	🔓
	Sealing Nonce	SN	🔓
Platform	Platform Sealing Nonce	PSN	🔓
	Platform Sealing Public Key	PSKU	🔓
	Platform Sealing Private Key	PSKR	🔁
	Platform Sealing Key Pair	PSKP	🔁
	Final Shared Key	FSK	🔁
	Final Initialization Vector	FIV	🔁
	Encrypted Base Key	EBK	🔓

🔒 Stored encrypted in s_x 🔓 Stored as cleartext in s_x 🔁 Derived

The amount of keying material is: one per group for entries under *Group*; and one per platform in the group for entries under *Platform*.

The SRX sealed data, s_x, is decrypted using a Common Sealing Key (CSK) and a Common Initialization Vector (CIV) that all platforms in group G_i can derive. CSK and CIV are derived from a Base Key (BK) and a Sealing Nonce (SN). BK is randomly generated, once, when the data is sealed for the first time, and SN is randomly generated each time the data is sealed to prevent key wear-out.

BK and SN are stored in s_x as AD. However, BK is not stored in cleartext but encrypted by a Final Shared Key (FSK) and a Final Initialization Vector (FIV), and the result of this encryption is called the Encrypted Base Key (EBK). There is one EBK for each platform in G_i but only one SN, which is stored in cleartext, shared by all platforms.

FSK and FIV are derived using a KDF that receives as input an Encryption Nonce (EN) and a shared secret. This shared secret is the result of ECDH between the Platform Sealing Private Key (PSKR) and the Group Sealing Public Key (GSKU). Each platform derives its own PSKR from a Platform Sealing Nonce (PSN) using the method detailed in Section III-B; and the group, as a whole, has a Group Sealing Key Pair where the private key (GSKR) is stored encrypted in s_x and the corresponding public key (GSKU) is stored in cleartext in s_x as AD.

1) Sealing algorithm

Algorithm 1 depicts the procedure for sealing the SRX data d_x. The `seal` function receives as input d_x (1). Then, it generates a random SN and updates the SN field in d_x with the new SN value (2). CSK and CIV are derived from BK and the new SN (3), and used to encrypt the updated

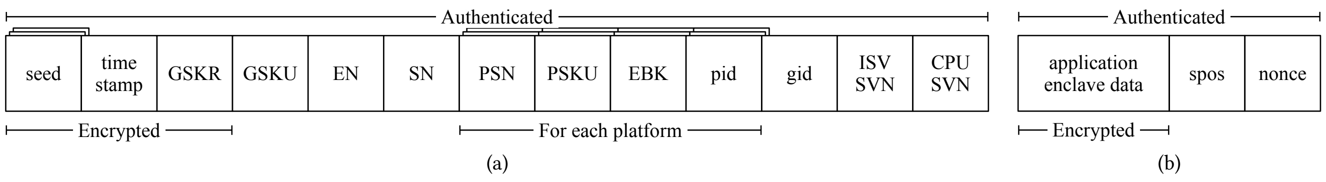

FIGURE 4. Contents of (a) s_x and (b) s_a. The acronyms are listed in Table 2 and Table 3.

d_x (4). Finally, the function returns the SRX sealed data, s_x (5).

BK is fixed, but SN is randomly generated on every re-seal to prevent key wear-out. For the sealing algorithm we assume BK is available, since this key must be decrypted during unsealing.

```
Algorithm 1 Sealing
Require: BK
1: function SEAL($d_x$)
2:     Generate random **SN** and update $d_x$ with new SN
3:     Derive **CSK** and **CIV** from BK and SN
4:     Encrypt $d_x'$ using CSK and CIV
5:     return $s_x$
6: end function
```

† Keying material in bold is derived or generated in that step.

2) Unsealing algorithm

Algorithm 2 depicts the unsealing procedure for the SRX sealed data, s_x. The `unseal` function receives as input s_x (1). Then, it derives PSKP from PSN (2); PSKP contains both PSKR and PSKU. SRX computes a shared secret, using ECDH, from PSKR and GSKU (3), and derives FSK and FIV from the shared secret and EN (4). BK is decrypted from EBK using FSK and FIV, and the AEAD scheme (5). CSK and CIV are derived from BK and SN (6), and used to decrypt s_x (7). Finally, the function returns the SRX data, d_x (8).

BK is kept in enclave memory for later use during sealing. For the unsealing algorithm we assume GSKU, EN, SN, PSN and EBK are stored in s_x as AD and therefore available to the unsealing procedure.

```
Algorithm 2 Unsealing
Require: PSN, GSKU, EN, EBK, SN
1: function UNSEAL($s_x$)
2:     Derive **PSKP** from PSN
3:     Compute **shared secret** from PSKP and GSKU
4:     Derive **FSK** and **FIV** from shared secret and EN
5:     Decrypt **BK** from EBK using FSK and FIV
6:     Derive **CSK** and **CIV** from BK and SN
7:     Decrypt $s_x$ using CSK and CIV
8:     return $d_x$
9: end function
```

† Keying material in bold is derived or generated in that step.

B. DETERMINISTIC KEY GENERATION IN SGX

The SGX SDK does not offer a mechanism to derive deterministic keys, for the enclaves of an ISV, out of the box. SRX requires this capability to generate the application enclave's Platform Sealing Private Key, PSKR, used in the algorithms of the previous section to derive the Common Sealing Key, CSK, that decrypts s_x. We constructed one such mechanism using the SGX instruction `EGETKEY`. The original use of the `EGETKEY` instruction, executed only inside an enclave, is to derive the seal, report, launch, and provision secret keys [2], [28], and return a 128-bit secret key from the processor-specific key hierarchy. In addition to being bound to the platform, this secret key is also bound to either the enclave measurement or the enclave signer.

`EGETKEY` receives as input a `KEYREQUEST` data structure containing parameters for the key derivation. These parameters include the `KEYNAME`, `KEYPOLICY`, `KEYID`, `ISVSVN`, and `CPUSVN`. `KEYNAME` identifies the type of key, `KEYPOLICY` identifies the derivation policy namely whether the sealing key is bound to the Enclave Identity (`MRENCLAVE`) or the Sealing Identity (`MRSIGNER`), `KEYID` is a value for key wear-out protection [8], [26], and `ISVSVN` and `CPUSVN` are the security version numbers of enclave and platform respectively. Invoking `EGETKEY` with the same `KEYREQUEST` structure produces the same output for the same enclave–platform pair.

In our case all parameters in `KEYREQUEST` are defined as constant with the exception of `KEYID`, which contains a nonce per enclave. The nonce can be safely stored in persistent storage in cleartext. The secret key output by `EGETKEY` is different for each enclave in each platform but is deterministic. This secret key is used as input keying material to a KDF to (i) prevent key wear-out and (ii) produce different keys. The output of the KDF has several potential uses namely secret key, private key, and ID. This approach can be stateless, for example, by hard coding the `KEYID` and using the output of `EGETKEY` as enclave–platform pair identification.

a: TCB updates

In order to enable enclave and platform updates, while maintaining the ability to derive older keying material, providing a `KEYREQUEST` data structure to `EGETKEY` with previous `ISVSVN` or `CPUSVN` values enables retrieving secret keys corresponding to those security version numbers.

In SRX the security version numbers used to derive the keying material granting access to s_x are stored as additional data in s_x, enabling access even after a TCB update. After a security update the application enclave can call `update` to reseal s_x with the up-to-date security version numbers as well as update seed and group keying material.

The shared secret keys granting access to s_a (§ III-F) are based on the seed stored in the encrypted part of s_x. The new seed is added to a set of seeds in s_x and is indexed by spos. Previous seeds are and not discarded to maintain access to previous application enclave sealed data.

C. INITIALIZATION PROTOCOL

The initialization protocol creates group G_i by initializing the internal state of SRX, adding platform p executing SRX to the group after it is attested by IAS, and saving the SRX sealed state s_x to persistent storage. When the initialization protocol completes successfully we have $p \in G_i$.

The application enclave invokes the `setup` ecall only once. This protocol involves SRX, RAP, IAS, and TUI.

a: Protocol

The application enclave ① calls the function `setup` of the SRX Trusted API (see Table 1). SRX ② generates the keying material, including PSN and PSKP, for the platform. SRX ③④⑤ computes a hash over PSN and PSKU and invokes `attest` to store it as user data in the quote sent to IAS, via RAP, during attestation of the enclave (see § III-G). SRX ⑨ verifies the signature of IAS over the report using the hard-coded Attestation Report Root CA Certificate of Intel [16], generates the group keying material (*seed*, BK, CSK, CIV, GSKU, GSKR, EN, SN, gid), requests a timestamp via `time`, and updates its internal state. Finally, SRX ⑨ seals its data (d_x) (see § III-A) and ⑩ returns the SRX sealed data (s_x) to the application enclave for writing to persistent storage.

D. ADDING A PLATFORM TO THE GROUP

Adding a new platform q to group G_i is a two-step process. Any platform $p \in G_i$ can add new group members. First q initializes its own state and attests itself with IAS, which produces a signed report containing a hash over the keying material of q. Then p adds q to G_i but only after user approval via TUI. The process completes after the internal state of SRX is updated, and d_x is sealed and the resulting s_x returned.

Initializing the platform involves SRX, RAP, and IAS. The new platform q does not need access to s_x, d_x or application enclave data during this first step.

Adding the platform involves SRX and TUI. The new platform q is used only during the platform initialization protocol, and is not involved in this second step. This means that a new platform can be added to the group without ever seeing the data it will be granted access to.

a: Protocol in q – initialize new platform

The application enclave ① calls `init_p`. SRX ② generates the keying material for the new platform, which includes PSN and PSKP, and ③④⑤ invokes the remote attestation protocol by calling `attest` with a hash over the platform keying material as argument receiving a report certified by IAS in return. Finally, SRX ⑨ serializes the platform keying material and the report produced by IAS, and ⑩ returns the result, d_p, to the caller along with the status of the operation.

b: Protocol in p – add new platform

The application enclave ① calls `add_p` with the details, d_p, of the new platform as argument. SRX ② unseals s_x obtaining d_x. Then, SRX verifies the signature of IAS over the report using the hard-coded Attestation Report Root CA Certificate of Intel and verifies that the hash stored in the report as user data matches the received keying material of the new platform, and ⑥⑦⑧ with user approval ⑨ updates d_x by adding the details of the new platform and refreshing the timestamp. Finally, SRX ⑨ seals $d_x^{\prime}$ (see § III-A) and ⑩ returns $s_x^{\prime}$ to the caller.

E. REMOVING A PLATFORM FROM THE GROUP

Removing a group platform requires deleting its details from d_x and generating a new seed. Having a new seed prevents the removed group member from using a previous s_x, from when it was still in the group, to access future application enclave data sealed with the up-to-date $s_x^{\prime}$. Any platform in the group can remove other group members, but not itself.

This protocol involves SRX and TUI. The platform being removed is not needed during this step.

a: Protocol

The application enclave ① calls `remove_p`. The request includes the pid of the platform to remove, which can be discovered using `list`. SRX ② unseals s_x obtaining d_x. Then, SRX ⑥⑦⑧ gets user authorization via `auth`, and ⑨ removes the platform details (PSN, PSKU, and EBK) and updates the seed and the timestamp in d_x. Finally, SRX ⑨ seals $d_x^{\prime}$ (see § III-A) and ⑩ returns $s_x^{\prime}$ to the application enclave to be stored in persistent storage.

F. SHARED SECRET KEYS GENERATION AND USE

Platforms in group G_i can retrieve secret keys shared by group members. These secret keys are based on a *seed* which is randomly generated during the initialization protocol (§ III-C), or by invoking `update`, and stored encrypted in s_x. The `get_sk` ecall receives a *nonce*, a *seed position*, and a *policy* as input. The nonce enables the generation of different secret keys based on the same seed via a KDF used internally by `get_sk`. This KDF uses the seed as input keying material and the nonce as salt. The seed position, *spos*, identifies which seed to use. The policy decides whether a user-authorization request via `auth` is needed before providing the secret key to the caller.

a: Protocol

The application enclave ① calls `get_sk`. SRX ② unseals s_x obtaining d_x. There are now two options depending on the policy, p, received as input. If the policy is 0, then SRX ⑨ derives and ⑩ returns the secret key to the application enclave immediately without requesting user authorization. If the policy is 1, then SRX ⑥⑦⑧ requests user authorization via TUI to satisfy the request, and only after such authorization is granted does it ⑨ derive and ⑩ return the secret key to the application enclave.

G. *REMOTE ATTESTATION*

We explain how remote attestation works in SGX, and how it applies to SRX.

a: In SGX

Remote attestation is part of SGX and ensures enclaves are legitimate and executing on authentic and up-to-date hardware. During remote attestation the enclaved application communicates with IAS via a service we call RAP in this paper. Enclaved applications do not communicate directly with IAS because IAS processes requests only from clients in possession of a *subscription key* that is licensed by Intel to service providers [16].

This subscription key must be kept confidential and can be rotated on demand by the ISV using an Intel-provided API. Having the subscription key stored in clients would greatly increase the possibility of compromise of this secret key, and would be a deployment problem when the key is rotated. For these reasons, providers do not store the subscription key in clients but in a remote service (RAP in our case), possibly in a hardware security module. Since this remote service mediates data exchanges between enclaved applications and IAS, it has access to all quotes and reports it forwards. This allows it to inspect IAS-certified reports during remote attestation, and refuse service to clients that have failed the attestation.

Authentication between RAP and IAS proceeds as follows. IAS authenticates itself to RAP using a certificate issued by a trusted certificate authority, and RAP authenticates itself to IAS by including in each HTTP request header the subscription key obtained by the ISV during registration with Intel [16]. The subscription key is protected during requests due to the use of TLS.

b: In SRX

RAP is a pure proxy in SRX forwarding quotes from applications to IAS and reports from IAS to applications. SRX requires no other service from RAP although the ISV could use RAP to provide additional functionality to enclaved applications. Remote attestation is used by the system initialization protocol (§ III-C) and by the platform initialization protocol (§ III-D) to ensure platforms being added to the group are legitimate.

In both cases, the enclaved application sends a quote, during attestation, to IAS to be validated and signed. The quote contains a user data field where SRX stores a hash computed over the nonce and public key of the new platform. The signed quote, which we call a report, and the nonce and public key of the new platform are returned from IAS to the application which can verify the authenticity of the data using the hard-coded public key of IAS.

SRX triggers the remote attestation process using the ecall `attest` of its Trusted API. However, this ecall is implemented by the ISV, and not by SRX. SRX cannot be attested on its own since it is part of the application enclave as a trusted library and is therefore implicitly attested when the application enclave is attested. The details of the communication protocol between the application enclave and RAP are not part of this work since this protocol is the responsibility of the ISV.

Note that the SRX functionality remains secure even when RAP is compromised since it is IAS that attests enclaves and platforms and certifies reports, not RAP. And these certified reports are validated, and the IAS signature over such reports verified, by enclaved applications. A compromised RAP can deny service to enclaved applications but can neither change group membership (§ III-D, § III-E) nor retrieve secret keys (§ III-F) to access application enclave sealed data.

IV. IMPLEMENTATION

This section describes the implementation of SRX and two applications that use it.

A. *SRX AND RAP*

The implementation follows the architecture in Fig. 2. SRX and RAP are implemented in C. The cryptographic functionality relies on OpenSSL and SGX SSL, when the functions provided by the SGX SDK are insufficient, and the communication between entities is based on ASN.1 using the *asn1c* [30] compiler adapted for SGX.

B. *EXAMPLE APPLICATION – BITCOIN WALLET*

The bitcoin wallet is based on two libraries: *libbtc* [31] for handling bitcoin data structures and exchanging data with the bitcoin network; and *libsecp256k1* [32] for handling ECDSA signatures. *libsecp256k1* is used internally by *libbtc*. The libraries were adapted to work on SGX and SRX, and are used by the two versions of the bitcoin wallet: the first version uses SGX (Wallet SGX) and the second version uses SGX plus SRX (Wallet SRX).

Wallet SGX sealed data is accessible only on the platform that sealed the data. Wallet SRX sealed data, however, is accessible to all platforms in the group of the enclave-platform pair that sealed the data.

TABLE 4. Lines of code of SRX and applications

Module	LoC	Change
SRX API	56	
SRX Implementation	4956	
Wallet SGX	1070	
Wallet SRX	1516	+41.7%
Titan Raw	2067	
Titan SGX	2117	+2.4%
Titan SRX	2464	+16.4%

C. EXAMPLE APPLICATION – PASSWORD MANAGER

We adapted the text-based *Titan* [33] password manager, as well as *SQLite* [34], to work on SGX and SRX. We refer to the original, untrusted, Titan as Titan Raw, and to the two versions we implemented as Titan SGX and Titan SRX. All versions use the SQLite library, internally, to store data. In Titan Raw the application decrypts and encrypts the database when explicitly requested by the user. In the period between decryption and encryption, the database is in an open state and accessible. In Titan SGX and Titan SRX, encryption is handled on the fly: the database is accessible only from within the enclave and only the requested password is handled in an open state outside the enclave.

Similarly to the bitcoin wallet example application, Titan SGX sealed data is accessible only to the platform that sealed it whereas Titan SRX sealed data is accessible to all platforms of the group owning that sealed data.

D. LINES OF CODE

Table 4 shows how many lines of code (LoC) are required for SRX and applications. For SRX the table shows LoC for the SRX API (i.e., the public interface) and for its implementation, excluding RAP and TUI since these are provided by the ISV and excluding the ASN.1 library we adapted for SGX. For the bitcoin wallet the table shows LoC for Wallet SGX and Wallet SRX excluding code for libbtc. For the password manager the table shows LoC for Titan Raw, Titan SGX and Titan SRX excluding code for SQLite.

V. ANALYTICAL EVALUATION

This section looks at the overhead of the library in terms of ecalls and ocalls, space requirements and scalability, and sealing and unsealing cost.

A. ENCLAVE CALLS AND OUTSIDE CALLS

In applications that use SGX there is a performance penalty when crossing the boundary between trusted and untrusted code [35], [36]. When the application enclave calls functions of the Trusted API all SRX functions are called from inside the trusted code and most of them are executed completely there. With respect to boundary crossing penalties, the corresponding overhead is only related to the regular interaction of the application with SGX.

Some SRX functions (`setup`, `init_p`, `add_p`, `remove_p`, `update`, and, depending on policy, `get_sk`) use `auth`, `attest` or `time` internally, as described in Section III. The execution of these functions requires boundary crossings. Nonetheless the main overhead from these functions comes from the ISV developed code and not from the boundary crossing. In addition, libraries used internally by the SRX implementation may cause boundary crossings during their invocation. For example, the SGX SSL library invokes the `sgx_cpuid` ocall during its setup routine which we confirmed using *sgx-perf* [37].

B. SPACE REQUIREMENTS AND SCALABILITY

The data storage requirements of SRX grow linearly with the number of platforms in a group G_i, i.e., is $O(n)$. Each platform in G_i requires space to store: *pid*, PSN, PSKU, and EBK with the respective tag. Additionally, there is a fixed cost for the keying material and metadata common to all group members: *seed*, GSKU, GSKR, EN, SN, *gid*, `CPUSVN` and `ISVSVN`. The exact space needed depends on the length of the cryptographic material but should be in the order of hundreds of bytes per platform. This data is stored in s_x which is shared by all platforms.

In terms of scalability all functions in Table 1 have a constant cost, $O(1)$, with the exception of `list` and `update`, which are $\Theta(n)$ where n is the number of members of the group.

C. DATA SEALING AND UNSEALING COST

This section evaluates the cost of sealing and unsealing data. There are three cases to consider:

1) The standard SGX sealing process that binds the sealed data to the processor and to either the current enclave measurement (`MRENCLAVE`) or the same enclave author (`MRSIGNER`);
2) The Intel Protected File System Library that encrypts data written to untrusted storage, and decrypts data read from untrusted storage, transparently using a subset of the C file API; and
3) The SRX sealing process.

Data encryption and data decryption is performed inside the enclave, in all these cases, using functions provided by the trusted cryptography library of the SGX SDK [26]. These functions are `sgx_rijndael128GCM_encrypt` and `sgx_rijndael128GCM_decrypt`.

The secret key for encryption and decryption, called sealing key, is obtained in different ways but all three cases rely on the `sgx_get_key` trusted function which is a wrapper for the `EGETKEY` instruction of SGX. Section III-B provides details on `EGETKEY`.

In SGX sealing the caller instantiates a key request structure, `sgx_key_request_t`, and invokes `sgx_get_key` using the key request structure as input to obtain the secret sealing key, `sgx_key_128bit_t`.

The Protected File System Library either uses a 128-bit secret key provided by the caller or the automatic keys functionality where the secret key is obtained using the same process of the standard SGX sealing and bound to the enclave author (`MRSIGNER`).

In SRX the caller first derives CSK to decrypt s_x (§ III-A) and then invokes `get_sk` to obtain a shared secret key (§ III-F). SRX also uses a key request structure and `sgx_get_key` during the decryption of s_x (§ III-B).

The cost of sealing and unsealing data can be divided in two parts: data encryption and decryption; and acquiring the sealing key. The cost of encrypting and decrypting the application enclave data depends on the size of the data, and is the same in all cases because they use the same encryption and decryption functions. The cost of obtaining the sealing key is constant for both SGX sealing and SRX sealing, and for the Protected File System Library when using the automatic keys functionality. Acquiring the sealing key in SRX sealing involves a longer process but the cost is still $O(1)$.

VI. EXPERIMENTAL EVALUATION

The performance impact is evaluated with the support of the two applications described in Section IV. The focus of the experiment is to quantify the overhead of the SRX library implementation. This is done by comparing the performance of the applications with and without integration with SRX.

Results do not include end user time to authorize operations via TUI because such operations are dependent on the user and also on the TUI implementation. TUI authorizations are filled in automatically via *xdotool* [38]. RAP is not used during the experimental evaluation since it is not necessary for the operations *derive* and *sign* of the bitcoin wallet, and *add* and *get* of the password manager. In addition, RAP latency and processing times would be dependent on the implementation which is provided by the ISV.

A. EXPERIMENTAL SETUP

The experimental evaluation has been carried out on a computer with an i5-7600 Intel processor running Ubuntu 18.04 LTS, and with an SSD. The Intel SGX software stack uses Intel SGX SDK 2.7.1, Intel SGX PSW 2.7.1, Intel SGX driver 2.6.0, and Intel SGX SSL 2.5. Untrusted cryptographic operations use OpenSSL 1.1.1a.

Experiments were executed on a shielded core, set to performance mode, and with CPU frequency scaling and turbo boost disabled. The applications' database and sealed files were stored on a *tmpfs* [39] memory-based file system. Each experiment was repeated 200 times, and each repetition is composed of a warm-up loop of 100 iterations followed by a collection loop of 500 iterations. Values are collected in nanoseconds using `clock_gettime`, during the collection phase, but presented in milliseconds.

B. EVALUATION OF SRX WITH THE BITCOIN WALLET

This experiment compares the SGX and the SRX versions of the bitcoin wallet for two operations: *derive* and *sign*. The operation *derive* returns an address for receiving funds, based on a derivation path [40], [41]. A derivation path defines a wallet structure that maps strings such as `m/44'/1'/0'/0/0` to accounts and addresses, based on a secret seed. The operation *sign* returns a signed transaction, based on the given inputs including the derivation path of the private signing key (which is never exposed outside the enclave). The secret seed is stored in application enclave sealed data s_a which is loaded into the enclave and unsealed, after s_x, on every *derive* and *sign* operation.

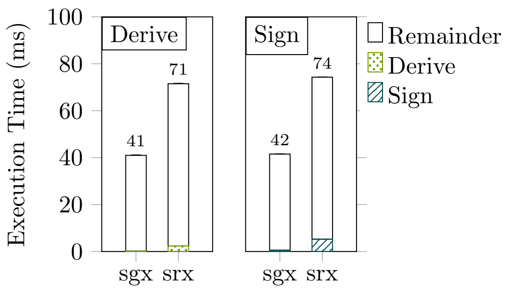

FIGURE 5. Total execution time of operations *derive* and *sign* of the bitcoin wallet. The remainder block includes the enclave life cycle.

Fig. 5 shows the total execution time of the bitcoin wallets. The execution time for the derive and sign functions is separate from the application life cycle, which includes creating and destroying the enclave as well as loading data from persistent storage into the enclave and unsealing that data. The overhead of SRX for the application life cycle is 27.8 ms for *derive* and 27.6 ms for *sign* and corresponds to the difference in total execution time minus the difference in execution time of these operations.

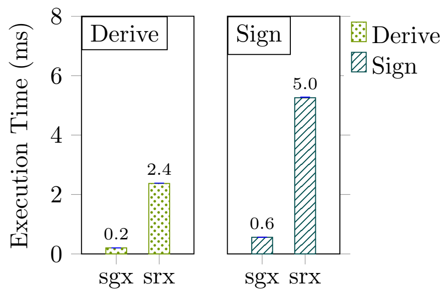

FIGURE 6. Execution time of functions *derive* and *sign* of the bitcoin wallet.

Fig. 6 shows only the derive and sign functions. These are a small part of the total execution time and are not easily seen in the previous figure. The implementation of derive and sign is similar for both SGX and SRX, but in SRX it includes an additional step to request user authorization before proceeding with the operation. User authorization is requested once for derive and twice for sign: before deriving a receiving address for the derive function; and before deriving the signing address and again before signing the transaction for the sign function. This is unnoticeable in an operation involving end-user input, as is the case with user authorization which was automated in these experiments, since such step is likely to take in the orders of seconds.

C. EVALUATION OF SRX WITH THE PASSWORD MANAGER

This experiment compares the raw, the SGX and the SRX versions of the password manager for two operations: *add* and *get*. The operation *add* inserts a new entry into the database. The operation *get* retrieves an existing entry from the database. Titan Raw handles all data processing in non-SGX space, and Titan SGX and Titan SRX decrypt the SQLite database when it is loaded into the enclave, then handle data processing inside the enclave, and finally encrypt the SQLite database when it is saved outside the enclave using the Intel Protected File System library. There are two versions of Titan SRX, one with secret key caching and the other without. In Titan SRX, the secret key that seals data is computed by invoking the function get_sk of SRX. The first version of Titan SRX, *srx+o* in the figures, computes the secret key once and then caches it (inside the enclave), whereas the second version of Titan SRX, *srx-o* in the figures, computes the secret key on every request. The *+o* or *-o* stands for with or without optimization.

The insertion of a new entry into the database asks for a description, a username, a website, a notes field, and a password. The fields are not mandatory. When the password is not specified then one is generated by the application; for the SGX-based versions of the password manager this is done inside the enclave. In this experiment the input is received from a file that contains random strings generated using *pwgen* [42].

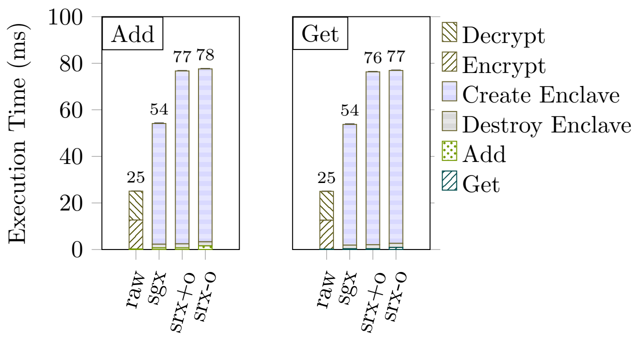

FIGURE 7. Total execution time of operations *add* and *get* of the password manager.

Fig. 7 shows the total execution time of the password manager. That time is composed of database encryption and decryption for Titan Raw, enclave life cycle for Titan SGX and Titan SRX, and either the add or get function. In Titan Raw the database is decrypted or encrypted on-demand by the user whereas in Titan SGX and Titan SRX the database is decrypted and encrypted on-the-fly when transferred into, or out of, the enclave. Database encryption and decryption, and in particular the enclave life cycle, consume most of the execution time. The overhead of SRX for the application life cycle is 23.1 ms for *add* and 22.4 ms for *get* and corresponds to the difference in total execution time minus the difference in execution time of these operations.

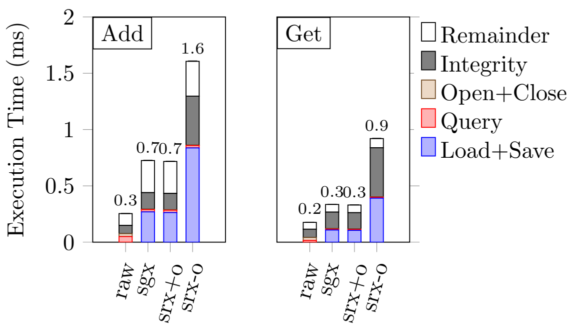

FIGURE 8. Execution time of functions *add* and *get* of the password manager.

Fig. 8 shows the breakdown of the add and get functions. *Integrity* is a database verification step applied during each operation, *Open+Close* refers to opening and closing the SQLite database, *Query* is the execution of the statement that adds data to, or retrieves data from, the database, and *Load+Save* is an SGX-only step where the data is loaded into the enclave and unsealed and vice versa. The implementation of Load+Save is similar for both Titan SGX and Titan SRX and is based on the Intel Protected File System library. However, Titan SGX uses the automatic keys functionality provided by the library, where secret keys are derived from the enclave sealing key, while Titan SRX retrieves the secret key using the function get_sk of SRX. The overhead of SRX for Load+Save is -2% for both add and get when using the srx+o implementation, and 212% for add and 263% for get when using the srx-o implementation. In the first case, the secret key is computed once during the warm up phase and cached, and in the second case the secret key is computed on every iteration of the measurement loop. The integrity check function opens, internally, the database for reading and then closes it. This explains why the integrity value is higher in srx-o when compared to srx+o. The execution time for Load+Save is greater for the add function than for the get function, because during the get operation the database is opened for read-only and closed without flushing it to disk whereas during the add operation the database is opened for read-write and must be flushed to disk after the add operation in order to save the entry insertion.

D. ENCLAVE CREATION AND DESTRUCTION TIME

This section evaluates the effect of the size of the enclave on its creation time, which helps understanding the 20 to 30 ms overall overhead measured in the previous experiments.

This experiment uses the sample enclave from Intel SGX SDK to find how long it takes to create and destroy an enclave based on the size of the corresponding shared library. We do not invoke the API of the sample enclave itself and only measure the times it takes enclave creation and destruction functions to execute. Function `sgx_create_enclave` loads and initializes the enclave, and function `sgx_destroy_enclave` destroys the enclave and releases the previously allocated resources.

For each data point, i.e. enclave shared library size, the experiment consists of a 100-iterations warm-up loop followed by a 500-iterations collection loop. The enclave size is increased by allocating a large array inside the enclave. The enclave configuration file is set to use a single thread, as is done in the previous experiments.

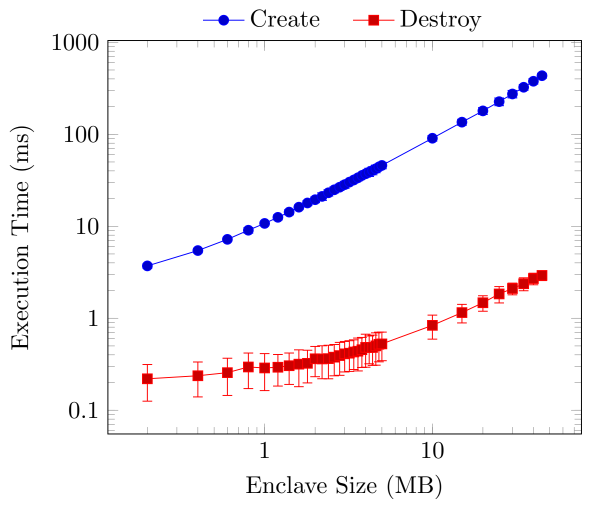

FIGURE 9. Execution time for creating and destroying an enclave.

Fig. 9 shows the time, in milliseconds, that it takes to create the sample enclave and to destroy the sample enclave. This is done for different enclave sizes ranging from 200 KB to 5000 KB, in 200 KB increments, and from 5 MB to 45 MB, in 5 MB increments. In both enclave creation and enclave destruction the execution time increases with the size of the enclave shared library, called `enclave.signed.so` in the case of the sample enclave, but the increase in the destruction time is small when compared with the increase in creation time.

All plots have error bars showing the standard deviation. However, the variation between values is small and therefore the error bars cannot be seen.

E. RECOVERING APPLICATION ENCLAVE SEALED DATA

The SRX API does not have explicit backup and recovery functions: a *backup* is created by copying s_x and s_a onto another medium as is done with regular files; and *recovery* is performed by decrypting s_a using the secret key obtained from s_x. The reason is that any enclave–platform pair from a group accesses SRX sealed data s_x and application enclave sealed data s_a of that group in the same way, independently of being the creator of the sealed data or not. Therefore, the recovery process time is the same as the time it takes for a group member to access s_a, assuming that member is already in the group.

The recovery process, after the enclave is created and SRX is initialized (§ III-C), is the following:

1) SRX unseals s_x obtaining d_x (§ III-A2),
2) the application enclave invokes `get_sk` to retrieve the secret key (§ III-F), and
3) the application enclave unseals s_a using the secret key.

From previous experiments we know this process takes less than 1 ms, excluding the enclave life cycle which is less than 80 ms for both Wallet SRX (Fig. 5) and Titan SRX (Fig. 7). The *Load+Save* for function *get* in Fig. 8 first loads s_x into the enclave and decrypts it obtaining d_x (step 1), then retrieves the secret key to decrypt s_a from the Trusted API of SRX (step 2), finally decrypts s_a (step 3). This entire process requires less than 1 ms, which in this case is the difference between Titan SGX (0.3 ms) and the *srx-o* version of Titan SRX (0.9 ms).

This assumes the invocation of `get_sk` with a policy of `0` which means user authorization via TUI is not required; with a policy of 1 the recovery time becomes the previous time plus the end-user authorization time. The end-user authorization time is variable depending on the TUI solution implemented by ISV developers and the skill of the end user in manipulating the equipment, but should be in the order of seconds and thus dominates the total recovery time.

VII. RELATED WORK

Multiple applications [43]–[50] use enclaves to strengthen their security and new tools and frameworks [22], [51]–[58] have been created to support developers using SGX. There is no related work that solves our specific problem of securely sharing data among platforms and enclaves from the same user, e.g., for backup purposes. The remainder of this section presents work on the migration of SGX-enabled virtual machines, key escrow and recovery, and secure group communication that has some relation to SRX.

A. SGX MIGRATION

Park et al. [59] identify a key challenge of migrating SGX-enabled virtual machines: the enclave code and data resides in PRM and is therefore not accessible to the VMM. The authors propose a conceptual scheme, without offering an implementation, in which a new hardware instruction would be used by source and destination machines to derive migration keys.

Gu et al. [60] present a system that supports the live migration of virtual machines with enclaves. A source enclave remotely attests a target enclave located in a different machine and subsequently passes its execution context to this enclave followed by a migration key; the source enclave destroys itself after sending the migration key thus ensuring no more than one enclave instance is running this execution context. The migration key is obtained from the enclave owner. We focus on the migration and backup/recovery of sealed enclave state across machines, by proposing a secure mechanism to distributed group sealing keys, while the authors' focus on the live migration of the execution context of the enclave. They mention checkpoint/resume operations [60, § V-C], which would be more similar to our own work, but give few details on how this is supposed to work.

Alder et al. [61] propose a mechanism for migrating enclaves with persistent state by creating a migration enclave on source and destination machines. In this case the enclave data is migrated from source machine to destination machine becoming unavailable to the source machine. In our work the data is available to both source and destination, and SRX is applied only to the data and not to entire virtual machines. Alder et al. [61, § III-B] consider having the data available to both source and destination machines a fork attack.

Park et al. [62] propose adding new instructions to support the migration of enclaves running in virtual machines on different hosts. The hosts share a secret key, accessible only to the enclave in question and the processor, to support the migration. The prototype is implemented on top of the open-source SGX emulator OpenSGX [63]. Their work is focused on migrating the enclave memory and does not deal with the transfer of sealed data which is relegated to future work.

Liang et al. [64] design and implement a solution for checkpoint-based and storage-based migration of containers. The source and target hosts share monotonic counters and a migration key via a *Boot Enclave Service*, which is essentially a trusted authority that "must be deployed in a trusted machine."

B. KEY ESCROW AND RECOVERY AGENTS

Key escrow is an arrangement whereby keying material granting access to encrypted data is made available to authorized entities, by one or more trusted third parties, under prescribed conditions [65], [66]. Recovery agents, on the other hand, have themselves the ability to recover keying material or plaintext.

In SRX the entity recovering data is the *same entity* that encrypted (sealed) that data. This contrasts with key escrow and recovery agents where keying material, or data, is provided to other entities. In addition, the keying material in SRX is bound to the processors forming the enclave–platform group.

C. SECURE GROUP COMMUNICATION

Key management and distribution is a challenge in secure group communication [67] where members may join and leave the group at any point in time. Backward secrecy prevents new members from reading messages exchanged before entering the group and forward secrecy prevents former members from reading future messages. SRX prevents former group members from reading future sealed data (§ III-E) but does not prevent new members from reading past data since this is a requirement for having data recovery.

VIII. CONCLUSION

The standard sealing procedure of Intel SGX provides strong guarantees that sealed data is not accessible to platforms other than the one that sealed the data. This approach to sealing is sufficient when there is only one platform using the data, but becomes a limitation when other platforms require access to the same sealed data. This is a scenario which could occur when the platform that seals the data becomes unavailable due to, for example, damage or theft.

Towards solving this issue we propose SRX where platforms form groups and platforms within the same group can access each others' sealed data. This is achieved without leaking the secret keys that seal the data outside the enclave. We built a prototype of SRX to prove its feasibility and evaluate it using two applications. We show that the overhead of using SRX is in the order of tens of milliseconds.

.

APPENDIX A CORRECTNESS

This section presents an argument that SRX satisfies the security properties of Section II-E. Proofs use four ancillary functions: $auth_p(f)$ that user-authenticates function f on platform p; $exec_p(f)$ that executes function f in the enclave on platform p; and $seal_p(s_{x_i}, a_i)$ and $unseal_p(s_{x_i}, s_{a_i})$ that respectively seals data a_i in platform p and unseals data s_{a_i} in platform p.

Definition. The *public keying material* of a platform are the nonces, public keys, and encrypted data which (i) are available to all parties that have access to s_x (but not necessarily d_x), and (ii) are required for deriving some non-public keying material such as private keys of the platform.

Definition. A sealed data bundle, s, is composed of two main parts: an encrypted part C and a cleartext part A.

$$s = C||A$$

Lemma 1 (Authorization). *A function* $f \in F$, *where* $F = \{add\_p, remove\_p, get\_sk\}$, *is executed by platform*

$p \in G_i$ without returning an error only if the execution of f is requested by p and the execution of f is authorized by the user.

$$\forall f \in F, \quad \forall p \in G_i \quad (exec_p(f) \Rightarrow auth_p(f))$$

Proof. Any function $f \in F$ is executed only after (i) unsealing s_{x_i} and (ii) obtaining end user authorization. The application enclave can unseal s_{x_i} only when the platform is in group G_i (proved in Lemma 3). In addition, SRX assumes the existence of a trusted path between the application enclave and the user as stated in Section II-C. This trusted path is invoked by the code of functions in F to request user authorization. Therefore a platform $e \notin G_i$ cannot successfully invoke functions in F and a platform $p \in G_i$ only executes functions in F successfully when authorized by the end user. □

Lemma 2 (Shareability)**.** *Any platform $q \in G_i$ with access to the application enclave sealed data, s_{a_i}, and the SRX sealed data, s_{x_i}, is able to unseal data s_{a_i} sealed by platform $p \in G_i$.*

$$\forall p, q \in G_i \quad (a_i = unseal_q(s_{x_i}, seal_p(s_{x_i}, a_i)))$$

Proof. Given $s_{x_i} = C_{x_i}||A_{x_i}$ and $s_{a_i} = C_{a_i}||A_{a_i}$ we prove members of the group G_i have their public keying material in A_{x_i} which is used to decrypt C_{x_i} that in turn decrypts C_{a_i}. Decrypting C_{a_i} is equivalent to unsealing s_{a_i}.

1) *Any platform $p \in G_i$ has its public keying material, necessary to derive CSK and CIV, in A_{x_i}.* There are two ways for a platform to join group G_i: the first is during system initialization and the second is after the system is set up. During the initialization protocol, described in Section III-C, the platform p that initializes the system and creates the group becomes the first platform in the group ($G_i = \{p\}$). After the system is set up, a platform $p \in G_i$ can add a platform not yet in G_i to the group, as described in Section III-D, resulting in $G_i = \{p, q\}$. When either protocol concludes, the public keying material of the newly added platform is in A_{x_i}.
2) *Any platform with its public keying material in A_{x_i} can decrypt C_{x_i}.* A platform that has its public keying material in A_{x_i} can use the unsealing algorithm, detailed in Section III-A, to decrypt C_{x_i}.
3) *Any platform able to unseal s_{x_i} can unseal s_{a_i}. In other words, the ability to decrypt C_{x_i} implies the ability to decrypt C_{a_i}.* A platform able to unseal s_{x_i} can use the protocol detailed in Section III-F to derive the input keying material that unseals s_{a_i}.

□

Lemma 3 (Confidentiality)**.** *Any platform $q \notin G_i$ with access to the application enclave sealed data, s_{a_i}, and the SRX sealed data, s_{x_i}, is unable to unseal data s_{a_i} sealed by platform $p \in G_i$.*

$$\forall e \notin G_i, \quad \nexists p \in G_i \quad (unseal_e(s_{x_i}, seal_p(s_{x_i}, a_i)) = a_i)$$

Proof. Given $s_{x_i} = C_{x_i}||A_{x_i}$ and $s_{a_i} = C_{a_i}||A_{a_i}$ we prove platforms with their public keying material in A_{x_i} are members of the group G_i, platforms without their public keying material in A_{x_i} cannot decrypt C_{x_i} that is equivalent to unsealing s_{x_i}, and that platforms unable to unseal s_{x_i} are unable to unseal s_{a_i}. Therefore concluding that any platform $e \notin G_i$ is unable to unseal s_{a_i}.

1) *All platforms with their public keying material in A_{x_i} are in group G_i.* As mentioned in the proof of Lemma 3 (1), a platform joins a group G_i either when the group is created during the initialization protocol or at a later stage with the support of a platform $p \in G_i$. In both cases, data about the platform is added to s_{x_i}, so the presence of its public keying material in A_{x_i} implies this platform is part of group G_i.
2) *All platforms without their public keying material in A_{x_i} cannot decrypt C_{x_i}.* The unsealing algorithm, detailed in Section III-A, requires the platform to have its public keying material in A_{x_i} in order to decrypt C_{x_i}, and platforms without their public keying material in C_{x_i} are unable to decrypt A_{x_i}.
3) *All platforms unable to unseal s_{x_i} are unable to unseal s_{a_i}.* Unsealing s_{a_i} implies verifying the tag over s_{a_i} and decrypting C_{a_i} using input keying material retrieved according to the protocol in Section III-F. To retrieve this input keying material the platform must be able to decrypt C_{x_i}, that is, unseal s_{x_i}. The inability to decrypt C_{x_i} (unseal s_{x_i}) therefore implies the inability to decrypt C_{a_i} (unseal s_{a_i}).

Notice that this property is about a platform q that was never part of the group G_i. If at some point $q \in G_i$ and later q is removed from G_i resulting in $s_{x_{i+1}}$, then q maintains access to s_{a_i} using s_{x_i} since it already had access to this data in the past, but is unable to access $s_{a_{i+1}}$ sealed using the newer version of the group $s_{x_{i+1}}$. □

APPENDIX B KEYS' DERIVATION AND HIERARCHY

Fig. 10 depicts the derivation of keying material in SRX and the relationship between its public, private, and secret keys. Section III-A describes the sealing mechanism of SRX.

REFERENCES

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. 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)*, Jun. 2013.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Jun. 2013.

[3] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Support for dynamic memory management inside an enclave," in *Proceedings of the 5th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Jun. 2016.

[4] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Jun. 2013.

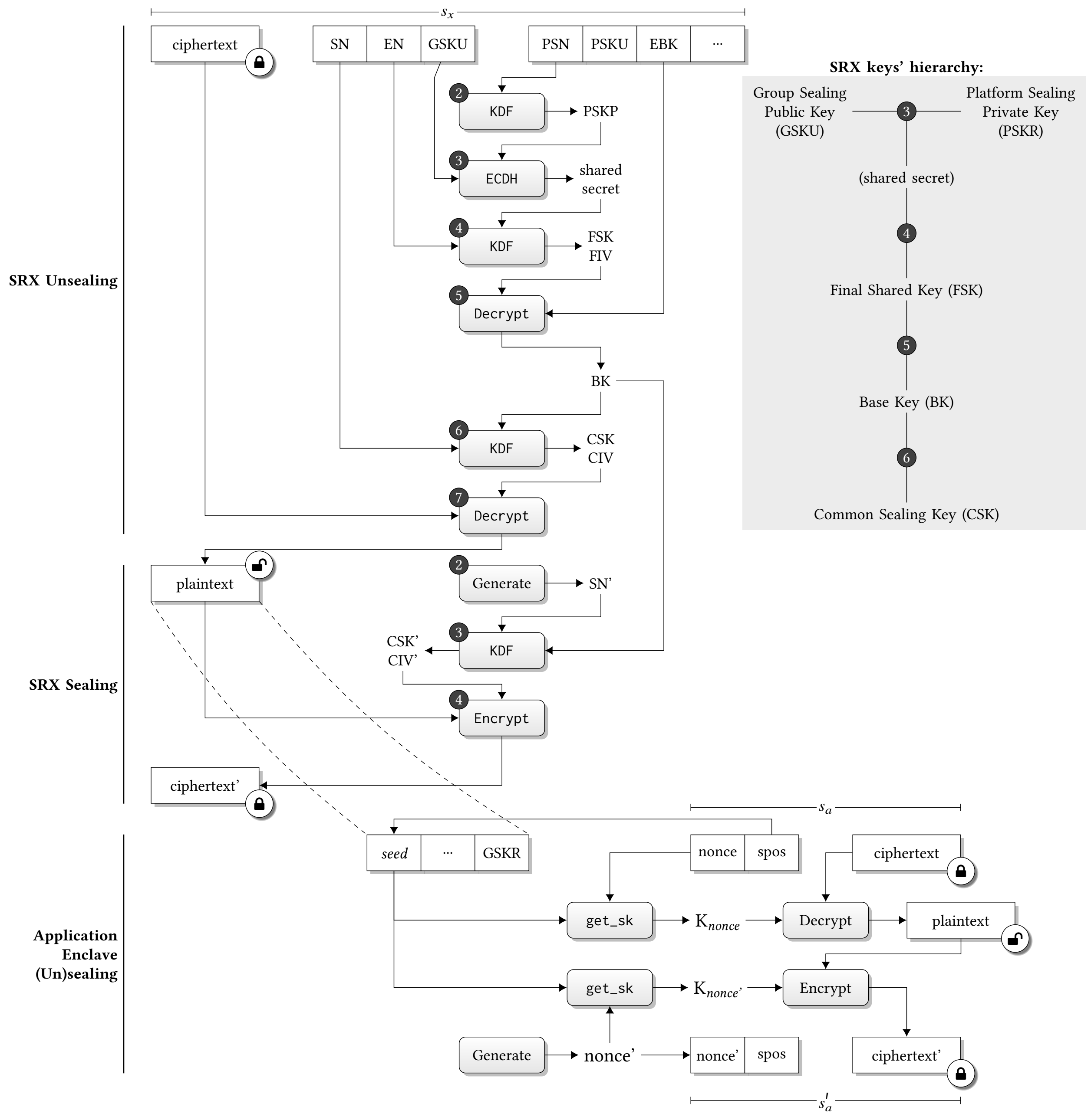

FIGURE 10. Derivation of keying material for unsealing, and sealing, s_x and s_a. Steps 2–7 in *SRX Unsealing* and 2–4 in *SRX Sealing* match line numbers in Algorithm 1 and Algorithm 2, respectively. The hierarchy between the SRX keys is shown on the top-right and the numbers match the steps in the function boxes.

[5] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, "Support for dynamic memory allocation inside an enclave," in *Proceedings of the 5th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Jun. 2016.

[6] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, Dec. 2016.

[7] G. Chen, Y. Zhang, and T.-H. Lai, "OPERA: Open remote attestation for intel's secure enclaves," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2019.

[8] *Intel Software Guard Extensions (Intel SGX): Developer Guide*, Intel Corporation, Nov. 2019, release v2.7.1 for Linux.

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

[10] 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 USENIX Security Symposium*, Aug. 2018.

[11] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, Aug. 2019.

[12] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Jul. 2020.

[13] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2002.

[14] D. Boneh and I. E. Shparlinski, "On the unpredictability of bits of the elliptic curve Diffie-Hellman scheme," in *Annual International Cryptology Conference*. Springer, Aug. 2001.

[15] D. A. McGrew, "An interface and algorithms for authenticated encryption," RFC Editor, RFC 5116, Jan. 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5116.txt

[16] *Attestation Service for Intel Software Guard Extensions: API Documentation*, Intel Corporation, 2020, revision 6.0.

[17] S. Weiser and M. Werner, "SGXIO: Generic trusted I/O path for Intel SGX," in *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Mar. 2017.

[18] H. Liang, M. Li, Y. Chen, L. Jiang, Z. Xie, and T. Yang, "Establishing trusted I/O paths for SGX client systems with Aurora," *IEEE Transactions on Information Forensics and Security*, vol. 15, 2020.

[19] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Jul. 2012.

[20] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of the 5th Asia-Pacific Workshop on Systems (APSys)*, Jun. 2014.

[21] M. Brandenburguer, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *Proceedings of the Annual IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jun. 2017.

[22] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *Proceedings of the 26th USENIX Security Symposium (SEC)*, Aug. 2017.

[23] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato, "Internet x.509 public key infrastructure time-stamp protocol (TSP)," RFC Editor, RFC 3161, Aug. 2001. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3161.txt

[24] H. Liang and M. Li, "Bring the missing jigsaw back: TrustedClock for SGX enclaves," in *Proceedings of the 11th European Workshop on Systems Security*, Apr. 2018.

[25] *Trusted Time and Monotonic Counters with Intel SGX Platform Services*, Intel Corporation, 2017.

[26] *Intel Software Guard Extensions (Intel SGX) SDK for Linux OS: Developer Reference*, Intel Corporation, Nov. 2019, release v2.7.1 for Linux.

[27] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, Nov. 1976.

[28] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Intel Corporation, Oct. 2019, 325384-071US.

[29] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography," National Institute of Standards and Technology, NIST Special Publication (SP) 800-56Ar3, Apr. 2018.

[30] L. Walkin, *asn1c*, ASN.1 compiler. [Online]. Available: https://github.com/vlm/asn1c

[31] *libbtc*, bitcoin library. [Online]. Available: https://libbtc.github.io/

[32] *libsecp256k1*, library for cryptographic operations on curve secp256k1. [Online]. Available: https://github.com/bitcoin-core/secp256k1

[33] N. Rosvall, *Titan*, password manager. [Online]. Available: https://github.com/nrosvall/titan

[34] *SQLite*, database library. [Online]. Available: https://sqlite.org/

[35] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2017.

[36] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, "Everything you should know about Intel SGX performance on virtualized systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 3, no. 1, Mar. 2019.

[37] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel SGX enclaves," in *Proceedings of the 19th ACM/IFIP International Middleware Conference*, Dec. 2018.

[38] J. Sissel, *xdotool(1)*, automation tool. [Online]. Available: https://manpages.ubuntu.com/manpages/bionic/en/man1/xdotool.1.html

[39] P. Snyder, "tmpfs: A virtual memory file system," in *Proceedings of the Autumn 1990 EUUG Conference*, Oct. 1990.

[40] P. Wuille, "Hierarchical deterministic wallets," BIP, BIP 32, 2012.

[41] M. Palatinus and P. Rusnak, "Multi-account hierarchy for deterministic wallets," BIP, BIP 44, 2014.

[42] *pwgen(1)*, Ubuntu 18.04 LTS. [Online]. Available: https://manpages.ubuntu.com/manpages/bionic/en/man1/pwgen.1.html

[43] 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 IEEE Symposium on Security and Privacy (S&P)*, Jul. 2015.

[44] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "PESOS: Policy enhanced secure object store," in *Proceedings of the 13th EuroSys Conference*, Apr. 2018.

[45] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using sgx," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, May 2018.

[46] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, V. Schiavoni, P.-L. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza, "EndBox: Scalable middlebox functions using client-side trusted execution," in *Proceedings of the Annual IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jul. 2018.

[47] S. Arnautov, A. Brito, P. Felber, C. Fetzer, F. Gregor, R. Krahn, W. Ozga, A. Martin, V. Schiavoni, F. Silva, M. Tenorio, and N. Thummel, "PubSub-SGX: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems," in *Proceedings of the 37th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2018.

[48] B. Fuhry, L. Hirschoff, S. Koesnadi, and F. Kerschbaum, "SeGShare: Secure group file sharing in the cloud using enclaves," in *Proceedings of the Annual IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jul. 2020.

[49] 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 *Proceedings of the Annual IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jul. 2020.

[50] J. Guerreiro, R. Moura, and J. N. Silva, "TEEnder: SGX enclave migration using HSMs," *Computers & Security*, vol. 96, Sep. 2020.

[51] 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)*, Oct. 2014.

[52] C.-C. Tsai, D. E. Porter, and M. Viji, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Jul. 2017.

[53] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[54] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB linux applications with SGX enclaves," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.

[55] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for Intel SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Jul. 2017.

[56] S. Wang, W. Wang, Q. Bao, P. Wang, X. Wang, and D. Wu, "Binary code retrofitting and hardening using SGX," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, Nov. 2017.

[57] T. Lazard, J. Götzfried, T. Müller, G. Santinelli, and V. Lefebvre, "TEEshift: Protecting code confidentiality by selectively shifting functions into TEEs," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX)*, Jan. 2018.

[58] V. A. Sartakov, S. Brenner, S. B. Mokhtar, S. Bouchenak, G. Thomas, and R. Kapitza, "EActors: Fast and flexible trusted computing using SGX," in *Proceedings of the 19th ACM/IFIP International Middleware Conference*, Dec. 2018.

[59] J. Park, S. Park, J. Oh, and J.-J. Won, "Toward live migration of SGX-enabled virtual machines," in *Proceedings of the 2016 IEEE World Congress on Services (SERVICES)*, Jun. 2016, Short Paper.

[60] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *Proceedings of the 47th IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jun. 2017.

[61] F. Alder, A. Kurnikov, A. Paverd, and N. Asokan, "Migrating SGX enclaves with persistent state," in *Proceedings of the 48th IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jun. 2018.

[62] J. Park, S. Park, B. B. Kang, and K. Kim, "eMotion: An SGX extension for migrating enclaves," *Computers & Security*, vol. 80, Sep. 2019.

[63] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An open platform for SGX research," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2016.
[64] H. Liang, Q. Zhang, M. Li, and J. Li, "Toward migration of SGX-enabled containers," in *IEEE Symposium on Computers and Communications (ISCC)*, Jun. 2019.
[65] D. E. Denning and D. K. Branstad, "A taxonomy for key escrow encryption systems," *Communications of the ACM*, vol. 39, no. 3, Mar. 1996.
[66] H. Abelson, R. Anderson, S. M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, P. G. Neumann, R. L. Rivest, J. I. Schiller, and B. Schneier, "The risks of key recovery, key escrow, and trusted third-party encryption," May 1997.
[67] S. Rafaeli and D. Hutchison, "A survey of key management for secure group communication," *ACM Computing Surveys (CSUR)*, vol. 35, no. 3, Sep. 2003.

DANIEL ANDRADE is currently working towards a PhD degree in Computer Science and Engineering at Instituto Superior Técnico, Portugal. He is a junior researcher at INESC-ID. His research interests include systems security, applications of Intel SGX, and systems programming.

JOÃO SILVA is an Assistant Professor at the Electrical Engineering Department at Instituto Superior Técnico, Lisbon University, Portugal. He has a PhD in Computer Science and has been doing research in distributed systems, including their security, for many years, with a focus on mobile computing.

MIGUEL CORREIA is a Full Professor at Instituto Superior Técnico (IST), Universidade de Lisboa, in Lisboa, Portugal. He is coordinator of the Doctoral Program in Information Security at IST. He is member of the board and senior researcher at INESC-ID. He has been involved in many international and national research projects related to cybersecurity, including the DE4A, BIG, QualiChain, SPARTA, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 200 publications and is Senior Member of the IEEE.