open science

TruShare: Confidential Key-Value Store for Untrusted Environments

Aghiles Ait Messaoud, Sonia Ben Mokhtar, Anthony Simonet-Boulogne

To cite this version:

Aghiles Ait Messaoud, Sonia Ben Mokhtar, Anthony Simonet-Boulogne. TruShare: Confidential Key-Value Store for Untrusted Environments. 20th European Dependable Computing Conference (EDCC '25), Apr 2025, Lisbon, Portugal. hal-04935076

HAL Id: hal-04935076
https://hal.science/hal-04935076v1

Submitted on 10 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

TruShare: Confidential Key-Value Store for Untrusted Environments

Aghiles Ait Messaoud
INSA Lyon, LIRIS, iExec Blockchain Tech
Lyon, France
aghiles.ait-messaoud@insa-lyon.fr

Sonia Ben Mokhtar
LIRIS, CNRS
Lyon, France
sonia.ben-mokhtar@cnrs.fr

Anthony Simonet-Boulogne
iExec Blockchain Tech
Lyon, France
anthony.simonet-boulogne@iex.ec

***Abstract*—Key-Value Stores (KVSs), commonly used for storing sensitive data, face significant security challenges when deployed in untrusted cloud environments. These environments are susceptible to various types of attacks exploiting a compromised OS or running a side-channel attacks. To protect sensitive data from these types of attacks distributed TEE-based KVSs have been proposed. However, these solutions are still vulnerable to side channel attacks that may compromise any node and leak all its data at once. Active defense mechanisms against side channel attacks, such as Oblivious RAM, are impractical to deploy due to their significant performance overhead or the requirement for additional hardware (*e.g.*, FPGA). Consequently, these defense mechanisms are often skipped in favor of performance in most existing TEE-based KVSs, which weakens their security. To address this issue, we present TruShare, a practical distributed in-memory KVS that integrates TEEs (Intel SGX) and Shamir Secret Sharing (SS) to provide security against high-privileged spywares while tolerating side-channel attacks on a fraction of storage nodes, without requiring costly active defense mechanisms. We implemented TruShare and evaluated its performance using 25 Microsoft Azure VMs. Compared to its closest competitors, TruShare considers a stronger threat model while providing more practical performance than solutions relying on active defense mechanisms against side-channel attacks.**

***Index Terms*—Key-Value-Store, Trusted Execution Environment, SGX, Secret sharing**

I. Introduction

Key-Value Stores (KVSs), such as Redis [1], DynamoDB [2], and RocksDB [3], have experienced a surge in popularity in the age of cloud computing since 2010. Initially designed to provide a high-speed and scalable data storage solution for unstructured or semi-structured data, KVSs have evolved to handle sensitive information, including session data [2], cryptographic keys [4], and wallet data [5]. While sensitive data is often stored in private infrastructures, there are situations where it may need to be outsourced to external public cloud-based KVSs to remain accessible and ready for use by authorized requesters. This use case is common across various industries, including Key Management Systems [4].

However, outsourcing confidential data to cloud-based KVSs raises significant security concerns [6]. In this paper we aim at considering a particularly strong threat model where all machines of a cloud environment could be subject to a spyware attack, *i.e.*, a software that exploits the high privileges of the operating system (OS) to leak data (*e.g.*, FinSpy [7]). Furthermore, we assume that up to half of the machines could be exposed to side-channel leakage attacks (SCAs) [8] where attackers can extract sensitive information by analyzing indirect signals (*e.g.*, timing, power consumption, memory patterns), even though these attacks typically have high prerequisites.

There exist many secure KVS solutions to protect private values in untrusted environments. These can be classified in three categories: (i) software-based encrypted databases (*e.g.*, [9], [10]); (ii) secret sharing-based solutions (*e.g.*, [11], [12], [13], [14], [15]), where secrets are split into n shares stored in different machines requiring a threshold of t shares to reconstruct a secret and (iii) solutions relying Trusted-Execution Environments (*e.g.*, [16], [17]). However, the two first categories of solutions are insufficient under our target threat model as a spyware attack running on all machines can leak decryption keys in the first family of solutions and can recover all the necessary shares to reconstruct the values in the second family of approaches. Hence, the most appropriate KVSs are those relying on Trusted Execution Environments (TEEs) [18]. TEEs are widely regarded as a hardware-based solution for creating secure environments that are isolated from the system's software stack, including the OS and other privileged software (including spywares). TEEs have various implementations by vendors (*e.g.*, Intel SGX [19], ARM TrustZone [20], and AMD SEV [21]), with Intel SGX being the most popular and widely used due to its extensive ecosystem (supported by Intel CPUs and Microsoft Azure) and Intel's continuous support. As such, existing SGX-based KVSs like Avocado [16] and Treaty [17] mitigate the spyware threat across all server nodes. However, bare TEEs fail to mitigate SCAs. Thus, existing SGX-based KVSs, that rely on simple replication or horizontal sharding of user data, may leak all the stored secrets if any node becomes a victim of a SCA.

Existing active defenses against SCAs can be categorized into cryptographic and non-cryptographic approaches [22]. Cryptographic methods, such as Oblivious RAM (ORAM), protect against access pattern-based attacks by accessing multiple memory locations per operation and frequently reshuffling or re-encrypting data with random seeds. However, this approach is prohibitively slow, with performance penalties as high as 83 times slower in some cases [23]. Non-cryptographic methods introduce comparatively lower overheads but typically either protect against only specific types

of SCAs or require additional hardware like FPGAs [22]. Due to these significant trade-offs in performance or reliance on extra hardware, practical SGX-based KVSs generally opt not to implement SCA defenses.

In this paper, we aim at addressing the lack of SCA tolerance in practical SGX-based KVSs by proposing TruShare, a distributed in-memory KVS that integrates SGX and Secret Sharing (SS) at its core. This combination allows both technologies to complement each other: SGX provides an enclave to protect each data share in memory across *all* storage nodes from high-privileged spyware attacks, while SS introduces a natural resilience to SCAs. Specifically, as long as fewer than t nodes are compromised, SS ensures that no data is leaked, regardless of the type of SCA. In an environment where the evolution of SCAs is unpredictable and existing mitigations are costly, our philosophy does not aim to prevent SCAs but to tolerate them, buying time while safeguarding data even if a subset of nodes is compromised. The most effective and efficient mitigation of SCAs remains patching emerging vulnerabilities through hardware or microcode updates, usually performed by the hardware provider.

TruShare goes beyond simply combining SGX and SS by incorporating several practical properties. It tolerates the unresponsiveness (*e.g.*, crashes) of up to $f(\leq \lfloor (N_{max}-1)/2 \rfloor)$ nodes and mitigates integrity threats of same fraction of nodes through client-side versioning. Additionally, TruShare supports horizontal storage scaling by adding new nodes up to N_{max}, while minimizing share redistribution using adapted Highest Random Weight (HRW) hashing algorithm [24] to save bandwidth. Last but not least, TruShare includes a node join protocol that generates or recovers missing key-share pairs when new nodes are introduced. This is accomplished through HRW and an adapted version of the Herzberg's scheme [25].

We implemented and evaluated TruShare through security comparison and practical experiments on 25 Microsoft Azure VMs. Specifically, we compare the security properties of TruShare architecture against near competitors [15], [14], [16], [17] and conclude that our system provides better confidentiality properties against leakage attacks (including SCAs). We evaluate TruShare's node throughput of *get*/*put* operations with different read/write ratios and compare it to our system without SGX. Results show a slight overhead induced by our use of SGX. We evaluate the scalability of TruShare according to different deployment settings. Finally, we performed experiments with faults to measure their impact on the latency of *get* operation and recovery.

The remaining of the paper is organized as follows. In §II and §III we present background and our system model. In §IV, we present an overview of TruShare before presenting TruShare protocol details in §V. In §VI, we compare the security properties of TruShare against its competitors and present its implementation details in §VII before assessing its performance in §VIII. Finally, we survey related work in §IX, before concluding in §X.

II. Background

In this section we introduce the necessary background used in this paper, namely: TEEs, Intel SGX, Shamir's Secret Sharing and the Highest Random Weight hashing protocol.

A. Trusted Execution Environments

TEEs are isolated, secure areas within processors designed to safeguard applications or specific application segments from other processes, including a potentially compromised Operating System. TEEs are commonly used for secure transactions, digital rights management (DRM), and protecting sensitive information like encryption keys or biometric data. Presently, TEEs are offered by major CPU vendors, including ARM TrustZone [20], Intel SGX (Software Guard eXtension) [19], and AMD SEV (Secure Encrypted Virtualization) [21]).

B. Intel SGX

Intel SGX [19] is the TEE implementation of Intel. SGX allows splitting applications into untrusted (non-sensitive) parts that do not necessitate protection and trusted (sensitive) parts that do. The latter runs within an encrypted secure memory region called an enclave, ensuring both confidentiality against non-authorized users and reliable execution. Code and data within the enclave are only decrypted inside the CPU during execution. The lastest iteration of SGX, namely SGX2 [26] expands the maximum enclave size from 128MB to 1TB but removed the built-in integrity check of enclave pages. While this improves the performance of protected applications, it opens the door to integrity threats, particularly rollback attacks which represent a significant threat in this context. Another important feature of SGX is Remote Attestation [27] (RA) that allows an entity, like a client, to verify that a remote SGX enclave is running the expected code on genuine hardware. Despite its security features and Intel regularly releasing patches to address newly discovered threats, SGX remains vulnerable to SCAs [8] that may leak data from enclaves.

C. Shamir's Secret Sharing

SSS [11] is a cryptographic method developed by Adi Shamir that enables the secure distribution of a secret into multiple shares (n) in such a way that a subset of those shares (t), known as the threshold, is required to reconstruct the original secret. The secret s is divided by a dealer using a polynomial P of degree *t-1* such that $P(0) = s$. Each participant with id i is given a share, corresponding to a point on the polynomial, $P(i)$. The ingenious aspect lies in the fact that knowledge of any number of shares less than t reveals no information about s, but once the threshold is reached, s can be reconstructed using polynomial interpolation.

D. Highest Random Weight hashing protocol

Highest Random Weight (HRW) hashing [24], also known as Rendezvous hashing, is a technique used in distributed systems for load balancing and data partitioning. In this hashing scheme, illustrated in Figure 1, each node S_i in the system is assigned a weight computed by a hash function h applied

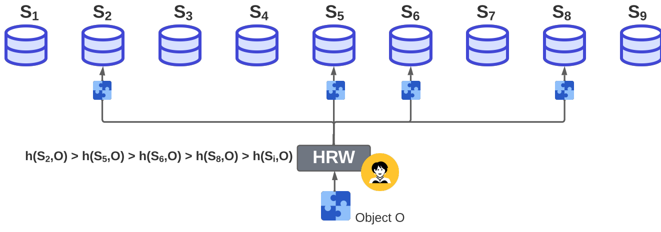

Fig. 1: HRW protocol followed by a client C to determine $K = 4$ servers among 12 servers S_i to store the object O. The top-4 nodes w.r.t to O are S_2, S_5, S_6, S_8

to both the node identifier S_i and the item being hashed O. Since we want to replicate O across K nodes, the rendezvous nodes should be the top-K nodes with the highest weights among all the nodes. This method ensures that each item is consistently mapped to the same specific servers. In the event that one of the top-K nodes is unavailable, the requester can seamlessly redirect the request to the subsequent top-K active nodes, where it can push the object O or locate it. Furthermore, HRW possesses the property of minimal disruption among hashing functions when a node S_i crashes or is spawned.

III. System Model

Let us consider a System Administrator that has a maximum quota of N_{max} nodes (servers) that could be deployed on the cloud to store confidential data. $N(\leq N_{max})$ denotes the number of already deployed nodes. Clients can interact with the storage API of the nodes by issuing *get* or *put* requests to pull or push their secrets.

System goals. TruShare seeks to provide four properties about the secrets stored in its nodes: *(i) Confidentiality*: the secrets of clients are not disclosed; *(ii) Integrity*: the secrets of clients can be correctly recovered; *(iii) Availability*: the secrets of clients are available even under failure assumptions; *(iv) Eventual consistency*: nodes will eventually converge to a state where the secrets of clients are correctly recoverable.

Data and access models. We consider that the secrets of clients are *KV* records where the key part K identifies the confidential data V that is modifiable only by the owner. We target use cases where each client is responsible for updating its own keys. This assumption implies that there is no *put* concurrency between clients about the same key. Such use case can represent Single Writer Multiple Reader databases [28]. The governance policy that authorizes or prohibits access to data to TruShare clients is a transversal work that is out of the scope of this paper.

Threat Model. Table I summarizes the adversarial capabilities of TruShare nodes where $f \leq [(N_{max} - 1)/2]$. CO, IN, and AV regroup attacks that threaten respectively the confidentiality, the integrity, and the availability of stored data. The f nodes exhibit a byzantine behaviour usually considered in BFT systems [29] including the ability to perform SCAs, whereas high-privileged spywares can affect all the nodes. We assume that clients are trustworthy and do not exhibit

		$N_{max} - f$	f
CO	High-privileged spyware	✓	✓
	Side-channel leakage attack	✗	✓
IN	Data tampering	✗	✓
	Rollback	✗	✓
AV	Crashing	✗	✓
	Unresponsiveness for user requests	✗	✓

TABLE I: Adversarial capabilities of TruShare nodes. ✓and ✗refer, respectively, to the ability or inability to carry out different attacks by two distinct proportions of TruShare nodes

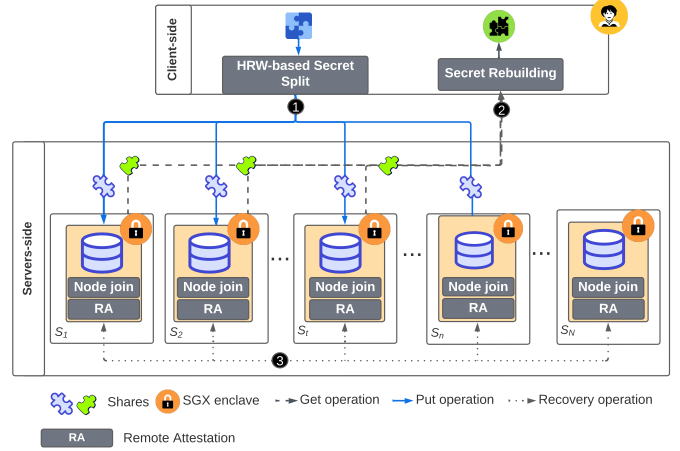

Fig. 2: TruShare overview.

malicious behaviour in their *put* operation because they do not have an interest in polluting their own data.

IV. TruShare Overview

Figure 2 illustrates the general architecture of TruShare, comprising server-side and client-side components. On the server-side, TruShare includes N nodes already deployed, each identified by a unique ID (S_i in the figure) and hosting an in-memory KVS within an SGX2 enclave. Clients perform two types of operations: (i) *put(k,v)* operation, where they divide a secret v into n ($\leq N$) shares $share_i$, using (t,n)-threshold SS, and assign each pair $(k, share_i)$ to a distinct node among N (step ❶ in Figure 2); and (ii) *get(k)* operation, where they reconstruct v from t $(< n)$ correct shares using SS (step ❷). Each session of operations is preceded by remote attestation (RA) in order to ensure that the client is communicating with genuine SGX enclaves and a secure channel establishment from client to enclave using SSL. The introduction of a new node initiates a node join protocol (step ❸), during which it constructs its KVS by interacting with existing nodes. At this step, we summarize the notations we introduced so far in Table II.

In a nutshell, TruShare relies on the following key principles: **Combining SGX with Secret Sharing**: We combine SGX2 with SS on TruShare nodes in order to inhibit high-privileged spywares threat on all the nodes and we choose the appropriate parameters for t, n and the necessary metadata to tolerate up to f byzantine nodes including SCAs (see III:System goals).

TABLE II: Notations used in TruShare

Notation	Designation
N_{max}	Maximum number of authorized nodes to deploy
$N(\leq N_{max})$	- Number of already deployed nodes - Number of nodes with high-privileged spyware
$f(\leq [\frac{N_{max}-1}{2}])$	Number of byzantine nodes among N_{max}
n	Total number of shares for secrets
t	Required shares to reconstruct secrets

Further details are given in § V-A.

HRW-based *put* and *get* operations: We define SS-specific *get* and *put* protocols based on HRW (i) to deterministically sample the *n* nodes that should hold the shares among the *N* nodes, (ii) and to save bandwidth during get operations by targeting only the appropriate nodes. Further details are given in § V-B.

HRW-based node join protocol: This protocol (depicted in each node of Figure 2) is used whenever a new server S_{new} is introduced to replace a crashed server or to extend the in-memory storage to host more secrets. Specifically, S_{new} gathers keys and generates shares that should belong to him in order to maintain the number of shares of each secret to n, through a two step protocol:

Keys discovery: S_{new} connects to all alive nodes through mutual RA and each node applies HRW to figure out the keys that S_{new} can claim. Specifically, we make sure that S_{new} obtain a key only and only if it ranks among the top-*n* servers for that key, according to the order established by HRW.

Shares recovery: We apply Herzberg scheme [25] to securely recover the shares associated with the discovered keys on S_{new}. We support the protocol with HRW to locate the domain of top-n nodes for each key.

Further details about each step are given in § V-C.

V. TruShare Detailed Description

In this section, we describe in details the TruShare key principles previously enumerated.

A. Composing SGX with Secret Sharing

By employing SGX2 across all servers, we inherently prevent high-privileged spywares from stealing shares on any of the nodes. To circumvent the remaining byzantine behaviours of f nodes (that should be fixed to any value between 0 and $[\frac{N_{max}-1}{2}]$), including SCAs, we build our solution step-by-step to mitigate different concerns.

SCAs concern: To prevent the reconstruction of a secret in case of collusion of f nodes that use SCAs, the minimum required number of shares to rebuild the secret should be set as (1) $t \geq f+1$.

Availability concern: Ensuring that the secrets remain recoverable during *get* operations in a context where f nodes may crash or not respond requires (2) $n \geq f+t$; this guarantees that t nodes are always responding.

Eventual consistency and freshness concerns: In TruShare, if a client sends two successive *put* requests for the same key, the system ensures that the nodes store the latest version of the share by using client-side versioning. The client associates a monotonic version number to each key it possesses and appends this version number to all the shares that belongs to the same secret during *put* operation. This ensures that nodes will only store key-share pairs that have a higher version number than the one currently stored, guaranteeing that the most recent data is retained.

Integrity concern: SGX2 removes integrity check from its TEE implementation, meaning that an attacker can either blindly write the memory of f nodes, or restore a previous snapshot of their memory. Because the memory remains encrypted, attackers can only inject blind faults into f nodes. To prevent this type of attacks using the version number appended to each share; both in case of blind writes and in case of replay attacks, clients performing *get* operations must consider t shares with identical version number and isolate the others as invalid. This approach is based on the assumption that an attacker will not be able to manipulate the encrypted memory precisely enough to inject an invalid share while retaining a valid version number.

From (1) and (2), the overarching rule for maintaining TruShare confidentiality and availability is $n \geq 2f+1$. It also means that at the initialization of TruShare nodes, the minimum number of nodes N should be $2f+1$ *i.e.*, $N \geq n$. The same rule allows TruShare to maintain secrets integrity because it guarantees that there is a majority of correct nodes ($n-f>f$) that holds correct shares.

B. HRW-based put and get operations

We adapt the HRW hashing algorithm to fit the sharding protocol of our SS-based KVS. Figure 3 illustrates the HRW-based *put(key, value)* protocol. First, (❶) the client applies HRW hashing to all node IDs S_i of the deployed nodes (N) alongside *key*. The resulting hashes are used to sort the node IDs in decreasing order and the top-n nodes are selected to host the shares. Subsequently, (❷) the *value* is split using (t,n)-threshold SS to generate *n* shares $share_i$ based on the top-*n* nodes. As we mentioned in V-A, the client appends the version number v of the key to the generated shares. Finally, (❸) each record $< key, share_i, v >$ is sent to a node S_i among the top-n through a secure SSL channel established after attesting that S_i is running in an enclave. The *put* is considered successful if the client receives $n-f$ positive acknowledgements at least.

The *get(key)* protocol is straightforward: Step ❶ remains unchanged. Subsequently, the client multicasts a get request to all nodes among the top-n and waits for t shares with identical versions numbers.

Our usage of HRW optimizes bandwidth usage when making *get* requests by reducing the potential number of share owners from N to n. It accomplishes this by shifting the network overhead of probing N nodes through the network to a local overhead involving hash computation and sorting.

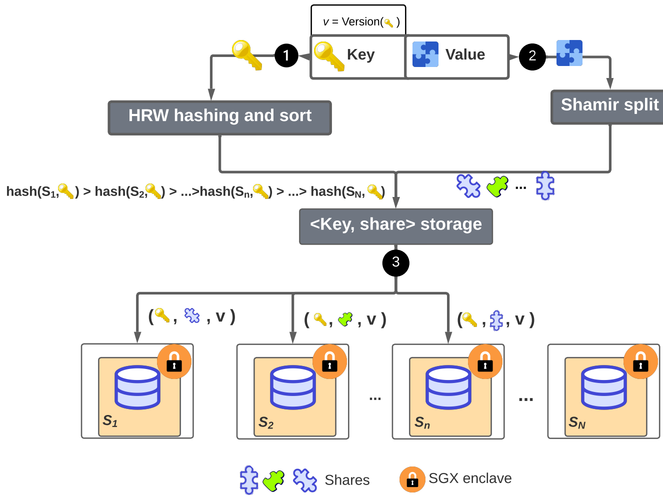

Fig. 3: HRW-based put(Key, Value) protocol.

C. HRW-based node join

In the event of a node crash, TruShare maintains functionality without requiring KV record rebalancing, thanks to the HRW protocol which preserves the node order and allows uninterrupted *put* or *get* requests. When S_{new} replaces a crashed node or expands storage, it establishes a secure channel with existing nodes, and follows a two-step protocol to recover and populate the missing KV records. To ensure successful recovery, system administrators must first spawn nodes with the same IDs as the crashed ones before considering extension, as HRW will not guarantee S_{new} ranks among the top-n for missing keys otherwise.

1) Keys discovery: The first step for a new node to recover its KVS is key discovery. The joining node broadcasts its ID, S_{new}, to all active nodes. Each active node then runs the `DISCOVER_LOST_KEYS(`S_{new}`)` procedure in Algorithm 1 which adds S_{new} to the list of nodes (line 22). Then, for each key k in its KVS, the node computes the HRW-sorted list of nodes (line 25). S_{new} receives key k only if it ranks among the top-n nodes, referred to as neighbors for key k. Additionally, S_{new} gets the ID of the new $(n+1)^{th}$ node in the HRW ranking which may hold a share of k (line 30). This information is essential for issuing a delete command to it later, to free up memory and ensure at most n shares exist for a given secret.

2) Shares recovery: Once S_{new} has discovered its keys, it employs the Herzberg scheme [25] to regenerate the shares associated with them. While Cobra [15] uses this scheme to create a broadcast domain to regenerate shares of a new node, we use it to create multicast domains, because our keys may belong to different top-n groups. We assume that the original polynomial utilized by a client to partition its secret into shares is P such that $degree(P) = t-1$, $P(0) = secret$ and $P(i) = share_i$, *i.e.*, the share of the S_i. The goal for S_{new} is to compute $P(new) = share_{new}$ without knowing or leaking $P(0)$.

Algorithm 1 Keys discovery

1: S_{new}: ID of the new node
2: S_{up}: Set of IDs of all nodes (including crashed nodes)
3: t, n: SS parameters
4: Ks: List of keys mapped to the current node
5: $h()$: Hash function used for HRW protocol
6:
7: **procedure** HRW-SORTING(S_{up}, k)
8: ▷ map each node of S_{up} with its hash w.r.t to k then return a sorted map according to the hashes
9: $nodes_to_hashes \leftarrow map < node_id, hash >$
10: **for** $S_j \in S_{up}$ **do**
11: $nodes_to_hashes[S_j] \leftarrow h(S_j, k)$
12: **end for**
13: $sort(nodes_to_hashes)$
14: **return** $nodes_to_hashes$
15: **end procedure**
16:
17: **procedure** DISCOVER_LOST_KEYS(S_{new})
18: $< key, node_id > lost_keys[\,]$
19: ▷ create a void array of $(key, node_id)$
20: ▷ key: a key that should be mapped to S_{new}
21: ▷ $node_id$: the last node among the top-n nodes for key
22: $S_{up} \leftarrow S_{up} \cup S_{new}$
23:
24: **for** $k \in Ks$ **do**
25: $nodes_to_hashes \leftarrow$ HRW-sorting(S_{up}, k)
26: $sorted_S_{up} \leftarrow nodes_to_hashes.get_keys()$
27: ▷ Get the keys part of the ordered map in an array (starting index is 0)
28: **if** $h(S_{new}, k) \geq nodes_to_hashes[sorted_S_{up}[n-1]]$ **then**
29: ▷ S_{new} is a neighbour for k
30: $lost_keys.add(< k, sorted_S_{up}[n] >)$
31: **end if**
32: **end for**
33: **return** $lost_keys$ ▷ transfer the $lost_keys$ list to S_{new}
34: **end procedure**

1. Distributed Polynomial Generation (DPG). S_{new} initiates a DPG involving all the N deployed nodes: each S_i of N generates a polynomial R_i of degree $t-1$, such that $R_i(new) = 0$, and broadcasts $R_i(j)$, *i.e.*, a share of R_i, to every other node S_j of N.

2. Shares reconstruction. For each discovered key k, S_{new} uses HRW to determine the list l_k of the top-n nodes that hold k (but excluding S_{new} and crashed nodes). l_k forms the multicast domain for k. Then, S_{new} requests each $S_i \in l_k$ to sum up the received polynomial shares $R_j(i)$, *i.e.*, to compute the polynomial share $R(i) = \sum_{j/S_j \in l_k} R_j(i)$ with $R(new) = 0$. Finally, each node $S_i \in l_k$ sends $(P+R)(i) = share_i + R(i)$ to S_{new} which possesses sufficient points to ex-

ecute polynomial interpolation and evaluate $(P + R)(new) = P(new) + R(new) = P(new)$. This process is repeated for each discovered key k.

3. Extra share deletion. As explained in V-C1, S_{new} issues a delete request of the KV-pair to the new $(n+1)^{th}$ node in the ranking to free up memory and ensure at most n shares exist for a given secret.

Ensuring consistency of the shares of R. In the case where a node S_i has not received a polynomial share from S_j, the computed share $R(i)$ may be incomplete and thus may impact the recovered share. To overcome that situation, each node S_i informs S_{new} of the list of nodes $Rlist_i$ from which it received a share R_j during DPG. In that way, S_{new} will be able to sample for each discovered key k a subset of size t or more that indicates the elements to include in the computation of R for each $S_i \in l_k$, leading to an implicit consensus.

Dealing with integrity concerns. A malicious node S_i can blindly modify either its shares of secret or one of its received polynomial shares $R_j(i)$. To overcome the former integrity violation, S_{new} applies polynomial interpolation only after receiving t shares $share_i + R(i)$ with identical version number (see V-A:Integrity concern), indicating that the correct shares have been used in the sum. In the same way, to detect a tampered polynomial share $R_j(i)$, S_{new} can choose a random code and ask the nodes S_i during DPG to generate R_i appended to the random code. Then each enclave can check, before computing its share $R(i)$, that the involved polynomial shares have all the same random code.

VI. Security comparison

Table III compares the security guarantees of TruShare against near competitors on confidentiality, integrity and availability fields. The percentages represent the level of resiliency, *i.e.*, the proportion of nodes that can be exposed to the considered threat. Methods that combine BFT [29] and SS such as [13], [14], [15] focus on the linearizability (Lin) property, *i.e.*, each operation is ordered in real-time ensuring that each read reflects the most recent write. For all types of threats, they tolearate 1/3 of compromised nodes. Distributed SGX-based KVSs such as [16], [17] completly inhibits high-privilged spwayre threat using SGX. However, they rely on simple state replication of values or horizontal sharding across nodes, meaning that they do not tolerate any side-channel leakage attack. Similarly, TruShare inhibits high-privilged spyware threat using SGX, but also allows to tolerate all other threats, including SCAs on up to half of the nodes (maximum possible value of f/N). However, TruShare trades linearizability for eventual consistency (EC), *i.e.*, nodes that store the same key will eventually converge to a consistent state with client-side versioning, but before the convergence, read operations are not guaranteed to reflect the most recent write.

VII. Implementation

We implemented TruShare using C++ and the official SGX SDK from Intel. Our implementation makes use of a modified version of fletcher's implementation of SSS [30] for the tasks of splitting a secret and combining shares. We integrate this implementation into an SGX enclave, incorporating the join protocol as an additional feature. The KVS itself is a sample C++ `std::map<string, string>` with our HRW-based get/put API interface. We utilize the asynchronous API of the official gRPC library [31] to handle communications. A preliminary version of TruShare is available here: https://github.com/aghia98/TruShare/tree/Sample_KVS_in_enclave_grpc_ss_HW_multiple_machines_recovery

	BFT+SS [13], [14] [15]	**Distributed TEE-based KVSs [16], [17]**	**TruShare**
High-privileged spyware	33%	100%	100%
Side-channel leakage	33%	0	Up to 50%
Integrity threats (data tampering or rollback)	33%	50% with SGX1	Up to 50%
Availability threats (crash or unresponsiveness)	33%	50% with SGX1	Up to 50%
Consistency model	Lin	Lin	EC

TABLE III: Security comparison between TruShare and near competitors. Values of the first four lines correspond to the proportion of tolerated faults. Lin = Linearizability, EC = Eventual Consistency

VIII. Performance Evaluation

We performed an experimental evaluation of TruShare. Our evaluation aims at answering the following questions:

Q1: *What is the throughput of TruShare servers w.r.t different number of worker threads and w.r.t different types of workloads (read=100%, 50%)?*
Q2: *What is the overhead induced by the use of SGX on a TruShare server throughput performance?*
Q3: *How does TruShare servers throughput scale when changing the parameters N and n?*
Q4: *What is the latency of get and recovery operations with different number of faults?*

We address questions **Q1** and **Q2** in § VIII-B, we respond to **Q3** in § VIII-C and we deal with **Q4** in § VIII-D.

A. Setup and Methodology

All experiments were conducted on a cluster of 25 Azure VMs. Ten VMs represented TruShare nodes, each equipped with 16 vCPUs (Intel Xeon 8370C, SGX2-enabled DC16s_v3) and 128GB of memory. The remaining 15 VMs were used to deply clients. Each of them had 32 vCPUs and 16GB of memory. All VMs ran Ubuntu Linux 22.04 LTS and were configured with C++17. Throughput was evaluated using the YCSB (Yahoo! Cloud Serving Benchmark) C++ implementation [32]. All experiments used 1KB secrets.

B. Single-node KVS

Figure 4 shows the throughput performance of a single TruShare node across two distinct types of workloads (read=50% and 100%), with varying numbers of worker threads until the CPU was saturated. This comparison includes

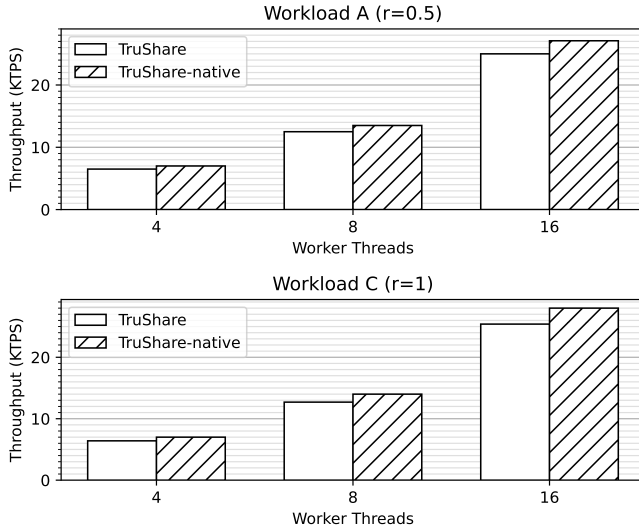

Fig. 4: Comparison of throughput (in KTPS) between TruShare and TruShare-native for different YCSB workloads and worker threads

TruShare-native to assess the overhead induced by SGX2. Regardless of the workload type, TruShare reaches approximately 25 KTPS with 16 worker threads, with a negligible advantage for read intensive scenarios. Moreover, results show that the throughput overhead induced by SGX2 enclaves is in the range of 8 to 12%. The overhead mainly comes from the cost of transferring data from in and out of the enclave. As performances are similar regardless of the workload, we keep workload A (r=0.5%) as a reference for the subsequent experiments.

C. Scalability

This section analyzes how varying the number of secret shares n and the total nodes N affects throughput, with a workload read ratio of 50%. In Figure 5, we select four values for N (4, 5, 7, 10) and vary n from 1 to 10. For each valid combination (*i.e.*, where $n \leq N$), we measure the overall throughput of TruShare. The results show that as N increases relatively to n, the throughput of TruShare improves. This is because additional nodes beyond the required n enhance parallelism, allowing more capacity to process requests.

To understand how the relationship between n and N affects throughput, we use previous experiment data to create Figure 6. For each combination, we compute the ratio n/N and record the corresponding throughput. The figure shows that as the n/N ratio increases, throughput decreases, and vice versa. This provides a clear indication of the throughput range achievable for other untested combinations.

D. Experimentation with faults

Figure 7 shows the latency of the *get* operation when $N = n = 9$ and $t = 5$ under varying numbers of faulty nodes (0 to 4). These faulty nodes represent those that may have rolled back or tampered with their shares. The total latency includes the various operations performed during the

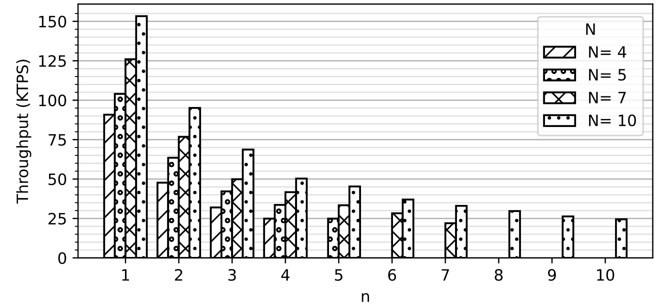

Fig. 5: TruShare's throughput with respect to different combinations of n and N

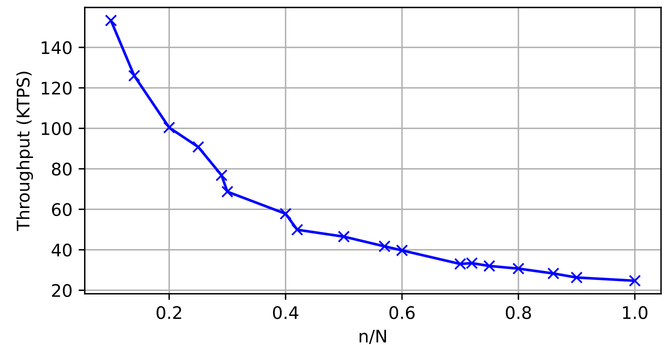

Fig. 6: TruShare's throughput w.r.t the proportion *n/N*

get operation (*i.e.*, HRW, sending the request and waiting for responses, and reconstructing the secret). The client waits to receive t correct shares (with the same version number) before performing reconstruction. The more faulty shares there are, the longer the client is likely to wait before receiving a set of t correct shares. We observe that the latencies of HRW and reconstruction are negligible (respectively $19 \mu s$ and $270 \mu s$) compared to the networking cost. Furthermore, increasing the number of faulty nodes from 0 to 4 only induces an additional $2ms$ of networking latency in average.

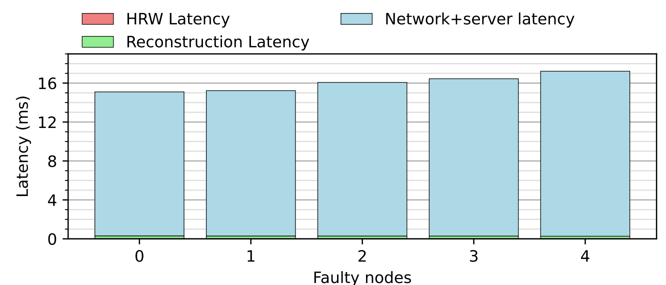

Fig. 7: Latency of *get* operation with $N = n = 9, t = 5$ and varying number of faulty nodes

Similarly, table IV presents the recovery latency of $10k$ secrets for a node when $N = n = 10$ and $t = 5$, thus considering up to 4 faulty nodes. The key discovery process demonstrates a near-constant duration of 1.2 seconds,

Faulty nodes	0	1	2	3	4
keys discovery	1.2				
shares recovery	29.72	29.79	29.84	29.92	30.09

TABLE IV: Latency of recovery protocol (keys discovery and shares recovery) in seconds for 10K secrets and $N = n = 10, t = 5$ and varying number of faulty nodes

while the shares recovery introduces a minimal overhead of $370ms$, which we estimate negligible considering the amount of reconstructed secrets ($10K$ of 1KB per share).

IX. Related Work

There exist two families of related work, that aim at bringing different security properties to KVSs or to optimize already-existing building blocks: BFT-based secret sharing (SS) and TEE-based KVSs.

BFT-based SS. Depspace [13], VSSR [14], and Cobra [15] combine BFT [29] and SS to enhance BFT's linearizability. These solutions provide confidentiality by sharding the global state using SS. However, these methods don't address confidentiality when all nodes are compromised by high-privileged spyware. Additionally, they distribute storage across all nodes ($n = N$), ignoring limited storage constraint. In contrast, TruShare leverages TEEs to mitigate high-privileged spyware on *all* nodes and uses HRW to select only a subset of nodes n from the available N for storing objects, conserving main memory.

TEE-based KVSs. Distributed TEE-based KVSs such as Avocado [16], Treaty [17] and methods that combines BFT SMR with TEE such as [33] aim to provide confidentiality guarantees in an environment where high-privileged spyware can spy on user code and data. However, they rely on entire state replication or horizontal sharding, meaning that compromising a single node with a single SCA may leak all the machine stored secrets at once. In contrast, TruShare uses (t,n)-threshold SS to split a secret into n shares across different machines and tolerate $f = t - 1$ SCAs.

X. Conclusion and Future Work

We present TruShare, a secure and confidential distributed KVS based on Intel SGX and Secret Sharing for untrusted environments. Our design includes three key contributions: (1) We combine Intel SGX and Shamir's Secret Sharing to ensure confidentiality, integrity, and availability against Byzantine nodes, high privileged spywares and side-channel attacks, considering a stronger threat model than competitors. (2) We use a deterministic approach for distributing and collecting shares via HRW, optimizing bandwidth and minimizing disruption from node crashes or spawning. (3) We introduce a node join protocol that enables new storage nodes to build their KV state while complying with HRW. Performance assessments of TruShare show its scalability potential by leveraging the ratio *n/N* and the negligible impact of faulty nodes on the latency of *get* and recovery operations. These results suggest TruShare is suitable for real-world applications.

We intend to extend this work in the following directions. First, we aim to add support for concurrent *put* operations by leveraging a tailored consensus protocol for HRW to achieve linearizability. We anticipate some performance overhead from this addition due to the consensus establishment. Thus, a linearizable-ready TruShare should be employed only for suitable applications. Another area of improvement is related to remote attestation, where we aim to decentralize the process of attesting TruShare nodes by utilizing blockchain to provide an immutable and consensual source of attestation verification.

Acknowledgments

This work was supported by a French government grant managed by the Agence Nationale de la Recherche under the France 2030 program, reference "ANR-23-PECL-0007" as well as the ANR Labcom program, reference "ANR-21-LCV1-0012".

References

[1] J. Carlson, *Redis in action*. Simon and Schuster, 2013.
[2] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
[3] Meta, "A persistent key-value store for fast storage," 2012. [Online]. Available: http://rocksdb.org/
[4] A. W. S. (AWS), "Aws kms cryptographic details," 2018. [Online]. Available: https://docs.aws.amazon.com/kms/latest/cryptographic-details/intro.html
[5] K. Karantias, "SoK: A taxonomy of cryptocurrency wallets," Cryptology ePrint Archive, Paper 2020/868, 2020. [Online]. Available: https://eprint.iacr.org/2020/868
[6] K. Ren *et al.*, "Security challenges for the public cloud," *IEEE Internet Computing*, 2012.
[7] W. R. Marczak *et al.*, "When governments hack opponents: A look at actors and technology," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.
[8] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of sgx and countermeasures: A survey," *ACM Comput. Surv.*, 2021.
[9] X. Yuan *et al.*, "Enckv: An encrypted key-value store with rich queries," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.
[10] E. Pattuk *et al.*, "Bigsecret: A secure data management framework for key-value stores," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013.
[11] A. Shamir, "How to share a secret," *Commun. ACM*, 1979.
[12] G. R. Blakley *et al.*, "Linear algebra approach to secret sharing schemes," in *Error Control, Cryptology, and Speech Compression*, 1994.
[13] A. Bessani *et al.*, "Depspace: a byzantine fault-tolerant coordination service," *SIGOPS Oper. Syst. Rev.*, 2008.
[14] S. Basu *et al.*, "Efficient verifiable secret sharing with share recovery in bft protocols," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
[15] R. Vassantlal *et al.*, "Cobra: Dynamic proactive secret sharing for confidential bft services," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.
[16] M. Bailleu *et al.*, "Avocado: A secure {In-Memory} distributed storage system," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.
[17] D. Giantsidi *et al.*, "Treaty: Secure distributed transactions," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.
[18] M. Sabt *et al.*, "Trusted execution environment: What it is, and what it is not," in *IEEE TrustCom/BigDataSE/ISPA*, 2015.
[19] V. Costan *et al.*, "Intel sgx explained," Cryptology ePrint Archive, Paper 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086
[20] S. Pinto *et al.*, "Demystifying ARM TrustZone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, 2019.

[21] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," 2016. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v9-Public.pdf

[22] H. Oh *et al.*, "Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[23] A. Ahmad *et al.*, "Obfuscuro: A commodity obfuscation engine on intel sgx," *Network and Distributed System Security Symposium*, 2019.

[24] D. Thaler *et al.*, "A name-based mapping scheme for rendezvous," *Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, Michigan*, 1996.

[25] A. Herzberg *et al.*, "Proactive secret sharing or: How to cope with perpetual leakage," in *Advances in Cryptology — CRYPT0' 95*, 1995.

[26] J. Ménétrey *et al.*, "Attestation mechanisms for trusted execution environments demystified," in *Distributed Applications and Interoperable Systems*, 2022.

[27] "Remote attestation," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html

[28] H5Py Developers, "Single-writer multiple-reader (swmr)," 2014, accessed: 2024-10-18. [Online]. Available: https://docs.h5py.org/en/stable/swmr.html

[29] M. Castro *et al.*, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[30] Fletcher, "c-sss," 2017. [Online]. Available: https://fletcher.github.io/c-sss/index.html

[31] grpc authors, "grpc – an rpc library and framework," 2015. [Online]. Available: https://github.com/grpc/grpc

[32] R. Jinglei *et al.*, "Ycsb-c," 2014. [Online]. Available: https://github.com/basicthinker/YCSB-C

[33] G. S. Veronese *et al.*, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, 2013.