SEEDS: Secure Decentralized Storage for Authentication Material

by

Stefanie Dukovac

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Applications that use passwords or cryptographic keys to authenticate users or perform cryptographic operations rely on centralized solutions. Trusted Platform Modules (TPMs) do not offer a way to replicate material, making accessing this information in a heterogeneous environment difficult. Meanwhile, remote services require a constant network connection and are a central point of failure.

We present SEEDS, a secure decentralized multi-user data store that generates, stores, and operates on users' authentication material such as passwords and cryptographic keys on local machines. To ensure the confidentiality and integrity of user accounts and cryptographic keys, SEEDS leverages Intel SGX —a hardware-based trusted execution environment, to store and operate on this data while protecting from a compromised host. We support user-defined policies that restrict users' operations to protect against a malicious user attempting to access data without sufficient privileges. In addition, we replicate data across machines to improve accessibility and support offline participants for high availability. We implement the storage data structure using Conflict Free Replicated Data Types (CRDTs) to replicate data, recover from network partitions gracefully and offer a horizontally scalable system.

We developed two applications that demonstrate the benefits of our system. First, we address centralized user authentication issues by implementing a database module that replaces and decentralizes LDAP user authentication. Next, we improve the management of users' cryptographic keys by developing a software U2F token that replicates this material across machines for high availability.

Acknowledgements

I would like to thank my supervisor — Prof. Ali Mashtizadeh, for his guidance and feedback during my graduate studies. I would also like to thank my parents for their unwavering support throughout my academic journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Managing authentication material in a secure and easily accessible way is a challenge. Both users and services rely on key-management services for authentication and cryptographic operations such as signing and encryption. Authentication is paramount as it protects sensitive resources from unauthorized access, therefore, the integrity and confidentiality of the authentication process are required.

Users typically resort to storing and copying this information insecurely to improve availability across machines. Solutions such as password managers store the data directly on the user's device and are known to suffer from flaws that expose passwords and encryption keys in memory [6]. As a result, malicious software on the host can learn these secrets and access the user's sensitive data. In addition, password managers do not protect against an adversary with system read access.

Alternatively, users may use dedicated hardware such as Trusted Platform Modules (TPMs) to securely store, manage, and operate on this data, even on an untrusted host. This way, even if a machine were to be compromised, the contents of the TPM are isolated and protected. However, TPMs have a restrictive API that only supports cryptographic operations and is relatively complex to use with the specification [8] containing 125 functions. Furthermore, TPMs offer a single-machine solution but lack availability across machines. They cannot replicate the data, requiring users to register a new key with each service

on new machines, which unnecessarily complicates the key management problem for users and services.

Cloud-based solutions such as Cloud Key Store (CKS) [29] rely on network connectivity in order to use the service. Solutions that require network connectivity are susceptible to DDoS attacks and can suffer from availability issues when immediate access to services is required. Alternatively, a cloud TPM solution such as cTPM [19] does not require constant connection since data is synced with the cloud periodically and accessed locally upon request. This approach establishes a pre-shared key between TPMs and the cloud, which is used to encrypt and authenticate messages, requiring the cloud to remain trusted. cTPM expects the key to be shared during TPM manufacturing which requires changes out of the end-user's control making this solution idealistic.

Trusted execution environments (TEEs) support isolated execution of programs that protects them from unauthorized access or modification. Even privileged software cannot access these regions, ensuring confidentiality and integrity of program code and data. TEEs are available in commodity hardware including smartphones, tablets, and laptops. Implementations such as Intel SGX [5] and ARM TrustZone [1] provide extensions to existing architectures that use hardware-based access control to provide a secure virtual processor.

There are a lack of services that use this technology to secure applications directly. Typically services using Intel SGX use a remote access model, where users connect to the service in a TEE on an untrusted cloud provider. The threat model of these services assumes that the host running the TEE can be compromised while the remote user remains wholly trusted.

SEEDS bridges the gap between users and their data by providing a secure distributed multi-user data store with policy-based access control. SEEDS runs in a TEE where it manages user accounts, generates and stores keys, and performs cryptographic operations such as verification, signing, and encryption. That is, SEEDS offers a flexible interface that allows users to securely store arbitrary data within the TEE while supporting standard TPM functionality. In addition, SEEDS enforces policies to protect the confidentiality and integrity of the data by authorizing each operation.

All data is stored in a key-value store and securely replicated to improve reliability and availability. Updates are made independently, without coordination, and applied in any order at each replica while still converging to the same state. We achieve this functionality by implementing the key-value store as a Conflict-Free Replicated Data Type (CRDT), allowing SEEDS to service local requests without requiring immediate connectivity. SEEDS only requires connectivity to share system updates, thus making the system highly available.

We developed and run SEEDS across a cluster of commodity desktop computers with Intel SGX. We implemented two services using SEEDS that show the different kinds of applications possible with the system. First, we built a decentralized replacement for LDAP authentication that replicates the data securely across machines to ensure high availability in a multi-user environment with more complex policies. Second, we developed a single-user replicated U2F software token.

Chapter 2

Background

SEEDS provides a key-value store service that protects high-level operations through a fine-grained policy layer. The entire service runs inside of a TEE and is accessible through a library. Individual cryptographic operations, including key generation, verification, encryption, and data signing are mediated through the policy.

The key-value store is represented internally as a composition of CRDTs to allow updates to happen asynchronously to support end-user environments where individual devices may come online and go offline anytime. Using the properties of CRDTs enables replicas to make updates and resolve conflicts independently and in a deterministic way.

2.1 Intel SGX

A trusted execution environment (TEE) isolates code and data to prevent other software, including system software, from reading or tampering with it during execution. Both hardware and software approaches exist to achieve these strict guarantees. Intel Software Guard Extensions (SGX) implement a hardware-based approach using custom on-chip technology and microcode extensions to create and mediate access to a protected memory region of main memory.

Intel SGX applications maintain separate trusted and untrusted parts that communicate through a call-gate mechanism. This allows the untrusted part to change the privilege level of the current process to execute the trusted code. The code and data of the trusted part reside in the protected region of memory known as an enclave. The memory is protected by an on-chip memory encryption engine (MEE) that encrypts and decrypts enclave memory (at cache line granularity) to enforce confidentiality and integrity[21].

The enclave contents are loaded into a single page within the protected region of memory which has a size of 128 MB in total. Approximately 32MB is dedicated to enclave metadata [12, 37], leaving 96 MB for pages to be shared amongst enclaves. Page swapping is supported but incurs high overheads. Therefore, developers must minimize the footprint of enclave code and data.

Intel SGX offers two methods of attestation: local and remote. Local attestation proves the identity of the enclave on the same host using a MAC and a symmetric key unique to the enclave that is protected by the hardware. Remote attestation proves the identity and contents of an enclave to a remote party and requires both a report and a signature of the report using the hardware key burned into the chip. Upon initialization, the CPU computes thr report which is a secure hash of the enclave's code and initialization parameters. The remote party must verify the report and ask Intel's Attestation Service (IAS) to verify the quote to ensure the enclave is running on genuine Intel hardware [2].

A background enclave runs independently and provides access to on-chip services such as encryption keys, counters, and trusted time. Data can be secured and persisted to untrusted storage through a process known as sealing. A sealing key is generated and used to encrypt the data. Only certain enclaves can regenerate the decryption key depending on the key generation policy. System software does not have access to the key, and therefore the data is securely encrypted before being offloaded to untrusted storage. Furthermore, an enclave can use the counters within replay-protected storage to ensure freshness of sealed data.

We strategically isolate SEEDS system components to run in an enclave using the Intel SGX SDK. There are several existing frameworks for running unmodified applications

in an enclave [44, 12], however, these frameworks are not compatible with the OS we used for system development. Furthermore, using the SDK lets us directly interface with trusted and untrusted parts, allowing for fine-grained control over the memory usage and performance.

2.2 Conflict Free Replicated Data Types

Distributed systems replicate data to improve reliability and performance, which requires a trade off between consistency and availability according to the CAP theorem. In systems where availability is essential, replicas independently handle updates causing replicas to diverge as updates are applied. Each replica propagates updates to other replicas and applies updates locally to reconcile state. Since updates occur concurrently and can conflict, replicas must agree on how to resolve conflicts which usually relies on consensus. CRDTs offer a unique solution to resolve conflicts that arise from concurrent updates without communication between replicas.

CRDTs extend a subset of traditional data structures to reconcile conflicting updates by strictly relying on additional metadata. As a result, CRDTs can integrate updates from other replicas and reconcile any conflicts without consensus. Instead, all operations are performed locally and independently, and they eventually propagate to other replicas by sharing updates. Each replica will resolve any conflicting updates deterministically and modify their copy of the data structure accordingly.

CRDTs guarantee that after no new updates, all replicas will converge to the same state assuming all updates are delivered. This consistency model is known as strong eventual consistency (SEC) and offers a greater guarantee than eventual consistency [41]. Eventual consistency may require replicas to roll back state and perform consensus to agree on the order of updates and converge to a final state. Meanwhile, SEC avoids the overheads of consensus because each replica can make these decisions independently.

The resolution of conflicting updates is known as concurrency semantics and depends on the underlying data structure. If operations of the data structure commute, the data

structure's final state converges regardless of ordering. Some data structures have a natural CRDT implementation. For example, consider a monotonically increasing counter initialized to 0. We expect two concurrent increments to resolve to the sum of the two increments, since the order in which these increments are applied does not influence the final value. Applications must be able to support the discrepancy of this counter in their application. Not surprisingly, many data types such as registers and sets do not have updates that naturally commute. For example, what should the final register value be when two concurrent updates set the register to differing values? We must define the outcome of such updates which depends on the application using the CRDT.

There are two main approaches to implementing CRDTs: state-based and operation-based. Updates are defined differently between the two but require that conflicting updates commute. This ensures that updates can execute at different orders at each replica.

A state-based implementation defines an update as the entire data structure itself and defines a merge function to merge the data structure in the update into the replica's local version. Since a merge is a union of two states, this should form a least-upper bound (LUB) and ensure a single outcome given any two states [41]. As a result, the merge function must be associative, commutative, and idempotent. Associativity ensures that the order of these merges does not impact the final result. Idempotence guarantees that we can merge the same state multiple times and achieve the same result. A consequence of the state-based implementation is that updates grow proportional to the amount of data stored within the data structure. Therefore, updates can become very large according to the size of the data structure.

An operation-based implementation defines an update as the metadata concerning the operation (i.e., the operation and parameters). Updates are twofold; each update implements a prepare phase and an effect phase. The prepare phase prepares metadata at the source replica. The effect phase occurs at all replicas and uses the metadata to update the data structure. The effect phase must be commutative concerning concurrent updates to avoid ordering across replicas. Typically, operation-based implementations assume causally-ordered communication to relieve the effect phase from being commutative

for all operations (not just concurrent ones) [15]. If the effect phase is not commutative for all updates, we require the messaging middleware to deliver updates in the same order that they were applied at the source replica.

Chapter 3

Threat Model and Security Goals

3.1 Threat Model

We differ from the typical SGX threat model where trusted users establish an attested and secure communication channel to an enclave running in a cloud provider's machine. Instead, SEEDS runs on SGX-capable user devices and allows local applications to submit commands using IPC or a locally attested channel between an SGX application and SEEDS enclave.

We assume hosts are initially trusted, and *may become* untrusted after initialization of SEEDS. An untrusted host may be compromised, meaning an adversary has some control over the system software. As a result, the system can fail to respond to enclave requests, respond incorrectly, and issue requests to the enclave. We expect a secure I/O between the user, the application and the SEEDS service. Therefore, we do not protect against key loggers or other malicious system software that reads or tampers with the I/O.

Furthermore, the applications using SEEDS are expected act in good-faith. We assume that there are no exploitable software bugs within the trusted code that can be used to influence enclave code execution. Additionally, the adversary can interfere with network traffic by modifying, dropping, delaying, and reordering packets.

We do not protect against side-channel [32, 46] or speculative execution [20, 17] 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 [36, 35]. Finally, we do not protect against denial of service (DoS) attacks [25] that prevent the enclave from executing.

3.2 Security Goals

The main goal of SEEDS is to ensure confidentiality and integrity of the generated data by preventing unauthorized access or modification. All SEEDS functions should be secure from any malicious software on the host. We ensure this by generating, managing, and operating on the data in an Intel SGX enclave using an access control policy.

We protect the data from compromised users by enforcing access control policies on the data itself. Users only access protected data if they are authenticated against the system, and there exists a policy to authorize the operation. An attacker who learns a user's login credentials and impersonates a user should not access or modify data that the compromised user does not have permission to access. That is, the damage of the attack should be limited to the scope of that user. Of course, if the compromised user has many privileges (i.e., permitted to modify policy), then the attacker can affect how other users interact with the data through policy modifications. Nonetheless, this can be thwarted by strict initial policies that prevents any user from making security-sensitive modifications.

Additionally, an attacker should not be able to launch a replay or rollback attack on SEEDS. Replay attacks attempt to 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.

Chapter 4

Design

SEEDS uses Intel SGX to secure system components, so it consists of a trusted and untrusted part as shown in Figure 4.1. The trusted part, or enclave, maintains all user account information, keys, policies, and other metadata using an in-memory key-value store. In contrast, the untrusted part manages communication with applications on the same machine and the correspondence with other replicas.

Applications invoke operations listed in Table 4.1 by issuing requests to SEEDS. All operations expect a uid to associate it with a user. Applications first issue an authentication request to bind a user's identity to the session. Then, the subsequent request is authorized by checking a list of policies that permit the request based on the operation and identity of the authenticated user. Only if authorization succeeds is the request performed. Optionally, applications can issue operations without prior authentication and an empty uid if the operation is permitted by all users on that data. Standard applications connect to SEEDS through IPC, while SGX applications can connect directly to the SEEDS enclave using local attestation. The latter ensures that the host cannot read or modify the contents of the request. We discuss more details in § 4.1.

All operations are performed locally, making the system highly available. Replication is transparent; replicas send, receive, and apply updates in the background while servicing requests locally. This update propagation method does not ensure that all updates on

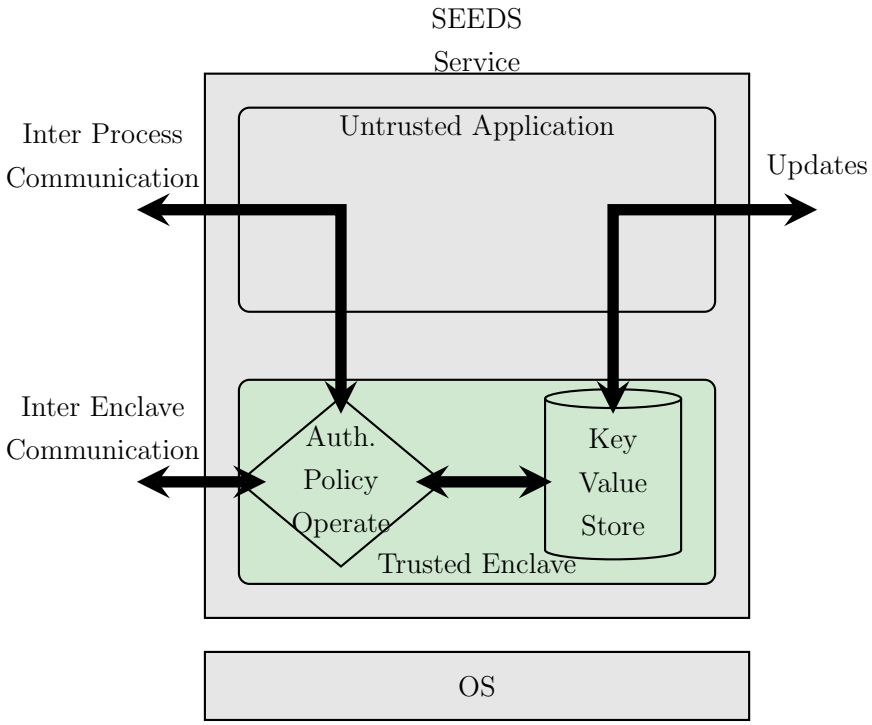

Figure 4.1: The basic architecture of SEEDS.

the data are immediately reflected at each replica. As a result, applications must tolerate temporary inconsistencies.

4.1 Users and Operations

SEEDS is a multi-user service, so users must verify their identity before accessing or operating on sensitive data. Each user maintains an account with SEEDS, corresponding to several unique entries in the key-value store that holds account information such as a username, user identifier, and login credentials.

Accounts are initialized with a password that authenticates the user to SEEDS. The username and password are passed to the auth() operation using the uid and cred parameters, respectively. We rate-limit the number of authentication attempts to prevent an adversary from guessing credentials. Each failed attempt exponentially increases the timeout for the next authentication attempt. We use Intel's trusted time primitive to accomplish this timeout in the enclave.

To change credentials, users call update_user() with the value of the new password that will update the account information for the specified user. The user must be authenticated prior to calling this operation. Policies restrict the operations users can perform, so a policy must exist that allows users to modify their credentials. More details about policies are covered in § 4.2.

Intel SGX does not provide secure I/O to the enclave, therefore, the host can read the credentials passed to SEEDS during each operation over an IPC connection. We do not take any precautions from an adversary intercepting request parameters. Instead, it is the application's responsibility to correctly and securely gather these credentials and provide them to SEEDS. Ideally, we want a secure channel between the user and SEEDS enclave over the host system. SEEDS supports locally attested communication channels with SGX applications to secure I/O between the application and the SEEDS enclave. However, such applications must still take measures to prevent intercepting input from a device. A possible solution is shuffle mapping, where the software scrambles the system message queue [47] to prevent an adversarial host from reading the keyboard input of the password.

If a user attempts to access other users' data, we expect that the user has sufficient permissions to view or modify it. Otherwise, the operation should fail since our goal is to protect data from unauthorized access or modification. Flexible policies describe which users can perform what actions on specific data and are checked on each operation. Next, we consider how data is stored to allow for such policies.

Operation(s)	Input
get	uid, name
put	uid, name, value
delete	uid, name
create_user	uid, new_uid
update_user	uid, cred
auth	uid, cred
gen_key	uid
sign,verify	uid, name, bytes
encrypt,decrypt	uid, name, bytes

Table 4.1: SEEDS API.

4.2 Entries and Policies

We store user accounts, cryptographic keys, policies, and all other data within entries inside an in-memory key-value store. The key-value store resides in enclave memory, therefore, the total number and size of entries are restricted by the memory limit. However, all entries are relatively small, so storing all this data in the enclave is feasible assuming a moderate number of entries.

Entries are key-value pairs, where the key is a string, and the value holds arbitrary data associated with the key. To avoid confusion with cryptographic keys, we refer to the key of an entry as its name. Operations in Table 4.1 interface with the key-value store using get(), put() and delete() operations. A get queries the key-value store for an entry, while the put inserts or updates an entry, and the delete removes an entry. For example, gen_key() generates a cryptographic key and stores it in the key-value store using a put. Meanwhile, sign() reads the key from the store using a get and then performs the signature.

To distinguish entries all entry names follow a naming convention that identifies the entry's type, owner, and resource name. This convention allows us to store different data within the key-value store while tagging similar data with a similar prefix. Specifically, the

entry names are period separated as follows: 'type.owner.resource', where the type is one of three types: accounts, machines and policy. The owner field maintains the username of the user who owns the entry. Specifying the owner simplifies access control on the entry. The resource maintains the name of the data held by the entry. Table 4.2 provides an example of some standard entries.

Each user maintains several entries with the accounts type that store data related to the user's account such as account information, as well as any SEEDS generated keys. Similarly, each 'machines' entry maintains the domain name of a machine in the replica group. Replicas share and receive updates from these machines, and we discuss more details about this in § 4.5. A single entry with the name 'policy' maintains a global list of policies that define relationships between entries, users, and operations. The owner and resource fields are omitted from this entry since it maintains information relevant to all users in the system and is used on each operation.

This organization of keys makes defining policies on the entries straightforward. Each policy is a string describing the permitted operation and consists of the desired entry, username, and operation. Policies may contain specifiers ¡USERNAME¿ and ¡ANY¿, where the former matches the username of the user issuing the operation, and the latter matches any username or operation. This way, a single policy can apply to more than one user or operation making it very flexible. Table 4.3 holds a list of example policies where each policy maps an entry to one or more usernames and the operation(s) that are permitted on the entry. For example, the first entry in Table 4.3 indicates that all users can perform a get on every user's email. Meanwhile, the second entry indicates that the user issuing the operation can only update their own email entry.

SEEDS validates all operations against the policies which are found in the policy entry. On each operation, a string describing the current operation is created and compared against each policy in the policy entry until a match is found. If there is no matching policy the operation fails, otherwise, the operation executes. Policies give explicit access since it is inherently safer to list the permitted operations rather than those that are not. The validation occurs entirely within the enclave, so an adversary cannot influence

Entry Name
policy
machines.admin.domain_name1
accounts.admin.userid
accounts.admin.passwd
accounts.alice.userid
accounts.alice.passwd
accounts.alice.key_rsa3072

Table 4.2: Sample key-value store entries in SEEDS.

Entry	**Username**	**Permissions**
accounts.<ANY>.email	<ANY>	get
accounts.<USERNAME>.email	<USERNAME>	put
accounts.<ANY>.pub_key	<USERNAME>	encrypt
policy	admin	put
machines	admin	put, delete

Table 4.3: List of policies that map key-value store entries, to users and permitted operations. The <USERNAME> specifier matches to the user preforming the action, and <ANY> matches to all valid values in that field.

the policy decision. Users can only influence the policy if they are permitted to do so according to the policy itself. The administrator is responsible for providing a correct list of policies that are loaded on initialization and done while the device is trusted to ensure the host cannot tamper with it.

As mentioned in § 4.1, we do not protect against attacks where the parameters of the operations are compromised. The security of SEEDS depends on the set of policies, and for operations that make changes to the policy such an attack could seriously impact

confidentiality and integrity of data. It is up to the application to ensure any operations that make such changes occur on a trusted host or through a trusted channel. Furthermore, the adversary does not gain anything from learning some operations' parameters since data is not revealed. For example, the sign() operation keeps the signing key in the enclave and only requires a username, key handle, and bytes to be signed. An adversary should not be able to forge a signature or learn the signing key with only the parameters and signed data.

4.3 Replication

The key-value store maintains all user and system data and is replicated across hosts. Using a single key-value store ensures all data is accounted for and we do not need to replicate multiple data structures. We trade-off consistency for availability to ensure that SEEDS can always complete a request since the data stored does not frequently change and can tolerate moderate divergence. Most data is unique to an individual user, so many conflicting updates are easily resolved since they make updates on different entries. For example, the gen_key() operation generates a cryptographic key on the user's behalf, and subsequent requests only read that key from the key-value store. Also, as long as the data is available on the machines of that user, it is not imperative that all machines immediately view updates from all users.

All gets on the key-value store read a local copy of the data, and all puts and deletes write to the local copy. Updates are not propagated to other hosts immediately, so updates may occur concurrently among replicas, leaving the key-value stores in inconsistent states. The CRDT guarantees that all replicas' key-value stores' converge to the same contents after no new updates.

We use state-based replication, where updates propagate the entire key-value store state. Replicas merge this state into their local state using a merge function that reconciles any conflicts found in the two states. We explain the motivation behind this choice in § 4.3.3, after introducing the key-value store CRDT design.

4.3.1 Key-Value Store CRDT

A key-value store (or map) is a composition of a set data structure, where each entry in the set maintains a register data structure. The set manages entries in the key-value store, while each register manages the value of an entry. The set API includes lookup(), add() and remove(), where only the last two operations mutate the data structure. Similarly, the register data structure maintains a value() and assign() interface, where only the assign() modifies the register by overwriting it. Joining the two data structures together, the key-value store CRDT exposes a get(), put() and delete() API. A put(key, value) adds key into the set if it does not already exist, and assigns value to the associated register. A delete(key, value) removes key from the set if it exists and frees any memory associated with value. Finally, a get(key) performs a lookup and returns the entry if it exists, not mutating the data structure in any way.

4.3.2 Conflict Resolution

Our key-value store CRDT must resolve conflicting updates to the data structure. Updates on the key-value store conflict if they occur concurrently and modify the same entry. A conflicting put and delete must either keep the updated entry in the key-value store or remove it entirely. Meanwhile, concurrent puts on the same entry must resolve to the same final value. The choice is not apparent in either case since concurrent updates do not commute, and the outcome depends on the order in which the updates are applied.

We define the reconciliation of concurrent updates on the set independently from concurrent updates on a register. Replicas resolve conflicting puts and deletes by adding the entry to the key-value store. That is, a put dominates all concurrent deletes; this is known as *add-wins* semantics since the final result ensures the entry is added. Replicas resolve conflicting puts on the same key by assigning all puts a timestamp and choosing the update with the highest timestamp. This is known as *last-writer-wins* semantics since the timestamps are totally ordered, so the last update (i.e., highest timestamp) will take effect.

Updates can become "lost" since they do not persist until all conflicting updates are

reconciled at each replica. The user can introduce conflicts by making updates at replicas where the latest updates have not yet arrived. It may be the case where an entry is deleted but still exists on future queries or that the value of an entry does not contain an update made at a replica since it happens to be concurrent with another update that took precedence. Policies allow users to share entries, thus further complicating how users view the state if multiple users are making updates to the same entries.

These scenarios are generally tolerable given the data in our key-value store, since most operations do not modify the entries. Operations that modify the key-value store generate cryptographic material, insert arbitrary data or update system configurations such as user accounts, machines, and policy.

When generating a new cryptographic key, SEEDS is responsible for assigning a name to the entry, and the application is responsible for using this name to access the entry. SEEDS can assign a unique name to each new entry to ensure that no concurrent updates insert the same entry. If a user issues an operation on that key on a different replica, it may fail if the replica still has not received the update from the source since the entry does not exist in its key-value store. Furthermore, no operations can update the key since the entry name is unique to the replica creating the entry, and so no conflicting updates can occur.

Applications must take precautions with operations that affect system configuration such as user accounts, machines and policies. Consider a scenario where a machine is removed by one replica but added to the group at another replica. The offending machine will remain in the group and continue to share updates, which is undesirable if we wish to revoke a machine's ability to share updates if we know it is malicious. Such scenarios are problematic since we expect consistency across replicas. Applications are responsible for not introducing such inconsistencies if they cannot tolerate the possibility of updates being overwritten.

4.3.3 How do we actually resolve conflicts?

SEEDS uses an Observed Remove Set (OR Set)[41] design to implement *add-wins* semantics for the set CRDT. The idea is to uniquely tag each put and store the tag alongside the entry. Tags accumulate as concurrent puts occur. A delete removes the entry and moves the tags to the tombstone set, which maintains all the puts observed by all deletes. A concurrent put will create a tag that is not visible to a concurrent delete. As a result, the entry is added to the set.

Tombstones maintain a list of all affected puts, preventing out-of-order puts from taking effect and allowing elements to be re-added since they are unique according to their tag. The tombstone set continues to grow as entries continue to be deleted. As a result, the tombstone set unboundedly grows as deletes continue to be applied. Wuu et al. [48] suggest a method of garbage-collecting tombstones already delivered to each replica. However, this approach depends on acknowledgments and introduces message delivery requirements which is undesirable given that SEEDS replicas can come online and go offline for varying periods.

We implement an Optimized OR Set (Opt-OR Set) [16] to avoid keeping unbounded tombstone metadata using version vectors. Version vectors track the causality of operations across replicas. A version vector is typically associated with a distributed object and they can be compared to determine if one object is more up-to-date or contains concurrent updates. They can be merged to reflect all versions that have occurred on the object at each replica, which is simply a pairwise maximum of each version. Each replica maintains a global version vector and a per-entry version vector that denotes the replica's local version and per-entry version, respectively.

The Opt-OR Set works best with state-based replication since state-based updates encapsulate all operations causally preceding a delete operation, which simplifies the ordering requirements of the updates. Conversely, assume each update contains a version vector and an operation, if updates arrive out of order, merging a version vector attached to the latest update will indicate that the replica has seen all previous updates associated with all consecutive versions. This is not necessarily true if updates arrive out of order and requires

that updates are delivered in the ordered they are applied at the source to ensure replicas do not miss any updates. This means, operation-based updates may require replicas to queue updates until earlier ones arrive. Since each state-based update maintains all causally related updates, these updates can be applied out of order and do not require strict guarantees from the messaging middleware.

On a delete, the replica deletes the entry and per-entry version vector from the key-value store. On each put, the local replica's version in the global version vector is incremented by one, an entry structure is created if necessary and the replica's version in the per-entry version vector is set to the replica's version in the global version vector. The global version vector indicates the number of puts at that replica. We can track which versions are associated with which entries by storing the version with the corresponding entry in the per-entry version vector. This way, replicas can compare version vectors to determine which state has a more recent update on each entry.

As mentioned earlier, we employ a state-based update scheme where each update maintains all entries in addition to a per-entry version vector and a global version vector. Merging works as follows, if the update is missing an entry that the local replica has in its key-value store, then the update either has a more recent delete, or the local replica has issued a more recent put on this entry. This situation can be resolved by comparing the version vector of the entry in question against the update's global version vector. If at least one version in the local entry's vector is larger (i.e., more up to date) than a version in the update's global version vector, it means that the local entry has a more recent put not known by the update. A similar process determines if the update has more recent puts than local deletes. Entries are only removed if the global version vector of the replica with the deleted entry dominates the entry's version vector. As a result, a concurrent put will update the entry's version vector to be concurrent with the replica's global version vector ensuring it is not removed. If an entry exists in both the local replica and the update, then the per-entry version vectors are merged. Finally, we merge global version vectors to indicate that the replica is aware of all operations in the update.

The merge operation mentioned above for the Opt-OR Set adheres to CRDT rules

since version vectors are CRDTs themselves. Specifically, merging two version vectors is commutative, idempotent, and associative. Furthermore, since the merge function is commutative and associative, updates may arrive out of order at each replica.

We extend the Optimized OR Set to associate each entry with a register CRDT to implement our key-value store. Each replica must decide which put assigns the register's value to implement *last-writer-wins* semantics. We tag each put with a timestamp by concatenating a counter and replica identifier. The counter on each update is larger than the counter currently associated with the entry to ensure that this update takes precedence over the visible update. This way, tags are unique and totally ordered, meaning all puts are ordered as well. As concurrent puts occur, replicas only apply the put with the higher tag. All replicas independently apply the same final put making this operation commutative since all replicas will converge to the same final register value. Therefore, a merge will first determine if the entry is in the set and then compare the remaining entries and update the tags and values accordingly.

Therefore, each entry contains a string denoting the name, a blob of bytes that maintain the value, a version vector, and a timestamp. The key-value store additionally maintains a global version vector. Although this adds moderate metadata per entry, we gain the ability to share updates in any order and do not require cleaning up tombstones. Furthermore, we accept the bandwidth necessary to send state-based updates since the alternative of sending operation-based updates with tombstones introduces similar problems but with the added requirement of garbage collection. Updates occur periodically and encapsulate one or more operations on the key-value store.

4.3.4 Policy Replication

Every operation only executes at the source replica if a matching policy exists in the policy entry. Different replicas may have different versions of the policy, impacting which operations are allowed and which updates are accepted. This introduces a dependency between differing entries that breaks the commutative property of updates.

Consider two replicas A and B, assume an admin on replica A modifies the policy in Table 4.3 to remove users' put permission on their email entry. At the same time, on replica B, Alice issues an operation that updates her email. Alice successfully updates her email since the policy entry at replica B permits this operation. However, after sharing updates, replica A's policy does not allow Alice's update on the email. Meanwhile, the policy update will reflect on replica B, and Alice's email entry will remain updated. This scenario leads to an inconsistent state across replicas if replica A rejects the operation in the update.

A straightforward approach to mitigate inconsistencies that arise from update ordering is to trust the updates from other replicas and apply them locally without verification against the local policy entry. Since each replica must act according to some old but valid version of the policy, all updates are correct. Furthermore, it is unclear how to isolate operations in state-based updates without maintaining a log of operations since the merge function only compares version vectors and timestamps, not the actual operations. Therefore, SEEDS only guarantees that a policy will take effect system-wide after the update eventually arrives at each node.

We still ensure eventual consistency since all replicas will accept the updates made by other replicas even if the updates conflict with local policy. Policies only prevent operations from being accepted by the source replica, once the update is accepted it is propagated to other replicas. Security critical updates must occur when all replicas have high connectivity to ensure the update is accepted system wide, and concurrent conflicting operation do not succeed.

4.4 Establishing Secure Channels

Replicas establish point-to-point connections with one another to share and receive updates. A list of SEEDS machines is available at initialization, which describes the machines that maintain replicas. Each SEEDS replica periodically checks these entries to determine which replicas it must stay in communication with. Replicas engage in a TLS handshake to

establish a secure communication channel and perform mutual attestation to ensure both endpoints are running the correct software.

Traditionally, public key infrastructure allows parties to establish a secure communication channel by verifying each other's identities. The party responding to the connection shares an x509 certificate that binds their identity to a public key. The initiating party verifies this binding by checking with a certificate authority (CA), who acts as the root of trust. The verified public key is then used to perform a key exchange to share encrypted messages over the network.

Instead, we rely on Intel SGX as the hardware root of trust rather than a CA. We link the public key to a specific instance of the SEEDS enclave and prove to the party initiating the connection that it is communicating with genuine Intel hardware [27]. This way, the initiating party can ensure that the key was generated in an enclave within a correct instance of SEEDS.

On initialization, each replica is loaded with a public-private key pair, then generates a report that describes the enclave, including the enclave's measurement and a hash of the public key. The report achieves the first goal of linking a key with a specific enclave instance. The report is then signed using an Intel hardware key to generate a quote used to verify the hardware. This data is placed into an x509 certificate (under it's own object identifier) and self-signed with the enclave generated key. This certificate is shared during the TLS handshake.

The initiating replica requests the x509 certificate of the remote party and presents its own x509 certificate to establish trust in both directions. Both parties verify the quote in the certificate by sharing it with Intel's Attestation Service (IAS), who responds by verifying whether or not the quote was produced by genuine Intel hardware. If verification is successful, both parties check the report within the quote to make sure each side's enclave is running the expected software. Upon successful verification, each party can verify the signature in the x509 certificate to ensure a secure and mutually attested communication channel is initialized and can be used to share updates.

It is important that we ensure the communicating party is authorized and belongs to

our desired group of replicas. This can be done by maintaining a list of authorized keys in each enclave and validating the keys upon connection initialization. Alternatively, we can rely on traditional PKI and rely on a CA to issue certificates to valid SEEDS replicas. This would require us to swap out the self-signed certificates for CA signed certificates, and load the replicas with these certificates at initialization.

4.5 Adding a Host

To scale the SEEDS service, privileged users add machines to the cluster by calling put() with an entry of type **machines**, which expects the IP address/domain name of the new host. This operation will add a new entry to the key-value store and share this entry with all replicas through an update. Replicas periodically check the lists of hosts in the key-value store to determine which machines are in the group. Newly added machines must be initialized with at least one valid machine to bootstrap the state. To remove a host and revoke its privilege of sharing and receiving updates, a user can issue a delete() operation to remove the entry of the of the target machine. Note that if a concurrent update adds the deleted machine to the list, the machine will not be removed and will continue to participate in updates.

4.6 Disconnected Updates

SEEDS allows replicas to make disconnected updates. Consequentially, an adversarial host may intentionally disconnect a replica from the network. As a result, the replica will continue to service requests but fail to synchronize state with other replicas, thus allowing access to data that another replica may have modified or deleted.

More importantly, the disconnected replica is permitted to make updates to the key-value store, and upon re-connection, it will share its state with other replicas. These updates can potentially overwrite any updates made and accepted by other replicas given how we reconcile conflicts using CRDTs.

Users are responsible for removing machines they believe should no longer send and receive updates. All replicas cease sending updates to the revoked machine once they receive the update that removes the machine from the key-value store. Alternatively, users can create a policy that limits the disconnected period of updates and acts as a "dead man's switch" that suspends the interaction with disconnected machines.

4.7 Persistence

SEEDS persists state to recover from a crash or shutdown by periodically saving the key-value store to disk. SEEDS leverages the serialized state in the update to accomplish this. Each time the replica prepares an update, it seals the update to disk, overwriting the previously saved update. Sealing is necessary to ensure the confidentiality and integrity of the persisted state. The sealed update is decrypted, deserialized, and loaded into the enclave as the key-value store state to reconstruct the replica after a crash or shutdown. SEEDS cannot protect against a malicious host deleting the persisted state.

Furthermore, the host may be adversarial upon restarting and present the enclave with a stale copy of the saved update. This update may contain more permissive policies and old entries that should no longer be valid. SEEDS ensures freshness of saved data by using Intel's trusted hardware counter. On each update, SEEDS increments and seals the updated counter value and the update to untrusted storage. On an unseal, the replica reads the hardware counter and ensures this value matches the value of the last update in the unsealed update. If the unsealed counter is less than the hardware counter, the host is attempting to load the enclave with old state since a sealed version must exist with a larger counter. In this scenario, the replica terminates.

There has been some debate on the usability of Intel's hardware counters due to their high overheads and the limited number of updates. ROTE [30] proposes a trusted counters solution using a set of distributed replicas and a quorum. This technique can apply to SEEDS in the event Intel discontinues support for hardware counters [9].

Chapter 5

Implementation

SEEDS is implemented in C with $\sim$ 14k LoC for FreeBSD 12.2 and relies on the FreeBSD port of the Linux SGX SDK. It runs as a background service, and standard applications connect using IPC or SGX-applications connect using local attestation. The service comprises the key-value store, state machine, and an SSL engine.

The key-value store wraps an in-memory hashtable [24] adding CRDT metadata such as version vectors and timestamps. Serialization of the key-value store relies on the C implementation of the protobuf library [22] that is statically linked against the enclave code.

A state machine in the enclave steps through authentication and authorization. Policies are checked using a regular expression engine [7] linked against the enclave. The security of the policies depends on the correctness of this engine to correctly match operations and policies, which has been formally verified using KLEE [18]. We do not require the full power of regular expressions. Instead, we can implement a subset of the functionality for policy checking purposes, reducing the memory overhead and simplifying verification.

To establish channels between replicas, we use the wolfSSL library [23] which enables TLS termination and management within the enclave. Specifically, the wolfSSL library permits extending an x509 certificate to add metadata required for remote attestation. We

Binary	Size (KB)
protobuf-c	48
tiny-regex	3
uthash	1
wolfSSL	383
Total	435

Table 5.1: Breakdown of libraries used by SEEDS.

use the Intel SGX library for cryptographic operations provided in SEEDS API and use wolfSSL to establish mutually authenticated and attested channels between replica.

Since we rely on the system software for network and IPC functionality, the untrusted code of SEEDS establishes and accepts connections and makes the enclave calls. A dedicated thread manages the sending and receiving of updates, while another thread manages IPC, however, a single lock protects the key-value store to ensure atomicity of updates.

Applications link against custom client stub code written in C to format the requests before sending them to the SEEDS service. The implementation went through several versions, including an operation-based construction which we consequently used to implement the client and server RPC code. Instead, we can use GRPC to allow for applications written in different languages to connect to SEEDS.

Table 5.1 shows a breakdown of the static libraries included in SEEDS enclave. Not surprisingly, the library with the most overheads is the one that implements the TLS protocol. Any parts not explicitly mentioned in this section, such as version vectors, CRDTs, and policy-related functionality, make up the remaining portion of the SEEDS enclave code.

Chapter 6

Applications

We implemented two applications to use SEEDS, a distributed authentication system, and a virtual U2F token. This section describes the motivation behind each application and the primary implementation details. We discuss how each application deals with conflicts and how policies are used to protect operations.

6.1 Decentralized Authentication

Computers in a network require seamless sharing of information about users, groups, and several other resources. A directory service stores these resources in a hierarchical database and defines a namespace to identify each resource to enforce access control. The Lightweight Directory Access Protocol (LDAP) defines how users can read and update the database over a network connection. Since a directory service stores users' account information, LDAP also supports authentication services by validating login credentials against the database.

Typically, a user forwards a request to an LDAP server responsible for the resources in that namespace, making each LDAP server a central point of failure. These servers are susceptible to DDoS attacks where adversaries flood the network with malicious traffic,

making LDAP and its services unusable. As a result, users cannot authenticate and gain access to their computers.

We implemented a decentralized replacement for LDAP user authentication that addresses these limitations. Specifically, we implemented system database modules that use SEEDS to ensure user account information is always available, secure, and replicated across hosts. As a result, our system does not suffer from DDoS attacks since information is readily available at each host. We only manage user account information, and do not address the other resources managed by LDAP. Each replica maintains a full copy of the user account information allowing for reliability in the face of replica crashes or faults.

Unix-like operating systems maintain system information in separate system databases (i.e., passwd, group, shadow). Each database has a unique API to retrieve entries implemented by independent modules. System administrators can load different modules to replace where and how the system looks up this information. The Name Service Switch (NSS) configuration file maps each database to a module implemented as a shared object. We implemented an NSS module for the *passwd* interface, which tells the system about user account information.

These functions do not need to be performed by an authenticated user since they provide lookup functionality to all users in the system. For example, the *getpwnam()* function expects a username and returns a *passwd* structure. This is implemented by making a call to the `get()` operation in Table 4.1 with the entry name corresponding to the user, which returns a similar structure. Notice that the password is not revealed in the structure, only information such as user ID and group ID.

NSS modules only provide lookup functionality for user accounts. To provide authentication services used by applications on the system, we implemented Pluggable Authentication Modules (PAMs). Applications call PAM functions to dispatch the desired authentication instead of implementing it themselves. Some examples of PAM aware applications are *login* and *su*. PAM offers four module types: account, authentication, password, and session management, each offering a unique set of authentication services. We implemented the authentication module responsible for requesting and verifying the user's credentials

and the password module that allows users to change their account passwords.

For this application, SEEDS is initialized with several users and policies similar to those in Table 4.3, and the system entries are similar to those in Table 4.2. SEEDS runs in the background, and each time an NSS or PAM function is called, the system consults the respective configuration file and dispatches a custom module. The module is simply a program that implements the functionality of the PAM or NSS function and connects with SEEDS, issues a request providing the necessary information, and terminates immediately upon completion. The modules can be implemented as an SGX application or an untrusted application. Therefore, the connection to SEEDS can be through a locally attested communication channel between the module and SEEDS enclaves, or through IPC, respectively. The decision to use either depends on the underlying threat level of the host.

To reiterate, we do not protect against a host that can read or modify the requests passed to and from SEEDS. We expect the system software to be trusted in this case for the host to correctly load our module. We do not implement any mechanisms for secure IO in our module implementations. For example, an application such as *login*, makes a call to the *pam_sm_authenticate()* function which prompts the module to collect a username and password from the keyboard input and sends the collected data to SEEDS, which validates the user against the local key-value store. Other existing TEE technologies support secure UI [3, 34], using such technology is a potential direction for future work.

Since users can update their authentication information using the *pam_chauthtok()*, a user can make an update at one machine and then go to another machine that has not yet received the update. In this scenario, the user would need to use old information to authenticate since the most recent information is not yet available on the second replica. We imagine that users do not frequently change machines, otherwise, replicas must regularly connect to ensure shared updates. Issues with concurrent policy and machine updates follow a similar requirement.

To test our implementation, we developed a simple PAM-aware application that authenticates a user using the *pam_authenticate()* function, and we compare the execution

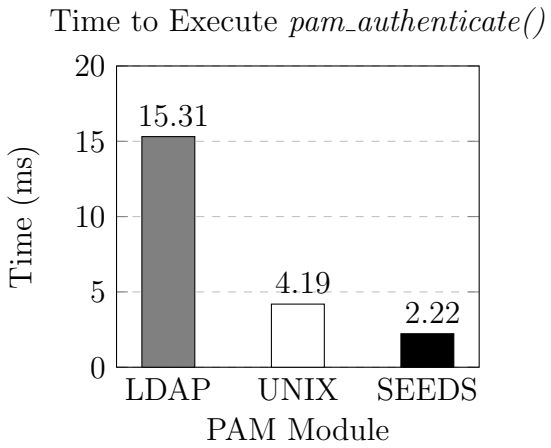

Figure 6.1: Time to execute PAM authentication function using different modules using an average of three runs.

time of different modules in Figure 6.1. Since the function uses a callback to gather the username and password from standard input, we rely on an expect script to submit the credentials. We initialized each database with 52 users, where our test user was the last in the list. The first module we tested was the LDAP module (pam_ldap.so), which requires a network connection to verify the credentials. For this reason, the LDAP module takes approximately 3.6 × longer to authenticate a user than a local method. Next, we tested the UNIX module (pam_unix.so), also known as the system database module, which reads the system flat-file databases (/etc/passwd and /etc/shadow) to verify credentials. Finally, we tested our implementation where our module connects to the SEEDS service through IPC. The UNIX module reads the databases into memory, meanwhile, our implementation has the account entries loaded in enclave memory. We believe this to be the reason for the 1.88 × slowdown in the UNIX module compared to the SEEDS module.

6.2 Software U2F Token

Over recent years there has been a rise in the use of second-factor authentication (2FA) protocols to authenticate users and achieve a higher degree of certainty that the user is whom they claim to be. The Universal 2nd Factor (U2F) protocol offers 2FA through a peripheral device that creates and stores cryptographic keys and computes digital signatures. A popular choice is a USB token, also known as a U2F token. This device plugs into a computer and communicates with a service, usually through a web browser, without the need for additional client software. The security lies in the cryptographic secrets provided by the token and the physical possession similar to a smartphone or house keys.

If users lose their U2F token, they must authenticate to the service using an alternative mechanism (i.e., email or SMS) and delete the lost token from the account. Many sites support multiple tokens to be registered with a single account [11] to avoid falling back on less secure authentication mechanisms. As a result, many users are burdened with possessing and managing multiple physical tokens to avoid these complications. Furthermore, there have been known vulnerabilities in cryptographic hardware, specifically relating to bugs in the cryptographic library used such that an attacker could learn the private RSA key from just possessing the public counterpart [33].

To address these concerns, we emulated a U2F token entirely in software that uses SEEDS to generate, store and operate on keys. We allow token material to be replicated across users' devices for high availability, this way, users do not need to worry about losing their authentication material along with physical hardware. In addition, users can remove a machine from the group, ensuring stolen devices do not continue to have access to cryptographic material. Like the U2F hardware token, the data in our software implementation cannot be read or modified by system software, meaning authentication material remains secure and only accessible to the owner. Finally, any bugs in the code can be updated but requires that each replica receive a new report of the code to remotely attest the new enclave.

To implement the token in software, we emulate a HID device using CUSE. Instead of relying on communication between SEEDS and a separate process, we implement the

token as an SGX application and link the SEEDS code statically. Therefore, the token must perform the SEEDS initialization process before use. For this application, it is not necessary to initialize SEEDS with multiple user accounts since a physical token typically holds authentication material for a single user. However, it is possible to allow multiple users to use our software token to manage authentication material. Policies for the U2F token are relatively simple and mainly describe system configuration permissions since we use a single-user model.

Hardware tokens are initialized with a per-model key pair [1] and x509 certificate. The devices generates a new key pair per service, and signs the newly generated service public key with the private key burned into the device [39]. This certificate authenticates the device model to the service through standard PKI, and allows the signed public key to be verified. The FIDO specification does not enforce that a service verifies this certificate since they assume that the client's device is trusted. Furthermore, the communication between the browser and the service is authenticated with TLS, making it difficult for an adversary to interfere with messages exchanged between the token and service.

Suppose the service wishes to authenticate the token. In this case, FIDO allows self-signed certificates that contain an identifier that corresponds to a record in the FIDO Metadata Service that can help the service identify the genuineness of a third-party token [39]. We generate a key pair and a self-signed certificate to initialize our software token, but we do not register it with the FIDO Metadata Service.

A U2F token supports two operations: registration and authentication. The former registers the token with a service, and the latter proves its identity to the service. Both operations use a challenge-response protocol between three entities: the device, the browser, and the service, where the browser acts as a middle man to relay messages. Alternatively, the browser can be replaced by any software that communicates with the token on behalf of the service.

On registration, the service sends a challenge to the web browser who forwards the messages to the device. The device mints a new key pair, generates a key handle to

[1] Observe that a per-token key pair would allow an adversary to identify and track users.

identify the key pair and saves this data along with a counter. With this data, the token generates a response message and submits it to the web browser who forwards it to the service. Assuming the response is correct, the service will register the provided public key with the user.

Authentication follows similar steps. The service sends a challenge and expects a response message that is signed by the private key associated with the public key registered with the service. The response message also contains a counter that is incremented prior to each authentication operation. The goal of the counter is to protect the user against a cloned U2F token. If the service receives a counter lower than expected, it refuses authentication and notifies the user, who should delete the potentially compromised key.

An issue with our decentralized software token is the reconciliation of counter updates. Assuming a counter is stored in a separate entry in the key-value store. When reconciling counter entries, we expect the highest counter to be taken since presenting an old counter will alert the service. With last-writer-wins semantics, we may not choose the highest counter since we compare timestamps, not counters.

To ensure counters are correctly synchronized, SEEDS offers a counter data type that can be initialized according to the entry's name. These entries store a counter CRDT instead of a register CRDT. The correct merge logic is dispatched according to the entry's type during a merge. SEEDS exposes additional API functions to create and update a counter entry, which is omitted from Table 4.1 for brevity.

This solves counter reconciliation but does not address issues that arise if replicas do not have the most up-to-date counter value. For example, a user can authenticate with a service on one replica which updates the counter and then at a later time authenticate with the same service from a different replica with a stale counter. Since the service maintains the last seen counter, it will suspect that it is replicated (since the U2F protocol does not support token replication) and refuse service to the replica with stale but valid data. To this end, we expect clients to ensure replicas regularly share updates.

Compared to the hardware token, our solution allows 2FA material to be replicated across users' machines. This way, they do not need to carry around a physical token and

worry about losing it since we gain reliability from replication. Furthermore, policies allow users to add new machines and delete old machines from accessing new keys. If a user removes a machine, that machine will still contain keys that are registered with services; it is the user's responsibility to re-register new keys so that the disconnected device cannot authenticate as the user. In terms of security, the keys never leave the enclave, which matches the guarantees of the hardware token.

Chapter 7

Evaluation

In this section, we first evaluate the performance of SEEDS. Next, we discuss how SEEDS addresses the security goals outlined in § 3.

7.1 Performance Evaluation

We benchmark SEEDS by considering the time spent checking policies and time taken to reconcile state between two replicas. We do not include SGX overheads (i.e., switch to enclave) in our measurements since the process of authentication, policy checking and merging are strictly preformed in the enclave. Furthermore, all benchmarks are reported as an average of three runs, and are single threaded since we require exclusion when updating the replica's global version vector.

First, we measure the time spent checking policies to determine the impact on each operation. Figure 7.1 lists the time taken to match an operation against a policy entry with a varying number of policies. We measure the worst-case scenario, that is, the operation matches the very last policy in the policy file. Since policies are scanned in order, the time to match an operating with each policy grows linearly with the total number of policies. We see that when we check less than 16 policies, the time taken to generate the operation

policy and read the policy entry dominates the actual policy checking since the time does not respect the linear trend seen for points greater than or equal to 16 policies.

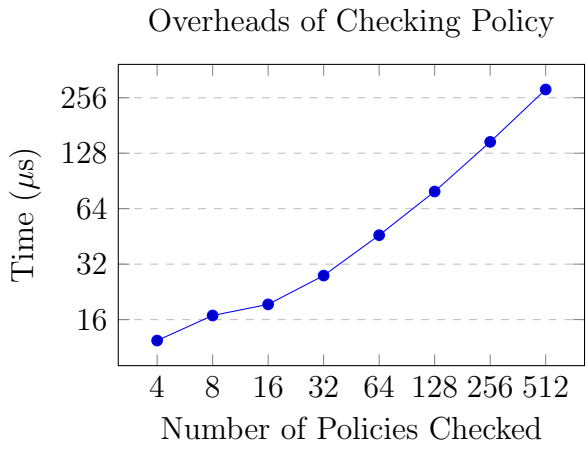

Figure 7.1: Time spent scanning through policy list with varying number of policies.

Finally, we measure the time it takes to merge two key-value stores using a key and value size of 16 bytes each. Merging occurs each time a replica receives an update from another replica. On a merge, all keys from the update must be merged into the local replica's state, and any local entries may be deleted. There are two types of conflicts: a set and register conflict. A set conflict means that one replica has an entry that the other does not have, while a register conflict means both replicas have the duplicate entry, and the final value must be resolved.

We benchmark the merge operation on two replicas with the same number of entries but differing conflicts. Merging identical replicas is the fastest since no new entries are created and no values are over-written. However, merging two replicas where each key in both sets has a register conflict is on average 1.22× slower than a merge with no conflicts. Meanwhile, merging two replicas where each entry in the update must be added to the local key-value store is 1.33× slower than a merge with no conflicts and only 1.11× slower than

a merge with only register conflicts. This is because a register conflict needs to update the value and some metadata, while a set conflict must create the entry structure and add it to the key-value store, requiring expensive operations. A merge with half conflicts of each type performs similarly when all conflicts are those of the set type since these dominate the register conflicts.

The time for replicas to converge depends on the time it takes to share updates over the network and apply the updates locally. Traditional eventually consistent systems, such as [42], require replicas to undo and redo updates according to an ordering decided by a coordinator, increasing the time it takes for the system to converge. CRDTs allow each replica to apply updates and resolve conflicting updates locally, therefore sidestepping such overheads.

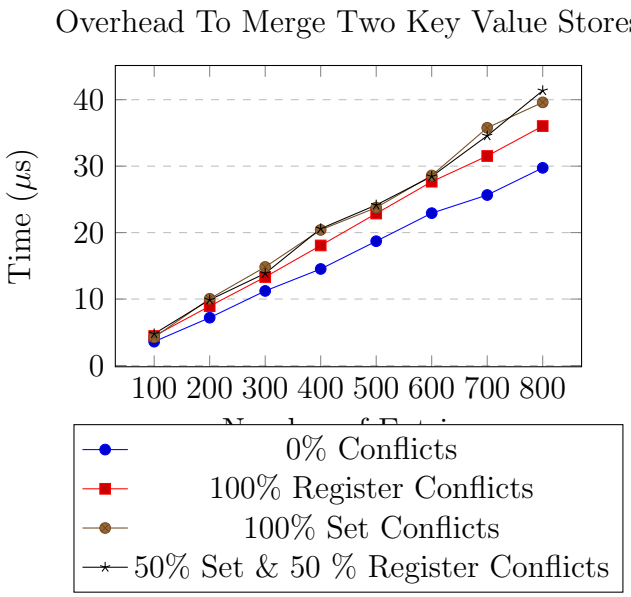

Figure 7.2: Time spent preforming a merge, where both replicas have the same number of entries and there is a conflict on each entry. The type of conflict varies (i.e., a set conflict or a register conflict).

7.2 Security Evaluation

We compare SEEDS against related solutions and discuss how the systems address the security goals outlined in § 3. Access control allows multiple users to access system resources and restricts which users can access what data. TEE key-value store solutions do not offer access control and operate as a single-user data store. For the solutions that offer access control and allow multiple users to access the services, a compromised user should not exfiltrate the user's data from the service. Otherwise, the user will need to change any information (i.e., cryptographic keys) since the attacker may clone the data. The Cloud Key Store (CKS) does not provide any functionality that explicitly reads cryptographic keys. Meanwhile, cTPM and SEEDS require that policies are in place to avoid exfiltration.

Additionally, SEEDS policies protect the operations that can be carried out on specific entries on behalf of the users. If a user becomes compromised, the policy protects what operations can be carried out on specific entries on behalf of that user. Unfortunately, if the impersonated user is privileged, they can change the policies to allow reading and writing to any data in the system.

Next, we consider a compromised host. TEE solutions ensure confidentiality and integrity of program execution and data by running the system componeWe compare SEEDS against related solutions and discuss how the systems address the security goals outlined in § 3. Access control allows multiple users to access system resources and restricts which users can access what data. TEE key-value store solutions do not offer access control and operate as a single-user data store. For the solutions that offer access control and allow multiple users to access the services, a compromised user should not exfiltrate the user's data from the service. Otherwise, the user will need to change any information (i.e., cryptographic keys) since the attacker may clone the data. The Cloud Key Store (CKS) does not provide any functionality that explicitly reads cryptographic keys. Meanwhile, cTPM and SEEDS require that policies are in place to avoid exfiltration.

Additionally, SEEDS policies protect the operations that can be carried out on specific entries on behalf of the users. If a user becomes compromised, the policy limits what operations can occur on the entries on behalf of that user. Unfortunately, if the impersonated

user is privileged, they can change the policies to allow reading and writing to any data in the system.

Next, we consider a compromised host. TEE solutions ensure confidentiality and integrity of program execution and data by running the system components in an enclave. Furthermore, memory dumps and DMA operations are not permitted by TEE technology and are protected from such attacks.

TEE solutions that execute in the cloud require the user to establish a secure connection to the service, thus allowing the data to be safely marshaled from the user to the cloud TEE. These systems adopt the traditional TEE threat model where the cloud provider may be untrusted while the user's machine remains completely trusted.

SEEDS differs from other TEE solutions since it runs on the user's machine and expects all data marshaled to the enclave to be securely handled by the application. Therefore, SEEDS does not protect against an untrusted host eavesdropping or manipulating I/O. Applications are responsible for securely collecting and passing data to SEEDS according to their desired level of protection. Applications that are worried about the host being compromised and using this attack vector can implement their program using Intel SGX to securely marshal data to the SEEDS enclave. Otherwise, applications that do not assume the host is adversarial can use our system to protect against other applications while ensuring their data is highly available across machines. Furthermore, upon initialization, any data marshaled into the enclave must be done on a trusted machine. We believe that this is a reasonable assumption, given that users would not use inherently compromised machines.

Only SEEDS remains available during a DDoS attack since all requests are handled locally, and connectivity is only required to share updates. Therefore, such an attack only impacts the time to share state and become consistent. The distributed TEE key-value store only protects against DDoS attacks if a quorum of machines remains available to approve the update and will fail to accept an update if a quorum cannot be met. Meanwhile, CKS and cTPM rely on asynchronous commands that communicate with the cloud to access resources stored in the cloud and will fail if the single machine is not

accessible.

	Single Node TEE KVSs [26, 14]	Distributed TEE KVS [13]	Cloud Key Store (CKS) [29]	Cloud TPM (cTPM) [19]	SEEDS
Security					
Multi-User Access Control	○	○	●	●	●
DDoS	-	○	○	○	●
Compromised User	-	-	●	○	●
Compromised Host	●	●	●	◐	◐
Functionality					
Replication	○	●	◐	◐	●
Disconnected Operations	-	○	○	◐	●
Arbitrary Data	●	●	○	○	●

Table 7.1: Comparison of related approaches.

Chapter 8

Related Work

Secure key storage. The Cloud Key Store (CKS) [29] generates, manages, and stores users' personal cryptographic keys in an enclave in an untrusted cloud environment TEEs for quick and reliable access. SEEDS provides similar services but maintains this data directly on user machines. This way, users do not need to maintain a network connection to the cloud to access their credentials. CKS provides a highly available service since all user devices can establish this connection to the cloud without running into compatibility issues. Instead, SEEDS maintains high availability by replicating data across user's machines, allowing applications on the same host to use the data. CKS does not consider replication in the design and acknowledges this as a possible extension to the system. SEEDS is replicated on many machines and uses CRDTs to scale the system without worrying about agreement overheads. CKS also provides access control to specify the number of key uses and expiration while delegating keys to other CKS users. SEEDS also offers access control and key delegation and allows finer-grained permissions on which operations can be performed on specific data.

Policy-based storage systems. Policy-based storage systems such as Guardat [45], and Pesos [28] protect the confidentiality and integrity of files by allowing users to specify per file policies concerning the read, write and delete permissions. These permissions encompass all the possible operations on files. SEEDS provides a more detailed API since

the data can be one of many types (e.g., a key, account information), each with its own operations. To this end, SEEDS policies better restrict what operations can be used on specific data. For example, a user may wish to allow another user to verify a signature using a specific key but not read the key. In this scenario, the user may assign a verify permission on the entry and nothing else. In all these systems, policies reduce the damage of an attack by ensuring that an impersonated user can only access data with respect to their policies.

To replicate files and associated policies, Guardat users issue commands that specify the Guardat device to create a replica. Meanwhile, Pesos replicates files and associated policies across persistent storage to improve reliability. SEEDS also replicates data together with policies but does so transparently without user involvement. Furthermore, Guardat and Pesos are remote and centralized solutions, meaning the user must maintain a network connection to make requests. If the node goes down, the service is unusable. SEEDS does not suffer from these issues since it is a decentralized system.

SGX key-value stores. Several works [14, 26, 13, 31] implement a key-value store using Intel SGX to improve confidentiality and integrity of data stored with the system. SEEDS differs from these solutions as it is not a general-purpose key-value store since all data is stored within enclave memory, severely restricting the size and number of entries. Furthermore, these secure key-value stores rely on the secure remote computation model, where users must establish a secure network connection to the system's enclave located in untrusted cloud storage. Instead, SEEDS integrates with applications on the same machine, not relying on the network as part of the service. Similarly, SEEDS and these systems rely on untrusted storage to persist key-value store data, which requires counters to ensure freshness across restarts and crashes. Avocado [13] is a distributed key-value store that offers strong consistency guarantees and implements a custom network stack to improve throughput. SEEDS relaxes consistency to provide a highly-available service with disconnected operations.

TPMs. Tian et al. [43] propose a system to run applications relying on SGX services inside Linux containers in the cloud. Unfortunately, these applications rely on common

cryptographic functionalities which cannot be shared due to EPC restrictions, thus increasing memory overheads. To address this, the authors implement a software TPM (*tpmsgx*) to reduce code redundancy. The software TPM runs as a daemon in the cloud and provides cryptographic operations, RNG, and secure storage. SEEDS provides a similar service but improves the standard TPM functionality to provide multi-user access control and improved reliability through replication. Furthermore, SEEDS runs on client machines and devices rather than in cloud environments.

cTPM [19] extends TPMs with an additional key to offload storage to the cloud and provide cross-device functionality. cTPM is limited to a single user and replicates across devices owned by the same user. SEEDS allows multi-user access and replication on machines shared by multiple users. Furthermore, TPMs do not provide remote attestation; therefore, cTPM relies on a one-time key-migration scheme to share a secret key from one TPM to another, which must occur on a trusted host. This key is used to encrypt the TPM data and store it in the cloud for access by all other TPMs that share the same key. SEEDS faces a similar issue to secure I/O and thus requires secure initialization on trusted machines. The authors assume that a device that is disconnected for an extended period means that the device is no longer in use and does not protect against this scenario. Similarly, we expect the user to issue an operation that removes a device that has been disconnected for too long and should no longer be apart of the group.

Unified Access Management. Identity access management (IAM) is a catch-all term for technologies that identify, authenticate and authorize users. Unified access management (UAM) is an extension of IAM that allows services (such as websites) to share IAM functionality. This way, services can share user account information and reduce the number of IAM instances.

UAM is a centralized solution where clients register with an identity provider (IDP), and many different service providers communicate with the IDP for access management. OAuth is an example of a popular UAM protocol. However, OAuth has several known vulnerabilities, such as man-in-the-middle, phishing, and replay attacks.

Wu et al. [47] propose SGX-UAM, a protocol that uses Intel SGX and One Time Pass-

words (OTPs) to offload the authentication process from the IDP to the client machine. SGX-UAM stores all client account information on the IDP, each time the client needs credentials from the IDP, they must authenticate to the IDP to retrieve the credentials. Each authentication requires a connection to a centralized IDP over the network. Conversely, SEEDS acts as an IDP for co-resident applications, so authentication occurs offline and relies only on the replica's local copy.

Upon registering a new login credential with the IDP, the client must input their password, which is marshaled into the enclave. Hostile system software such as a keylogger poses a threat to input security. The authors implement "shuffle mapping" to substitute the user's keyboard input in the system message queue before a keylogger can access them. This process is periodically re-installed to ensure the substitution executes ahead of the keylogger. This technique can be applied by applications using SEEDS to secure password-based authentication.

Chapter 9

Conclusion

This thesis presents SEEDS, a secure decentralized multi-user service that manages user accounts, generates and stores keys, and performs cryptographic operations. SEEDS is different from other solutions because it secures system components using TEE technology on user machines. Additionally, SEEDS replicates data seamlessly to improve the availabilitiy of system services while gracefully handling offline operations.

References

[1] Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed: 2021-12-15.

[2] Code sample: Intel® software guard extensions remote attestation end-to-end example. https://www.intel.com/content/www/us/en/developer/articles/code-sample/software-guard-extensions-remote-attestation-end-to-end-example.html. Accessed: 2022-02-02.

[3] Globalplatform technology tee system architecture version 1.2. https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE_SystemArch_v1.2_PublicRelease.pdf. Accessed: 2022-02-02.

[4] Intel sgx support for freebsd. https://www.google.com/search?channel=trow5&client=firefox-b-d&q=freebsd+sgx. Accessed: 2021-12-15.

[5] Intel software guard extensions. https://www.intel.ca/content/www/ca/en/architecture-and-technology/software-guard-extensions.html. Accessed: 2021-12-15.

[6] Password managers: Under the hood of secrets management. https://www.ise.io/casestudies/password-manager-hacking/. Accessed: 2021-12-05.

[7] tiny-regex-c. https://github.com/kokke/tiny-regex-c. Accessed: 2021-12-20.

[8] 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.

[9] Unable to find alternatives to monotonic counter api. https://www.intel.ca/content/www/ca/en/support/articles/000057968/software/intel-security-products.html. Accessed: 2021-12-15.

[10] Simjacker vulnerability, 2019.

[11] How to register your spare key. https://support.yubico.com/hc/en-us/articles/360021919459-How-to-register-your-spare-key-, 2021. Accessed: 2021-12-15.

[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703, 2016.

[13] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Vijay Nagarajan, Pramod Bhatotia, et al. Avocado: A secure in-memory distributed storage system. In *2021 {USENIX} Annual Technical Conference ({USENIX} {ATC} 21)*, pages 65–79. USENIX Association, 2021.

[14] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.

[15] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 126–140. Springer, 2014.

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

[17] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, August 2018. USENIX Association.

[18] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

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

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

[21] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[22] Robert Edmonds. protobuf-c. https://github.com/protobuf-c/protobuf-c. Accessed: 2021-12-20.

[23] David Garske. wolfssl. https://github.com/wolfSSL/wolfssl. Accessed: 2021-12-20.

[24] Troy Hanson. uthash. https://github.com/troydhanson/uthash. Accessed: 2021-12-20.

[25] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System*

Software for Trusted Execution, SysTEX'17, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

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

[28] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–17, 2018.

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

[30] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. {ROTE}: Rollback protection for trusted execution. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1289–1306, 2017.

[31] Ines Messadi, Shivananda Neumann, Nico Weichbrodt, Lennart Almstedt, Mohammad Mahhouk, and Rüdiger Kapitza. Precursor: A fast, client-centric and trusted key-value store using rdma and intel sgx. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 1–13, New York, NY, USA, 2021. Association for Computing Machinery.

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

[33] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used rsa moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1648, 2017.

[34] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

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

[36] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1481–1498, 2020.

[37] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.

[38] Thomas Porin. Bearssl. https://www.bearssl.org/git/BearSSL. Accessed: 2021-12-20.

[39] Adam Powers. Fido technotes: The truth about attestation. https://fidoalliance.org/fido-technotes-the-truth-about-attestation/?fbclid=IwAR2XP6eefkNX_7ajw9lsIk5wDLWPrdsXPY5C1vl62JS32xtfWdd2oDK87Bg. Accessed: 2021-12-15.

[40] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM:

A Software-Only implementation of a TPM chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, Austin, TX, August 2016. USENIX Association.

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

[42] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.

[43] Dave (Jing) Tian, Joseph I. Choi, Grant Hernandez, Patrick Traynor, and Kevin R. B. Butler. *A Practical Intel SGX Setting for Linux Containers in the Cloud*, page 255–266. Association for Computing Machinery, New York, NY, USA, 2019.

[44] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017.

[45] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[46] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Liangshun Wu, H. J. Cai, and Han Li. Sgx-uam: A secure unified access management scheme with one time passwords via intel sgx. *IEEE Access*, 9:38029–38042, 2021.

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