# SEEDS: Secure Extensible Enclave with Decentralized Storage for Secrets

Stefanie Dukovac
RCS Lab, University of Waterloo
smdukova@uwaterloo.ca

Emil Tsalapatis
RCS Lab, University of Waterloo
emil.tsalapatis@uwaterloo.ca

Ali José Mashtizadeh
RCS Lab, University of Waterloo
ali@rcs.uwaterloo.ca

## ABSTRACT

Applications that use passwords or cryptographic keys to authenticate users or perform cryptographic operations rely on centralized solutions. Hardware Trusted Platform Modules (TPMs) do not offer a way to replicate material, making access in a distributed environment difficult. Meanwhile, remote services require a constant network connection and are a central point of failure. Administrators access to such services means total and permanent compromise of the secrets.

We present SEEDS, an SGX based secure decentralized multi-user store for cryptographic secrets. SEEDS prevents secrets from leaking even against attackers with access to user credentials by providing an API to use keys without reading them. SEEDS tolerates long network partitions and uses CRDTs to reconcile state, making it a viable mechanism for local authentication.

SEEDS provides a base for distributed SGX capable applications through an expressive policy engine. It represents both all data and metadata like users as entries in a key value store, both of which are governed by policy. SEEDS also allows developers to add cryptographic protocols to existing enclaves by composing existing SEEDS operations.

We demonstrate how to use SEEDS as a secrets manager with two applications. We present a decentralized and highly available alternative to LDAP plus Kerberos. We also describe a software U2F token implementation.

**ACM Reference Format:**
Stefanie Dukovac, Emil Tsalapatis, and Ali José Mashtizadeh. 2022. SEEDS: Secure Extensible Enclave with Decentralized Storage for Secrets. In *Proceedings of Proceedings of the 5th Workshop on System Software for Trusted Execution (SysTEX '22 Workshop).* ACM, New York, NY, USA, 7 pages.

## 1 INTRODUCTION

Security applications ranging from authentication systems like LDAP and Kerberos to password managers all have the problem of securely storing and using cryptographic secrets across multiple hosts. These systems decrypt secrets in memory before using them, allowing attackers to exfiltrate them even if the secrets are encrypted at rest. Replicating these secrets across hosts for availability or caching further exposes them to attackers.

A possible solution is to store keys in a TPM or a TEE based system that allows cryptographic operations without exposing them to the host. TPMs [1] are hardware devices installed on a host that securely store secrets and provide an API for applications to perform cryptographic operations. TPMs are centralized and cannot share state across devices. cTPM [6] and CKS [12] provide similar services over the network to allow a user or service to perform cryptographic operations across machines.

| | TPM [1] | CKS [12] | cTPM [6] | SEEDS |
|---|---|---|---|---|
| Security | | | | |
| Multi-User Access Control | ○ | ● | ● | ● |
| Denial of Service | ○ | ○ | ○ | ● |
| Compromised User | - | ● | ○ | ● |
| Compromised Host | ● | ● | ◐ | ◐ |
| Functionality | | | | |
| Replication | ○ | ◐ | ◐ | ● |
| Disconnected Operations | - | ○ | ◐ | ● |
| Arbitrary Data | ○ | ○ | ○ | ● |

**Table 1: Comparison of TEE and TPM related secrets managers. Single node TEEs are accessible only locally. Distributed TEE systems do not ensure availablity and do not support multiple users. CKS and cTPM use a cloud machine that is a single point of failure.**

Table 1 compares these three systems. Both cloud based systems depend on network connectivity, limiting their use for offline applications. Both are similar to a local TPM in that they provide a fixed API of cryptographic protocols and primitives. Both do not provide perfect confidentiality for the keys. For CKS, an attacker with admin privileges can trivially extract the secrets by abusing the system's update mechanism. Clients also depend on constant connectivity to the cloud machine. cTPM fully trusts the remote cloud machine it uses to share keys between devices.

We present SEEDS, a TEE based decentralized secrets store that provides strong security, high availability and extensibility to support a broad set of security applications. SEEDS runs on each device and provides a replicated key-value store that exposes cryptographic primitives. Developers define policies that limit the possible damage from a compromised host or user. SEEDS provides an extensibility mechanism to allow developers to implement cryptographic protocols inside the TEE.

SEEDS is motivated by the observation that many security applications require the ability to store cryptographic secrets, replicate those secrets between hosts and perform cryptographic operations. These applications use relatively simple logic that does not require communications with other hosts. SEEDS provides a decentralized storage system that supports the application logic without exposing cryptographic secrets.

SEEDS provides control over fine-grained policy to allow applications to customize their security guarantees. Developers define a policy that restricts what cryptographic operations can be executed on a given key-value pair. The policy also controls modifications to the set of hosts and the policy itself. For example, private keys can be restricted to only signing when authorized by a specific principal (but never read), or the policy can be made immutable so that even a malicious admin cannot enable reading cryptographic keys.

While cryptographic primitives are common across applications, cryptographic protocols are not. SEEDS allows applications to add new API calls at runtime through small scripts interpreted inside the enclave. The new calls are subject to the policy, which prevents an attacker from adding malicious scripts that export secrets.

Our extensibility subsumes the need for certain upgrades to SEEDS. Enclave upgrades typically involve migrating data from an old to a new trusted instance. An attacker with admin privileges can then ‘upgrade’ to an instance that exposes all data to the untrusted host. Similarly, if reading out a key is truly impossible then that key is bound to the enclave instance. Replacing it means we lose the key.

We use *strong eventual consistency* to ensure high availability while guaranteeing that updates converge when devices come online. Conflict-Free Replicated Data Types (CRDTs) allow us to provide eventual consistency with well defined semantics. These semantics allows us to use simple application logic to avoid any security implications of reconciling state.

We developed two applications based on SEEDS: First, we replace LDAP and Kerberos with a distributed, secure, and highly available service built using NSS and PAM modules that communicate directly with SEEDS. Second, we use SEEDS to build a replicated password manager with support for PKI authentication methods.

| Operation | Passwords | Symmetric Keys | PKI | Counters | General |
|---|---|---|---|---|---|
| `get` | ✓ | | | ✓ | ✓ |
| `put` | ✓ | ✓ | ✓ | ✓ | ✓ |
| `delete` | ✓ | ✓ | ✓ | ✓ | ✓ |
| `inc/dec` | | | | ✓ | |
| `sign/verify` | | | ✓ | | |
| `encrypt/decrypt` | | ✓ | ✓ | | |
| `authenticate` | ✓ | ✓ | ✓ | | |
| `generate` | ✓ | ✓ | ✓ | | |
| `hmac` | ✓ | ✓ | | | |
| `key exchange` | | | ✓ | | |

**Table 2: The SEEDS API by field type. The policy can restrict operations further conditional on the appropriate user credentials. We support multiple cryptographic algorithms for each operation. The policy and machines are stored in the key-value store itself as a reserved portion of the namespace.**

## 2 OVERVIEW

SEEDS stores all data and metadata as a key-value (KV) store. Each key is of the format `<type>.<user>.<identifier>`, for the type of data, user it belongs to, and unique key. Each piece of data has a specific type corresponding to the data it represents. For example, a private key has a different type from a symmetric key or a monotonic counter.

SEEDS provides a separate API for each type as shown in Table 2. For example, the public-private key type supports `generate`, `encrypt/decrypt`, `sign/verify`, and `delete`. Notice that there is no call to directly read the key. Since SEEDS only allows access through the API, the keys are inaccessible from outside the enclave by all accounts.

The same rules hold for the metadata types, like those for machines or users. For example, the metadata for each machine in SEEDS is represented as a KV pair of type ‘machine’. The API for this type is `get`, `put`, and `delete`.

Each key has an associated policy that further restricts operations on the key. Policies are allow lists that define the API calls allowed by each user for a key. A possible use of policies is, for example, to temporarily revoke the private key a user by preventing all API calls to it. Policies support multi-user environments and prevent compromised users from damaging or leaking other users’ keys. As we show in Section 3, policies are flexible enough to prevent administrators themselves from reading secrets.

SEEDS allows developers to dynamically define new API calls to enable new cryptographic protocols. Developers write scripts that use our cryptographic primitives and other basic functions for string manipulation to implement cryptographic protocols that execute entirely inside of the enclave. API calls are themselves protected by the policies to prevent writing malicious scripts that leaks otherwise inaccessible secrets.

For example, we can create a new SEEDS call that implements U2F authentication. The call increments a counter, concatenates several strings and uses the private key to sign a SHA256 hash of the concatenated string. All operations happen fully withing the enclave, and SEEDS enforces existing policies at each step.

Each node in SEEDS stores a full copy of the key-value store and exchanges updates with other nodes opportunistically. Updates are exchanged using TLS connections terminated inside the enclave. If the device loses connectivity, SEEDS reestablishes communications when connectivity is restored and merges updates with other nodes.

SEEDS implements strong eventual consistency that ensures all nodes converge and presents applications with a consistent view. Strong eventual consistency allows nodes to survive network partitions, offline operations, and even DoS attacks. SEEDS specifically relies on conflict-free replicated data types (CRDTs) [4] (see Section 3) to achieve strong eventual consistency.

## 2.1 Threat Model

SEEDS runs on SGX-capable user devices and allows local applications to submit commands. Communication happens through untrusted IPC or a locally attested channel between an SGX application and a SEEDS enclave.

The threat model assumes a powerful adversary that can control any machine in the SEEDS cluster. As a result, the system can fail to respond to enclave requests, respond incorrectly, or generate malicious requests to the enclave. The adversary can read main memory by performing memory dumps or DMA operations but cannot read or write enclave memory.

All untrusted software can be subverted, including the untrusted code of SEEDS. We assume that there are no exploitable software or hardware bugs within the trusted code or the firmware that can be used to influence enclave code execution. Additionally, the adversary can interfere with network traffic by modifying, dropping, delaying, and reordering packets.

In addition, the attacker can get hold of a user's credentials. With these credentials the attacker can impersonate a user to the SEEDS enclave, and make calls on their behalf. The attacker still cannot directly access the enclave, but only submit API calls on behalf of a valid user. All API calls and operations will still be protected by the policy.

We do not protect against side-channel [13, 20] or speculative execution [5, 7] attacks as they are out of the scope of this work. Existing research has developed side-channel countermeasures and tools to assist in detecting potential speculative execution bugs using code instrumentation and static analysis [15, 16]. Finally, we do not protect against denial of service (DoS) attacks [8] that prevent the enclave from executing.

## 2.2 Security Goals

The security goal of SEEDS is to protect the confidentiality of user secrets from system and user compromise. All SEEDS functions should be secure from any malicious software on the host. We ensure this by using SGX to prevent an attacker from reading or writing enclave data from a compromised untrusted host.

SEEDS should also prevent an attacker from exfiltrating data even when they have access to user credentials. An attacker might use enclave secrets, but should not be able to read them out. That is, the damage of the attack should be limited to the scope and privileges of that user.

Additionally, an attacker should not be able to launch a replay or rollback attack on SEEDS. Replay attacks subvert a program by executing previously authorized operations, while rollback attacks revert state. Hence replaying any requests between an application and SEEDS or updates between SEEDS replicas should not execute or influence the state in any way.

Since updates are applied immediately at the local replica, an adversary can interfere with a replica and prevent it from discovering new updates. As a result, the disconnected replica will service local requests with stale data. SEEDS should gracefully continue execution after connectivity is restored. Moreover, SEEDS should be usable for local authentication while offline.

```
passwd.* allow ANY:{authenticate} OWNER:{get,put}
passwd.pw allow ANY:{authenticate} OWNER:{put}
policy.* allow ANY:{get}
```

**Figure 1: A subset of the SEEDS policy for passwords in our LDAP replacement application. Access is guarded by both a global and a local policy. The user cannot change the password with ID `pw` because the local policy does not allow them, even if the global one does. The final policy prevents any policies, including itself, from being modified.**

## 3 DESIGN

SEEDS has three main elements: the API and policy, extending functionality through scripts, and the replication mechanism.

## 3.1 API and Policies

Table 2 shows the SEEDS API for each type. SEEDS provides applications with a key-value store and a rich set of cryptographic primitives. The cryptographic primitives operate on passwords, symmetric keys, public-private keys, and counters. We chose these primitives because they allow us to support a wide range of application.

The table shows that by default certain types of data can be but not read. This includes PKI and symmetric keys, which can be used for authentication and encryption/decryption. We allow reading passwords from the enclave by default, but developers can prevent reading passwords through the use of policies.

SEEDS represents every piece of information as a tuple in the key-value store. This includes metadata such as policies and host membership. A policy's key is the identifier of the KV pair it corresponds to, and its value is the set of operations allowed by a user (e.g., `<user>:{<operation>,...}`). The user can refer to a particular user or be `ANY` if it refers to all users, or `OWNER` if it applies to the owner of the key.

Policies can globally apply to types. For example, an application might declare all passwords to be writable by their owners, but not readable. This prevents the passwords from leaking in the event of a compromise. For an operation to succeed it should be permissible by both the global policy and any applicable per-key policy. This prevents an attacker from adding policies that give them read privileges to an unreadable key.

Policies are key-value pairs in SEEDS and their permissions are thus also governed by policies. Users can turn policies read-only to prevent them from being modified by a malicious user. We use a type based policy to permanently block policy updates. This policy also applies to itself, preventing attackers from undoing it.

Figure 1 shows a subset of the policy for our LDAP replacement. The password for user `U` for key `pw` uses a global policy that allows its owner to call `get` and `put`, and a per-key policy that allows only `put` calls. Both policies allow any user to call `authenticate`. Since a call is only allowed if allowed by all applicable policies, the user `U` can change the password using `put`, but not read it. Other users cannot change user `U`'s password. The last policy in the figure turns all policies including itself read-only by preventing `put` calls. This prevents an attacker from adding `get` to the list of allowed calls.

```
u2f_authenticate(appID, U2FKeyID, challenge)
{
    counter = seeds.inc(U2FKeyID+"_cnt");
    presentBit = 1;
    response = concat(appID, presentBit, counter,
               challenge);
    hash = sha256(response);
    signature = seeds.sign(U2FKeyID+"_key", hash);

    return (counter, signature);
}
```

**Figure 2: The dynamically loaded SEEDS function for authenticating using the U2F protocol. The user sends their credentials, the unique ID of the application requesting authentication, and the remote host's challenge. The function signs the authentication response and increments the token's counter.**

## 3.2 Safe Extensibility

SEEDS allows developers to extend the API by adding new cryptographic protocols using an embedded interpreter. The interpreter allows small scripts to chain together cryptographic operations on keys and basic string manipulations inside of the enclave. This allows SEEDS to expose new cryptographic protocols without requiring any code changes and protect the intermediate steps of any cryptographic protocol.

New API calls are implemented as scripts that execute other key-value store cryptographic primitives. These calls run with the privileges of the user that invoked them and the policy is enforced on the individual cryptographic calls. The script does not have direct access to the key-value store, which prevents a malicious administrator from undermining the security model.

The interpreter runs a simple subset of Scheme that prevents malicious scripts from damaging the enclave. The scripts cannot make OCALLSs, and are prevented from using more than a few kilobytes of memory. This is because API calls are expected to combine together cryptographic primitives to implement protocols, and do not do any significant processing.

Figure 2 shows pseudo code for the U2F authentication provided by our password manager. It uses a private key and a monotonic counter to implement the U2F protocol. The U2F protocol uses the monotonic counter when sending authentication requests to prevent replay attacks. The protocol concatenates this counter with the application ID and the challenge, and hashes the result. It then signs the hash with the private key and sends the result out. All the steps in the process are simple and in this case the algorithm has no flow control. We can implement the authentication function as a new API call that uses SEEDS' already existing PKI and monotonic counter data types.

## 3.3 Distributed Updates

SEEDS has relaxed consistency guarantees to ensure availability regardless of connectivity. The relaxed consistency guarantees do not affect the system in practice. Updates are relatively uncommon because users create new keys infrequently. Moreover, most users are typically using one or more connected machines at the same time and they mostly update their own data. Thus concurrent updates from different users are rare in most applications we have considered.

Machines use point to point TLS sockets that are terminated inside the enclave for updates. Enclaves combine SGX attestation with the TLS handshake [10], embedding the enclave report in the X.509 certificate. Enclaves prevent forking attacks by inspecting each other's global version vector (see below) during TLS setup to ensure they are not stale.

SEEDS reconciles state updates between machines using conflict-free replicated data types (CRDTs). SEEDS uses state based CRDTs that send all of enclave state instead of just the operations to make reconciling state easier. SEEDS has CRDT operations that correspond to the KV store's `put/get/delete` operations.

These three operations are not commutative, so we define a deterministic resolution strategy to ensure the CRDTs converge. A `put` supersedes concurrent `deletes`, and concurrent `puts` use last-writer-wins semantics. We define 'last' using the version vector as our notion of time relative to other hosts. Concurrent updates are deterministically resolved using the machine identifier.

SEEDS also offers a monotonic counter data type that is trivial to implement as CRDT. We use it exclusively for the authentication U2F token in our password manager application (see Section 4).

**Conflict Resolution** SEEDS uses Optimized OR Sets (Opt-OR Sets) [4], for CRDTs. Regular OR Sets [18] define last-writer-wins semantics using tags whose total size grows indefinitely. Opt-OR sets retain these semantics and ensure bounded space usage without garbage collection (GC [21]). GC methods require acknowledgements and ordered message delivery, both incompatible with SEEDS' model.

Opt-OR sets need a local version vector for each key value pair on each replica, and a global one for the whole key-value store. SEEDS implements updates by sending out the store's state all at once as a single message. State based CRDT replication allows us to handle out-of-order state updates. SEEDS uses each key-value pair's local vector to enforce the CRDT's semantics when merging.

The version vector modestly increases the space requirements for each pair, but is acceptable for our applications. Similarly, the bandwidth cost for sharing state based updates is low because each key-value pair is small. It is also only incurred only when SEEDS sends out updates, which happens infrequently because of relaxed consistency.

# 4 APPLICATIONS

We have been developing two applications on top of SEEDS to explore these ideas. First, we built a decentralized user management system to replace the traditional LDAP plus Kerberos architecture. Second, we built a password manager that uses SEEDS to replicate passwords and keys between a user's devices.

## 4.1 Decentralized LDAP/Kerberos Alternative

Lightweight Directory Access Protocol (LDAP) and Kerberos are popular protocols for managing user accounts and login credentials. They are often used together, with LDAP managing account information and Kerberos managing credentials. Each machine is issued keys that give it read-only access to LDAP and Kerberos to prevent making account information widely available. Correctly configuring these services is difficult. These services are also a single point of failure and susceptible to DoS attacks.

To improve performance and allow offline operation administrators often use a persistent cache on clients that stores password hashes and account information. A variety of programs triggers LDAP calls, e.g., running `ls` requires mapping user and group IDs to names that result in costly calls to the LDAP service if the information is not cached. By persisting the cache, users can continue to use their laptops while offline.

*SEEDS Decentralized User Management Service.* We built a decentralized user management service using SEEDS. To add a machine to the cluster, the administrator initializes the SEEDS enclave and pairs it with any existing machine. Even though the service is decentralized, only specific users can add and remove hosts, users and groups from the system.

Our user management service consists of a command-line tool for setup and administration. On each host, we install our Pluggable Authentication Modules (PAM) and Name Service Switch (NSS) modules. These modules use SEEDS to implement several functions that service user and group information from the local replica.

Administrators can use our command-line tool for adding a new host to an existing cluster, and managing users and groups.

The PAM module, `libpam_seeds.so`, implements the PAM authentication and account APIs to allow machines to login by validating passwords or hardware tokens against the SEEDS database. Applications like `login` and `su` use PAM to call into the enclave to verify user credentials.

The NSS module, `libnss_seeds.so`, provides the POSIX C library with access to user and group information including a user's UID, home directory, shell, group membership. NSS provides access to all the information a POSIX system would typically query from the `passwd` and group files. The NSS API allows several other files in the `/etc/` directory to be served to hosts.

We implement the service by storing user passwords in SEEDS using the key `passwd.<user>.pw`. The password type by default has the methods `put`, `get`, `delete`, and `authenticate`. We remove the `get` method from the type as shown in Figure 1 to prevent reading a password and only allow the `put` method by the owner, and `authenticate` method by anyone. The `delete` method is available only to the root user.

The service is distributed and eventually consistent, which provides similar guarantees to a multi-master OpenLDAP deployment. Our consistency guarantees ensure that machines can use the SEEDS NSS/PAM modules to log in locally regardless of connectivity. Complications due to the eventual consistency model are minor and also can occur in any multi-master deployment.

**Kerberos Functionality**

We can simplify our Kerberos implementation to skip large parts of the protocol since the service is now distributed and includes the client's local machine. The Kerberos ticket granting service (TGS) normally first creates a ticket granting ticket (TGT) for the client that it encrypts with a hash of the client's password before sending it over an untrusted channel. The client uses the TGT to request service tickets from the TGS to authenticate with other machines.

We can instead create service tickets for network services directly from SEEDS without needing a TGT and a TGS. SEEDS can locally generate the equivalent of a service ticket after authenticating the user. To fully support the single sign-on benefits, we would need to implement a GSS authentication module similar to Kerberos. Unlike Kerberos, we do not need the keytab for the machine and service keys to be stored in plaintext.

## 4.2 Password Manager and Virtual Token

We built a password manager based on SEEDS that replicates credentials between a user's devices. Updates to the credentials stored in SEEDS propagate to all devices when those have connectivity. The SEEDS password manager stores passwords and public-private keys. PKI keys can be used as a software U2F token.

The password manager has a software Universal 2nd Factor (U2F) token. The U2F protocol provides two-factor authentication (2FA) to protect user accounts even when an attacker successfully steals their password. Most U2F tokens are physical devices which means that losing them locks a user out of their accounts. Users usually need to register multiple U2F tokens for each account and store them in separate places which is time consuming and a burden for many.

To address this practical issues we built U2F support into our password manager to emulates a U2F token through SEEDS. This simplifies the use of U2F to prevent phishing or stealing passwords but does not provide a physical button due to limitations in SGX. These U2F tokens are available on all devices a user owns, allowing the user to not worry about registering and securing multiple physical tokens. They protect the U2F authentication process, essentially a PKI signature, with the use of a password or pin.

We use `libcuse` to emulate a U2F HID USB device that looks like a physical token to applications. Web browsers and OpenSSH use the standard `libfido2` library to authenticate the user against the software token. No code changes to applications or `libfido2` is necessary.

We implement the authentication operation using the code shown in Figure 2. The code implements the construction of the authentication response using the initial challenge and the unique key identifying the U2F private key. We use the private key only for signing without exposing it to the untrusted host.

*Thwarting Replay Attacks.* An issue with the SEEDS U2F token is the reconciliation of counter updates. U2F devices use monotonic counters to detect stolen credentials. If the U2F counter is stored in the key-value store, the highest counter should always be used. Our CRDT counter is monotonically increasing even in the case of conflicts.

However, the asynchronous nature of SEEDS makes it possible to use the token from two machines against the same service, resulting in the service believing a key may have been compromised. Services typically alert the user of this potential issue. If the user runs SEEDS across all their devices they are unlikely to physically travel between two machines that have no connectivity to each other. Carrying any device running SEEDS would prevent such a case.

## 5 DISCUSSION AND FUTURE WORK

**Policy Namespaces** SEEDS currently has a set format for the key namespace using each key's type and user. This has implication on the granularity of policies, which are either per-key or apply to all keys of the same type. For example, having a policy that turns all policies read-only prevents us from adding any more policies to the enclave. This limits us from expanding enclave functionality.

We are looking into adding arbitrary namespaces in the enclave that are decoupled from key types and users. The type of each key in this scheme is not part of the name. This change allows us, for example, to turn all policies in a namespace read-only without interfering with other namespaces.

**Dynamically Added Types** SEEDS supports adding scripts that chain together existing commands, but does not allow new ones. This prevents attackers from adding malicious calls that leak otherwise unreadable keys. This is not a limitation because we only store secrets, which are mathematical objects with a set API of available operations. Adding a new fundamental operation for, e.g., a private key is improbable.

We are adding the option to dynamically load new types into SEEDS. This gives us forward compatibility with new cryptographic primitives. Developers load the new type into SEEDS along with all valid operations. The type's API is set at load time and cannot be expanded, much like that of builtin types. An attacker can thus add maliciously crafted types, but cannot subvert the security of existing ones. SEEDS provides an API to allow all users to inspect dynamically loaded functionality.

**SEEDS Language Semantics** SEEDS prioritizes the safety of the scripting language. The interpreter's performance does not matter, because cryptographic computations are done using builtin native functions that dominate processing time. It does matter, however, for the dynamically added types described above. For future work we are looking into ways of implementing cryptographic primitives efficiently.

**Mandatory Access Logging and Rollbacks** We are implementing a mandatory access logging (MAL) mechanism for SEEDS to allow for auditing. This logging mechanism is part of the main KV data structure and interposes on operations. The implementation of the log is simple and is just another type with an associated namespace.

State-based replication captures multiple operations in a single update, and gives all or nothing behaviour we expect. Only a few of the most recent operations will be kept to ensure the logs do not consume too much space. MAL can help users determine if an adversary gained physical access to their device, since no operations should succeed without authentication.

The MAL also allows us to implement rollbacks of state. We currently use state based CRDTs, which allow eventual consistency without message delivery guarantees. However, since they also overwrite past state they force us to implement simple conflict resolution policies. With logging we are able to roll back individual operations using the semantics of SEEDS operations and resolve conflicts between dependent operations. The MAL could also allow us to apply undo computing [9] techniques to repairing damage from a compromised user account.

## 6 RELATED WORK

There has been extensive research on storage systems based on SGX[2, 3, 11, 19]. These systems are performance oriented and assume connectivity between the nodes of the system. They use strong consistency at the cost of availability. Moreover, these systems use SGX to prevent a system compromise from allowing an attacker to access sensitive data. They do not protect against an attacker with access to the right credentials from exfiltrating data.

TPMs have also been implemented as an abstraction in software. cTPM [6] uses a remote machine as a TPM, assuming a preshared key between the remote machine and the local node. Compromising the remote machine permanently compromises the preshared key. fTPM implements a software TPM for a single node using firmware and ARM TrustZone, but is easily applicable to SGX.

CKS [12] provides an interface similar to that of a TPM, providing encryption and decryption capabilities using a key. CKS allows auditing every key operation and thus requires constant connectivity, which prevents high availability and offline operation. Moreover, CKS faces the issue that an attacker with access to a user's credentials can abuse the enclave update feature. The attacker can 'update' the enclave to a version they have created, which in turns directly exports the secrets. SEEDS is resistant to such compromise.

The use of interpreters in enclaves is a well-established practice. Twine compiles code to WebAssembly before running it in the enclave [14]. MapReduce for SGX uses a Lua interpreter to run user defined map and reduce scripts at runtime [17]. We adopt similar techniques, to support expanding the enclave API at runtime without changing the SGX enclave quote.

## 7 CONCLUSION

We present SEEDS, an SGX based decentralized, highly available secrets manager with an expressive cryptographic and policy API. SEEDS is based on the observation that a lot of security oriented applications use simple logic on top of a storage layer. Our system prevents attackers from extracting secrets from the enclave even when they compromise the system and intercept user credentials. It is also distributed and highly available even on devices with intermittent connectivity. We demonstrate the system's usefulness by using it to secure two security sensitive applications.

## REFERENCES

[1] [n.d.]. Trusted Platform Module Library Part 3: Commands. https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.38.pdf. Accessed: 2022-01-10.

[2] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Vijay Nagarajan, Pramod Bhatotia, et al. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 65–79.

[3] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 173–190. https://www.usenix.org/conference/fast19/presentation/bailleu

[4] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368* (2012).

[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[6] Chen Chen, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. cTPM: A cloud TPM for cross-device trusted applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (Euro SP)*. 142–157. https://doi.org/10.1109/EuroSP.2019.00020

[8] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (Shanghai, China) *(SysTEX'17)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3152701.3152709

[9] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/intrusion-recovery-using-selective-re-execution

[10] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863* (2018).

[11] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3190508.3190518

[12] Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N Asokan. 2018. Keys in the clouds: auditable multi-device access to cryptographic credentials. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 1–10.

[13] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[14] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An Embedded Trusted Runtime for WebAssembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 205–216. https://doi.org/10.1109/ICDE51399.2021.00025

[15] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. 2019. DifFuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 176–187.

[16] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*. 1481–1498.

[17] Rafael Pires, Daniel Gavril, Pascal Felber, Emanuel Onica, and Marcelo Pasin. 2017. A Lightweight MapReduce Framework for Secure Processing with SGX. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 1100–1107. https://doi.org/10.1109/CCGRID.2017.129

[18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types.* Ph.D. Dissertation. Inria–Centre Paris-Rocquencourt; INRIA.

[19] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. Rkt-Io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 490–506. https://doi.org/10.1145/3447786.3456255

[20] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2421–2434. https://doi.org/10.1145/3133956.3134038

[21] Gene TJ Wuu and Arthur J Bernstein. 1984. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*. 233–242.