Treaty: Secure Distributed Transactions

Dimitra Giantsidi
University of Edinburgh

Maurice Bailleu
University of Edinburgh

Natacha Crooks
UC Berkeley

Pramod Bhatotia
TU Munich

Abstract*—Distributed transaction processing is a fundamental building block for large-scale data management in the cloud. Given the threats of security violations in untrusted cloud environments, our work focuses on: *How to design a distributed transactional KV store that achieves high-performance serializable transactions, while providing strong security properties?

We introduce TREATY, a secure distributed transactional KV storage system that supports serializable ACID transactions while guaranteeing strong security properties: confidentiality, integrity, and freshness. TREATY leverages trusted execution environments (TEEs) to bootstrap its security properties, but *it extends the trust provided by the limited enclave (volatile) memory region within a single node to build a secure (stateful) distributed transactional KV store over the untrusted storage, network and machines*. To achieve this, TREATY embodies a secure two-phase commit protocol co-designed with a high-performance network library for TEEs. Further, TREATY ensures secure and crash-consistent persistency of committed transactions using a stabilization protocol. Our evaluation on a real hardware testbed based on the YCSB and TPC-C benchmarks shows that TREATY incurs reasonable overheads, while achieving strong security properties.

I. INTRODUCTION

Transactions (Txs) are an integral part of modern cloud computing systems [1]–[4]. They hide complexities (data distribution, concurrency, failures, etc.) from programmers and, at the cloud scale, they provide a powerful abstraction to atomically process massive sets of distributed data sets [5]–[8].

While distributed transactional key-value (KV) stores are extensively used to build scalable applications with a high degree of reliability and cost-effectiveness, offloading Tx processing in the cloud also poses serious security threats [9]. In untrusted cloud environments, adversaries can compromise the confidentiality and integrity of the data and application's execution state while they can also violate Txs semantics (isolation, atomicity) by intentionally returning stale/uncommitted data. Prior work has shown that software bugs, configuration errors and security vulnerabilities pose a real threat for storage systems [10]–[14]. These security threats are amplified in distributed stores as the state is distributed across machines connected to the untrusted storage and network system stacks.

This work pursues the following question: *How to design a high-performant, serializable, distributed transactional KV store that offers strong security properties?*

A promising direction to this question is to use trusted execution environments (TEEs)—the new trend in confidential computing—to build a secure distributed transactional (Tx) KV store. TEEs provide a secure memory area (enclave) where the residing code and data are protected even against privileged code (e.g., OS, hypervisor). Based on this promise, TEEs are now being streamlined by all major CPU manufacturers [15]–[19], and adopted by major cloud providers [20]–[22].

However, we cannot use TEEs out-of-the-box to build a secure distributed KV store with Txs. Particularly, we need to address the following three challenges:

First, TEEs only protect a limited volatile memory region (enclave) within a single node. These security properties do not naturally extend to the untrusted persistent storage and network over a distributed set of nodes, which are essential to build a secure distributed transactional KV store.

Secondly, TEEs primarily rely on the expensive syscall mechanism for I/O operations, where the enclave threads need to perform an extremely costly *world switch* to execute the syscall. While modern confidential computing frameworks [23]–[25] provide an asynchronous syscall I/O mechanism [26] to alleviate the performance overheads, they are still inadequate for modern distributed storage systems that prominently rely on high-performance networking such as RDMA or kernel-bypass [2], [27]–[29]. Unfortunately, it is not trivial to combine high-performance networking with the TEEs because TEEs fundamentally prohibit unauthorized access to the protected enclave via a DMA connection.

Thirdly, distributed stores need to ensure secure and crash-consistent persistency for committed Txs. Secure persistency for distributed systems can be a challenge in an untrusted environment where adversaries can rollback the database state, to a stale yet consistent snapshot violating correctness. While SGX [17] provides h/w trusted counters, a fundamental building block for rollback protection, writes incur prohibitively high latency [30], [31]. Further, we need to establish trust between the nodes in the distributed setting to protect against forking attacks. TEEs' attestation mechanisms provide a building block to bootstrap trust. Unfortunately, they cannot provide collective trust for a distributed set of nodes [32].

To address these challenges, we present TREATY, a distributed KV store with serializable ACID transactions [33] and strong security properties: *integrity*—unauthorized changes can be detected, *confidentiality*—unauthorized entities cannot read the data, *freshness*—stale state of the system can be detected. TREATY embodies three core contributions:

1. **Distributed Tx protocol:** The design of a secure two-phase commit (2PC) protocol for distributed Txs providing strict serializability. Our protocol leverages TEEs for security, and it is co-designed with a kernel-bypass network stack for TEEs to ensure high-performance execution (§ V).
2. **Stabilization protocol:** The design of a stabilization protocol that guarantees secure and crash-consistent persistency

of committed Txs. That is, the protocol ensures crash-consistency, recovery, and data freshness across rollback and forking attacks in distributed settings (§ VI).

3. **Trusted substrate for distributed TXs:** The design of a trusted substrate for distributed Txs—with which we build TREATY—that overcomes the limitations of TEEs. Specifically, we propose *(a)* a secure network library for Txs based on kernel-bypass I/O within TEEs, *(b)* a secure storage engine for Tx processing, *(c)* a userland-scheduler for low-latency requests, and *(d)* a memory allocator for secure Tx buffers management (§ VII).

We implement TREATY from the ground-up as a distributed KV store [34], [35], where we layer a distributed Tx layer (2PC) on top of per-node storage engine based on a secure version of RocksDB's [3] storage engine: SPEICHER [31]. Our secure 2PC is co-designed with Intel SGX as the TEE and a secure networking library based on eRPC [36].

We evaluate TREATY with TPC-C [37] and YCSB [38] on a real hardware cluster. Our evaluation shows that TREATY incurs reasonable overheads—6×-15× and 2×-5× for distributed and single-node Txs, respectively—while providing serializable distributed Txs and strong security properties. The overheads derive mainly from TEEs as (1) native runs of TREATY perform similarly to RocksDB, (2) encryption increases the overhead up to 1.4× compared to non-encrypted versions and (3) stand-alone evaluation of TREATY's 2PC shows 2× slowdown w.r.t. a native version of the protocol.

II. BACKGROUND

A. Distributed Transactional KV Stores

Distributed KV stores [34], [35], [39]–[42] reliably store and process large data-sets by offering Tx APIs. Such systems (ZippyDB [34], CockroachDB [35], etc.) traditionally layer query processing and Txs on top of a per-node storage engine, e.g., RocksDB [3] or LevelDB [43]. We also adopt this architecture. These persistent storage engines are increasingly based on log-structured merge-trees (LSM) [3], [34], [39], [43]–[45] due to their superior read/write performance. TREATY builds on RocksDB [3] where the data is stored in multiple levels, increasing in size. Higher levels (MemTables) are stored in the memory while the bulk of the lower levels (and thus of the data) is stored on disk in SSTables. Updates are applied to the MemTable and when it exceeds a maximum size, it is merged into the next lower level (*compaction*). If this causes the next level to exceed its own maximum size, the compaction cascades further. The system remains correct under failures through a combination of write-ahead logging (WAL) and a MANIFEST file that records all changes in the system.

RocksDB supports Txs in two ways: pessimistic Txs acquire locks as they go along (*two-phase locking*). Whereas, optimistic Txs validate their R/W sets at the commit time.

B. Confidential Computing

TEEs [15]–[19] offer a tamper-resistant confidential computing environment that guarantees the integrity and confidentiality of code and data, even in the presence of a privileged attacker (hypervisor or OS). TREATY relies on Intel SGX, a set of x86 ISA extensions for TEE [46] that offers the abstraction of an isolated memory, the enclave. Enclave pages reside in the Enclave Page Cache (EPC)—a specific memory region (94 MiB in v1, 256 MiB in v2) that is protected by an on-chip Memory Encryption Engine. For larger enclave sizes, SGX implements a, rather expensive, paging mechanism [23].

Confidential computing frameworks leverage TEEs to secure unmodified applications. They can broadly be categorized as libOS-based systems [25], [47]–[49], and host-based systems [23], [24], [50]. All of these efforts seek to minimise the number of enclave transitions, *world switches*, due to their high cost (e.g., TLB flushing, security checks [46]). TREATY is built on top of SCONE [23] that exposes a modified libc and combines user-level threading and asynchronous syscalls [26] to reduce the cost of syscall execution.

C. SPEICHER *Storage Engine*

SPEICHER [31] is a secure storage system based on RocksDB and SGX that offers authenticated and secure LSM data-structures. SPEICHER neither supports Txs nor distribution. Clients execute PUTs whose ordering is only secured in a future synchronization point. Shutdowns/crashes in the meantime requires clients to re-execute the operations which might change their initial order. TREATY uses SPEICHER as the underlying storage system but it extends the following to support Txs processing (§ VII-B): controller, buffer management, I/O subsystem, and LSM & logging data structures.

D. High-Performance Networking

Distributed systems mandate high-performance communication. Conventional applications use syscalls that incur the overheads of kernel context switches [51]–[55]. Consequently, approaches like RDMA and DPDK [56] are widely favored for high-performance as they *(i)* map a device into the users address space, and *(ii)* replace the costly context switches with a polling-based approach. In our work, we build a network stack modifying eRPC [36], a general-purpose and asynchronous remote procedure call (RPC) library for high-speed networking for lossy Ethernet or lossless fabrics on top of DPDK. eRPC uses a polling-based network I/O along with userspace drivers, eliminating interrupts and syscall overheads from the data path which is extremely imperative in the context of SGX. Lastly, eRPC supports a wide range of transport layers such as RDMA, DPDK, and RoCE.

III. THREAT AND FAULT MODEL

TREATY extends the standard SGX threat model [47] to provide stronger security guarantees even for a distributed setting, where we also consider the untrusted storage and network. An adversary can (1) control the entire software stack outside the enclave (including the network stack, i.e., they can drop, delay, or manipulate network traffic) and, (2) view/modify all non-enclave memory, i.e., untrusted host memory and persistent storage (SSDs). The adversary can perform rollback attacks and revert nodes to a stale state by

intentionally shutting them down and replaying older logs. We assume a *crash-fail* recovery model: nodes can crash at any point and will eventually recover. In-memory state is lost upon failure; persistent state (SSDs) is preserved. TREATY guarantees serializability in the presence of failures, and maintains data integrity, confidentiality and freshness.

We do not protect against side-channel attacks: cache timing, speculative execution [57]–[64], access pattern leakage [65], [66], memory safety vulnerabilities [67], [68] or denial of service attacks.

IV. OVERVIEW

A. System Overview

Figure 1 illustrates our system architecture. TREATY is a sharded transactional KV system, where we layer the Tx layer that implements a secure 2PC protocol (Agreement protocol) on the top of on a persistent KV store (SPEICHER): multiple nodes in the system store subsets of the data and coordinate to maintain consistency. Each node consists of two parts: 1) a trusted set of components that resides in the enclave memory and contains the Txs layer, lock manager, and Txs KV engine, and 2) the untrusted network and storage stack.

Clients communicate with the system through a mutually authenticated channel. TREATY exposes a standard transactional API: Txs begin and end through BEGINTXN() and TXNCOMMIT()/TXNROLLBACK() calls, and execute operations through TXNPUT() and TXNGET() operations. More specifically, TREATY maintains the following properties:

- *Security.* TREATY guarantees confidentiality, integrity and freshness for all Txs in the presence of untrusted storage and networking over a distributed set of nodes.
- *Programmability.* TREATY offers general serializable ACID Txs, offering the strongest possible correctness guarantees, combined with general purpose, interactive Txs that minimize the programming burden on developers.
- *Performance.* TREATY's careful design minimizes the performance limitations of TEEs (limits on EPC memory, high latency of trusted counter and I/O execution).

TREATY achieves security by designing two protocols: *(i)* a 2PC protocol for the correct and secure execution of distributed Txs (§ V) and, *(ii)* a stabilization protocol for secure and crash-consistent persistence of the committed Txs (§ VI). Lastly, TREATY's substrate (§ VII) for distributed Txs is designed and implemented with consideration to the TEEs architectural limitations (enclave memory, I/O, scheduling).

TREATY shares the Tx execution workflows of existing systems. Authenticated clients start Txs by selecting a transaction coordinator, who is responsible for driving the Tx's execution. Upon receiving a read or write request for a key, the relevant node acquires respectively a R/W lock, storing it in a local lock table. When the Tx is ready to commit, the Tx coordinator initiates a 2PC protocol consisting of a prepare and commit phase. The Tx commits if all involved shards vote to commit. Otherwise, the Tx aborts. In either case, locks are released.

B. Design Challenges

#1: TEE for distributed transactional KV stores. In the untrusted cloud, adversaries can tamper with *(i)* Txs' execution (e.g., compromise the confidentiality and authenticity of the running Txs and the 2PC's state), and *(ii)* the KV store's content (e.g., unauthorized modifications to the store's data).

For secure distributed Txs, we can rely on a simple 2PC protocol that leverages the security guarantees of TEEs. Unfortunately, TREATY cannot use a TEE as a black box as its security guarantees are restricted only to the (*limited and volatile*) enclave memory of a single node. In contrast, modern transactional systems like TREATY are distributed, communicate over the network, and store their data on a persistent storage medium (SSDs). To implement distributed Txs with TEEs, TREATY needs to overcome the following system challenges.

Security and correctness for Txs. Our 2PC needs to ensure confidentiality and integrity along with serializability detecting adversaries that aim to double execute Txs.

Untrusted persistent storage. TREATY needs to protect the persistent data by detecting unauthorised modifications since attackers can tamper with logs to compromise the history of executed Txs and the 2PC state and/or can delete/modify/access the persistent data.

Enclave memory. TREATY needs to overcome the limited enclave memory challenge. The limited enclave memory is especially problematic for LSM-based systems which rely on a large MemTable to absorb recent read/write requests (before compacting them to the SSTable). Moreover, the Tx layer on top of LSM storage system must also buffer the uncommitted writes for ongoing Txs. Lastly, network buffers for communication further pressurize EPC.

We discuss TREATY's approach for secure distributed Txs in § V. TREATY offers secure and correct execution of distributed Txs by implementing a secure 2PC (§ V-A) leveraging TEEs and a secure network library (§ VII-A). TREATY also adopts SPEICHER's [31] LSM data-structures as a secure store for the untrusted storage (§ V-B, § VII-B), but it extends and adapts SPEICHER's storage engine and data structures for the Tx processing and the design of the 2PC protocol for TEEs.

#2: Networking for distributed Txs. TREATY's nodes communicate with each other. Traditional kernel-based approaches for network I/O (e.g., sockets) experience high overheads due to context switches that are further deteriorated inside the SGX due to the costly enclave transitions.

Confidential computing frameworks, such as SCONE [23], implement async syscalls to eliminate the expensive world switches, but they still rely on the syscall mechanism for the I/O, which is slow and requires two additional data copies (enclave↔host memory↔kernel). This I/O mechanism is ill-suited for distributed systems [2], [27]–[29], like TREATY, that prominently rely on high-performance networking with direct I/O or kernel by-pass. Unfortunately, these direct I/O mechanisms are incompatible with TEEs, since TEEs prohibit enclave memory access via the untrusted DMA connection.

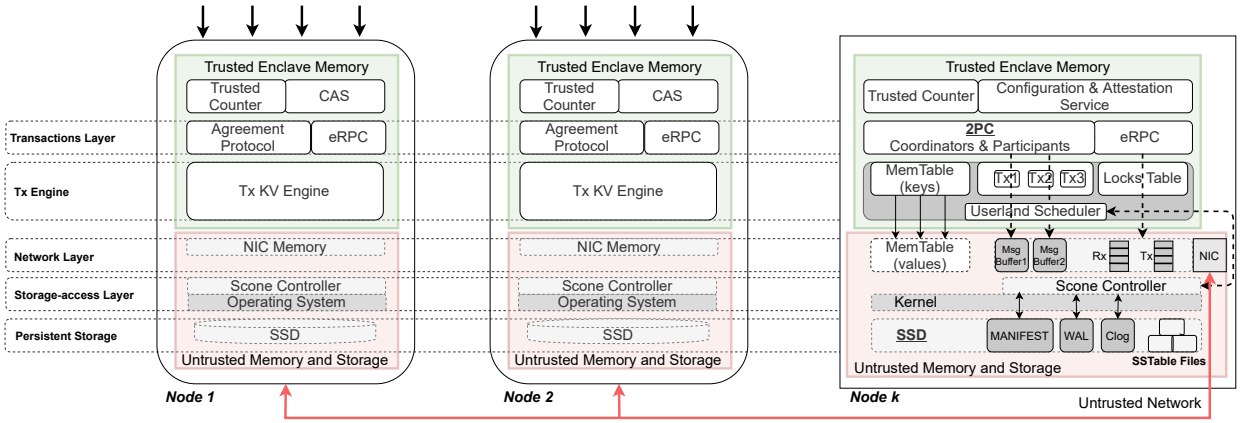

Figure 1: TREATY's system architecture.

Therefore, we need to adapt this mechanism in the context of SGX to use it and design the secure distributed 2PC protocol.

TREATY implements a secure network library (§ VII-A) through which we build the secure 2PC protocol to enable user-space direct I/O (DPDK [56]) based on eRPC [36]. TREATY's secure network library provides high-performant network I/O overcoming the limitations of SGX.

#3: Secure persistency. TREATY needs to ensure that committed Txs are persisted, remain crash consistent across reboots and are protected against forking/rollback attacks.

Trust establishment. Remote attestation (RE) ensures that the expected code is running, thus, protecting against forking attacks. SGX's RE, provided by Intel Attestation Service (IAS) [69], verifies a measurement of the enclave. Unfortunately, it is designed for a single-node attestation, not offering collective trust for distributed nodes in a data center, while it incurs high latency (requires explicit communication with the IAS). This can significantly slowdown recovery after reboots/migrations, where nodes require re-attestation.

Crash consistency. Logs are commonly used to persist the state and updates of Txs for durability. As these logs reside in the untrusted storage, recovery needs also to verify their freshness and integrity.

Distributed rollback protection. Trusted counters are widely used to protect against rollback attacks. TREATY further extends their scope to preserve serializability where Txs are stored along with a trusted counter value that cannot be overwritten. Consequently, the trusted counter values reveal Txs' order as well as the latest trusted state of the system.

While SGX does provide us with monotonic h/w counters, they suffer from three limitations: 1) high latency (e.g., increments can take up to 250 ms [70]) 2) non-recoverability if the CPU fails—indeed, at high-rate, counters wear out after a couple of days [70], and 3) they cannot offer rollback protection to a set of machines as they are private per-node.

TREATY designs a stabilization protocol—incorporated into the 2PC–to ensure crash consistent and secure persistency for Txs (§ VI). First, TREATY uses a Configuration and Attestation Service (CAS)—hosted within the data center to avoid the calls to IAS—to attest all its nodes. Secondly, it provides crash consistency for Txs through secured persistent logs. Lastly, we build on an asynchronous trusted counter service to avoid the SGX counter limitations and ensure distributed rollback protection (e.g., all parts of a distributed Tx are securely committed (persisted) to all participant nodes).

V. TRANSACTION PROTOCOL

TREATY's 2PC protocol ensures the correct and secure execution of distributed Txs (§ V-A). To achieve this, we leverage TEEs to harden the security properties of the 2PC, which we co-design with a high-performance network library based on kernel-bypass (§ VII-A), that guarantees strong security for the untrusted network. To realize distributed Txs, we also design single-node Txs support in SPEICHER (§ V-B).

A. Secure Distributed Transactions

Distributed design. TREATY partitions data into shards that may be stored on separate machines that fail independently from each other. Each TREATY node runs a transactional single-node KV storage engine built on top of RocksDB/SPEICHER [31], as shown in Figure 1. We implement a secure 2PC protocol with the userspace network stack based on eRPC [36] to execute distributed Txs and guarantee security properties. For securing the state of the protocol as well as providing secure recovery we make use of authenticated log files (MANIFEST, Clog and WAL). MANIFEST logs the changes in the state of the persistent storage (e.g., compactions, live logs). WAL stores the MemTable updates and the prepared Txs. Lastly, Clog is written by Txs coordinators and keeps the 2PC protocol state.

The system's initialization requires a trusted configuration and attestation service to establish trust in the distributed system. It distributes to nodes important information about the cluster configuration (e.g., secrets and keys' distribution to nodes, network connections).

Clients access TREATY over the network. For each Tx, a TREATY's node initialises a global Tx handle that is uniquely identified by a monotonically sequence number and the node id. A Tx coordinator interacts with the client and distributes their requests to the involved participant nodes. Participants

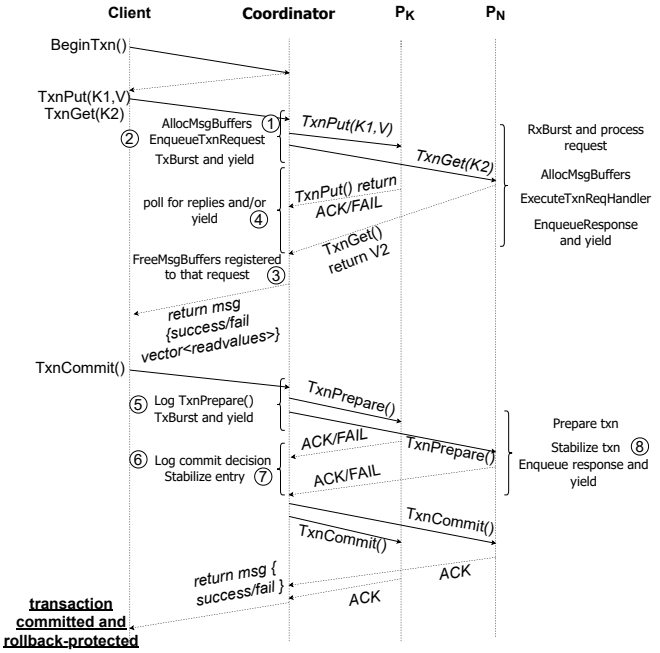

Figure 2: TREATY's two-phase commit protocol.

create local private Txs through TREATY's single-node transactional KV store (§ V-B). To ensure isolation, TREATY's engines own a private (per-node) keys lock table.

Lastly, we leverage the exit-less approach of executing syscalls provided by SCONE for accessing the persistent storage. Prior work [31], has introduced SPDK [71]. However, we did not use SPDK for two reasons: *(i)* in our experiments the database fits entirely in the kernel page cache therefore read access was much faster than SPDK which would have to read from SSD and *(ii)* we configured SCONE to best fit TREATY for storage I/O syscall execution.

Integrity, confidentiality and freshness. Each node runs a single modified SPEICHER instance. TREATY engine runs inside the enclave to ensure integrity, confidentiality and freshness for the execution and the resided run-time data (e.g., MemTable, transactions' local buffers, hash values).

To extend the trust to persistent storage, we adapt SPEICHER which offers a secure authenticated SSTable hierarchy. SPEICHER stores encrypted blocks of KV pairs as well as a footer with the blocks' hash values (for integrity checks). TREATY extends the persistent data structures by adding an extra log file, the Clog for the 2PC. Lastly, to ensure crash recovery in TREATY, we defer deleting the old SSTables and logs until MANIFEST's entries for that compactions are stabilized.

TREATY also extends the trust for the network I/O by constructing a secure message format for Txs (§ VII-A). A message encapsulates an Initialization Vector (12 B) and a MAC (16 B) for proving its authenticity and integrity. In addition to Tx's data, we also add some metadata (e.g. node, Tx and operations identifiers) that allows TREATY to protect against duplication of packages by an attacker.

Two-phase commit. TREATY offers serializable distributed ACID Txs with strong security guarantees throughout a secure 2PC protocol implemented over our secure network stack (§ VII-A). Figure 2 illustrates the complete protocol design. Clients are registered to TREATY nodes and thereafter, are able to execute transactions. Upon a client's request, the transaction's coordinator node (TxC) initializes a global Tx which is uniquely identified in the entire cluster and associated with a specific RPC communication channel. Each RPC is strictly owned by one thread, which minimizes shared resources.

The TxC distributes the Tx's requests to the responsible nodes and/or processes its own requests. As shown in Figure 2, before forwarding the requests to the participants, each Tx reserves (untrusted) memory for the requests and responses ①. These message buffers have to remain allocated until the entire request has been served ③. To eliminate paging overheads, they reside encrypted in the untrusted host memory.

Once the message is constructed, the TxC en-queues the request ②. Note that en-queuing the request does not transmit the message. In case of multiple requests, coordinators can defer the transmissions until all requests are en-queued. Once the TxC has executed its own-managed requests and has forwarded all requests to the participants, it yields and periodically checks if the participants have replied ④.

At a commit, TREATY first prepares the Tx for a distributed commit accross all parties involved. Every Tx/operation is logged to Clog with its own unique trusted counter value ⑤. Afterwards, all participants prepare their local Tx. Participants delay replying back to the coordinator until the prepare entry in the log is stabilized ⑧. TREATY's stabilization ensures that coordinators will not consider the Tx as successfully prepared until all participants ensure that they are able to recover and commit the transaction after a crash. If not all participants ensure that their prepare phase is stabilized, after a crash this entry cannot be safely recovered. Especially in cases where the participants had already committed the entry but only some of them could recover the committed Tx after a crash, the system would be in a inconsistent state where distributed Txs are partially committed to some, but not all involved, nodes.

The TxC, before committing/aborting, also stabilizes the prepare's phase decision on the Clog ⑥-⑦. If the TxC crashes before this entry is stable, the recovered coordinator will re-execute the prepare phase. Once this is rollback protected, the Tx can commit. We do not need to wait for the commit entry to be stable to reply to the client. Even if the system crashes, this Tx can be committed in the exact same order.

B. Secure Single-node Transactions

KV Storage engine and single-node Txs. TREATY's storage engine runs inside the enclave for which the security properties are guaranteed. TREATY leverages SPEICHER's data model that offers an authenticated LSM structure for the persistent storage but also optimizes the usage of EPC memory. Particularly, TREATY adapts SPEICHER's MemTable design by separating the keys from the values. We keep keys along with their version number inside the enclave, while we place the encrypted values in the untrusted host. To access values and

prove their authenticity we similarly keep a pointer to the value as well as its secure hash value along with the key.

However, SPEICHER cannot support Txs; therefore we extend it to integrate both optimistic and pessimistic Txs exporting an interface to the upper Tx layer to access the LSM-data structure. We preserve the RocksDB's interface and semantics. For the persistent storage, TREATY extends the persistent data structures by adding an extra log file, the Clog for the 2PC. TREATY's distributed Txs can then be viewed as the set of all participants' single node Txs.

Pessimistic Txs take locks on their keys while optimistic Txs use sequence numbers to identify conflicts at the commit phase. For optimistic Txs, each key has a seq. number showing its the latest version and is atomically increased during the commit phase. At commit, Txs log their updates to the WAL and update the MemTable. We only reply to a client after the Tx becomes stable, ensuring that upon a crash, clients will not have to re-execute successfully committed transactions. Thus, conflicting transactions will maintain their initial ordering.

Lock tables. Nodes store a table of locks for their keys that is divided across shards, each protected with a lock, by splitting the key space. TREATY runs with a big number of shards to avoid locking bottlenecks. Txs that fail to acquire a lock within a timeframe, return with a timeout error.

VI. STABILIZATION PROTOCOL

TREATY's stabilization protocol ensures *secure and crash-consistent persistency* for the committed Txs. To achieve this, our protocol relies on three core principles. First, TREATY establishes trust between the nodes based on collective remote attestation. Secondly, after the 2PC's execution (§ V), TREATY ensures crash consistency for the committed Txs. Lastly, once Txs are crash-consistent, TREATY ensures rollback protection in distributed settings. We next explain these three principles.

Distributed trust establishment. Upon startup TREATY bootstraps a Configuration and Attestation Service (CAS) on a node in the network to provide scalable remote attestation and authentication. For attestation, the service provider verifies the CAS over Intel Attestation Service (IAS). On success the service provider deploys an instance of TREATY's local attestation service (LAS) on all nodes, verified by the CAS over IAS. The LAS replaces the Quoting Enclave (QE), collecting and signing quotes for all TREATY instances, running on the node. After the CAS verified a new instance, it supplies the instance with the necessary configuration, e.g., network key, nodes' IPs, etc. The CAS is also used to authenticate clients and establish trust between TREATY and clients.

Crash consistency and recovery. After the 2PC's execution, TREATY ensures crash consistency and recoverability using three persistent log files; MANIFEST, WAL and Clog. As discussed in § V, Clog logs the 2PC states, WAL the committed data and MANIFEST stores the state changes in the SSTables. TREATY relies on these logs being written sequentially; thus, it assigns to each of their entries a unique, monotonic and deterministically increased trusted counter value. The recovery protocol relies on that property to detect rollback attacks or verify freshness and state continuity. Precisely TREATY's recovery verifies that the state of the persistent storage and logged Txs is the most recent (through the verification of the logs) and recovers the most recent stable state.

Upon restart MANIFEST is replayed first; it recovers the SSTable hierarchy and loads metadata (hashes of SSTable's blocks) that will be used to verify the integrity and the freshness of a SSTable upon access into the enclave. Note that TREATY's garbage collector only deletes SSTable files when the newly compacted ones refer to stabilized entries in MANIFEST. MANIFEST also recovers all the "live" WAL and Clog files. Similarly, TREATY makes sure that the old versions of the logs are not deleted before their effect to the database has been rollback protected (stabilized). For example, a WAL is marked for deletion as long as the matching MemTable has been successfully compacted and this compaction action refers to a stable entry in the MANIFEST. The Clog is deleted as long as there are no unstable entries and does not contain any unfinished prepared transaction entry.

After the MANIFEST, TREATY replays in order all live WALs to restore the latest MemTables. The WAL also contains the prepared Txs. Therefore, each node will also re-initialize all prepared Txs that are not yet committed. For each prepared Tx, the node communicates with the Tx's coordinator for either committing or aborting.

Lastly, Clog is replayed. TREATY restores the state of the 2PC protocol for all prepared on-going Txs. The coordinator will re-execute the prepare phase, if it cannot guarantee that the Tx will succeed. If the prepare phase decision is logged, then, thanks to the stabilization function of TREATY, these Txs are also prepared in the participant nodes. The coordinator will then instruct the participants to commit. If a node has already committed the Tx, this message is ignored.

Distributed rollback protection. For secure persistency, TREATY provides rollback protection across distributed Txs by leveraging a trusted counter service. While our design is independent of the trusted counter service, we adopt Rote [70], a fault-tolerant distributed system where enclaves preserve the counters freshness with $2\,\mathrm{ms}$ average latency.

For each log file, TREATY initializes a unique trusted counter and assigns a monotonically and deterministically increasing counter value to each log entry. TREATY's criterion for freshness is that 1) only log entries with counter value less than the trusted service's value can be recovered, 2) the counter values are deterministically increased—for *state continuity*, e.g., deleted or reordered entries are detected, and 3) last log entry's value match the counter's value.

TREATY accesses the trusted counter service through the network. The communication is asynchronous to maximize CPU usage. As discussed in § V the 2PC incorporates the stabilization protocol ensuring distributed rollback protection—Txs are only considered committed (and clients get notified) after the commit decision has been stabilized in the logs.

TREATY's trusted counter service implements an *echo broadcast* [72] protocol with an extra confirmation message

in the end. A sender-enclave (SE) sends the counter update to all enclaves of the protection group. Receivers-enclaves (REs) send back to the SE an *echo*-message which they store along with the counter value in the protected memory. Once the SE receives echo-messages from the quorum (q) it starts a second round of echo-messages. Upon receiving back the echo, each RE verifies that the received counter value matches the one it keeps in-memory and RE replies with a (N)ACK message. After receiving q ACKs, the enclave seals its own state together with the counter value to the persistent storage.

Secure persistency guarantees. TREATY's attestation and its secure LSM-data structure [31] ensure that TREATY maintains its security properties after a crash as (1) only trusted nodes obtain the encryption keys for the persistent storage, (2) nodes perform integrity checks on accessed persistent data blocks and, (3) at recovery, TREATY verifies the logs' freshness. As the underlying cloud infrastructure is owned by a third-party, TREATY detects but cannot *prevent* unauthorized modifications to persistent state.

Stabilization protocol correctness. TREATY stabilization protocol remains correct as TEEs guarantee its correct execution on all nodes. Any faults, e.g., crashes or network partitions, can only affect availability. While TREATY's trusted counter offers crash fault tolerance, CAS can be a single point of failure. In case CAS fails, crashed nodes cannot recover.

VII. Trusted Substrate for Distributed TXs

To support secure Tx processing, we design the following four cross-layer subsystems for our trusted substrate: a secure network library (§ VII-A), a secure storage engine for Txs based on Speicher [31] (§ VII-B), a userland thread scheduler (§ VII-C), and a memory allocator for Tx buffers (§ VII-D).

A. Network Library for Txs

To implement TREATY's 2PC, we build a secure networking library that implements asynchronous remote procedure calls (RPCs) for Txs execution. Our network library relies on eRPC [36], but we had to extend and adapt the codebase to *(i)* overcome the architectural limitations of TEEs (I/O, enclave memory and DMA-ed memory) and, *(ii)* ensure confidentiality, integrity and freshness for the over-the-network-communication in the presence of malicious attackers.

Architectural limitations of TEEs. To avoid the execution of expensive syscalls for network I/O, we adapt eRPC with DPDK as the transport layer. DPDK offers direct I/O, bypassing the kernel and eliminating the syscalls overheads using userspace drivers and polling.

To secure the software stack, we build eRPC/DPDK with SCONE assuring that the device's DMA mappings reside in the host memory, thus accessible by both enclave and NIC. We achieve this overwriting the `mmap()` of SCONE to bypass its shield layer and allow the allocation of untrusted host memory as well as the creation of memory mappings to the hugepages.

Furthermore, we change the library's memory allocator to place all message buffers in the host memory (in hugepages of 2 MiB), thus reducing the EPC pressure at the cost of encrypting them. While eRPC by default creates shared memory regions for message buffers in hugepages, a naive port of eRPC with SCONE allocates all of these buffers inside the enclave triggering the costly EPC paging. Lastly, we eliminate `rdtsc()` calls to reduce the number of OCALLs from the hot path by replacing the call with a monotonic counter.

Message layout. TREATY's networking library constructs a secure message to guarantee the integrity and confidentiality of messages through a en-/decryption library based on OpenSSL [73]. Additionally, we ensure freshness, i.e., *at-most once* execution semantics for Txs' execution. The message is comprised of a 12 B Initialization Vector (IV), a payload of 4 B (for memory alignment), a 80 B Tx metadata and Tx data that contains the size of the data and the size of the key and/or value followed by the key and/or value. The message is followed by a 16 B MAC. MAC and IV are necessary to prove the authenticity and integrity in the remote host. Only the metadata and data are encrypted; in case IV or MAC are compromised the integrity check will fail. The metadata contains the coordinator node's id (8 B) and the Tx id (64 B), monotonically incremented in the coordinator node. Both are necessary for uniquely identifying the transaction in the recipient side. The operation identifier (8 B) is also unique for each Tx request. This unique tuple of the node's, Tx and operation ids ensures that an operation/Tx is not executed more than once. Therefore, along with the two-phase locking which ensures that only one Tx can modify a resource, nodes can verify that no already executed Txs are processed again. Similarly, the participants' reply, except for the ACKs, also include the coordinator's node, Tx id and the operation id.

TREATY' networking protocol enqueues requests, e.g., a user-defined message, that triggers a request handler for this request type in the remote machine. The execution returns after enqueuing the request. The node can enqueue more requests or process received ones. Once the request is processed in the remote machine, the receiver replies back to the sender. A continuation function is triggered in the host machine to notify that the request has been completed. The sender can now deallocate any related resources, e.g., message buffers.

B. Storage Engine: Extensions to SPEICHER for Txs

To offer persistent Txs in TREATY, we extend SPEICHER's storage engine/controller [31] to support single-node pessimistic and optimistic transactions as discussed in § V-B.

Additionally, we implement an extra persistent log file, the Clog. Clog's entries are similar to MANIFEST and WAL entries format; they are comprised of a counter value, the encrypted Tx data and metadata and a cryptographic hash. Clog's deletions are also logged in the MANIFEST. Clog is thread-safe; coordinators append independently their entries.

In TREATY, we allow group commits for Txs to flush bigger data blocks to the persistent storage and optimize the SSD throughput. Each group elects a leader that merges their and all followers' Txs buffers into a larger buffer. The leader then writes this buffer into WAL and MemTable. We further

defer logging (yield) at commit, allowing us to format group commits of bigger data blocks. For the LSM structures, we implement a MemTable skip list that supports parallel updates for concurrent Tx processing.

Lastly, we change the I/O sub-system of SPEICHER, where we replace the SPDK-based direct I/O for accessing the SSDs with async syscalls to optimize the usage of cores for our eRPC/DPDK-based networking library.

C. Userland Scheduler

Timer based scheduling in the enclave is extremely expensive, as it involves interrupts that result in world switches. While SCONE implements its own userspace scheduler, it is non-preemptive relying on threads to either go to sleep or issue syscalls for ensuring progress. This design is not well-suited for TREATY; *(i)* our direct I/O networking library leads to starvation and high latency, and *(ii)* in the presence of multiple clients creating too many threads is inefficient.

We overcome these by implementing a *userland scheduler* on top of SCONE's scheduler. Precisely, each thread spawns one userland thread (*fiber*) for each connected client. Our userland scheduler implements a per-core round-robin (RR) algorithm for fibers' scheduling and a set of queues (run queue and sleeping/waiting queue) for the fibers.

When a fiber needs to block, e.g., acquiring a lock, waiting on condition variables or sleeping, TREATY's userland scheduler places the fiber into a sleeping queue. It picks and schedules the next eligible fiber from the run queue (based on the RR algorithm). Our userland scheduler does not involve interrupts, syscalls and context/world switches when scheduling another fiber. Lastly, we adapted our scheduler to frequently yield threads allowing SCONE to schedule others. Precisely, if no fiber is in a running state, our scheduler sleeps; thereby invoking a syscall. Our scheduler's sleep function yields to another SCONE thread and increases the amount of time before future yields are triggered. In this way, fibers allow us to both maximize CPU utilization and increase scalability.

Our userland scheduler's implementation is based on Boost [74]. We configure SCONE with 8 kernel and 8 application threads each spawning one fiber per client.

D. Memory Management

We minimize EPC usage or paging; TREATY's in-memory data structures are divided between the enclave and untrusted host memory. All network buffers are kept in host memory at the cost of encryption. Note that transmission is asynchronous so heavy network traffic could exceed EPC limit and trigger paging if the message buffers were allocated in the enclave.

TREATY's engine keeps the updates of uncommitted in-progress Txs into local buffers. We implement Txs' buffers as a stream of bytes (`std::string`) that allocate continuous memory to eliminate paging.We also explored the case to adopt a design similar to the MemTable for Txs buffers, where we keep only the keys in the enclave (for the read-my-own writes semantics). However, we decided against it as it does not offer any performance improvements; at commit, we still need to perform integrity checks, re-collect and encrypt all the KV pairs in the enclave memory for logging. We implement a scalable memory allocator for host and enclave memory that relies on a mempool. It assigns threads to different heaps based on the hash of the `get_id()` and recycles unused memory, drastically reducing the amount of mapped memory.

Implementation details. We implement TREATY in C/C++; 4000 LoC for the 2PC, encryption library and modifications to eRPC, DPDK, boost and SPEICHER codebases. We use Java and Rust for the workload generator and CAS respectively.

VIII. EVALUATION

A. Experimental Setup

Testbed. We perform our experiments on a real hardware testbed using a cluster of 6 server machines. We run TREATY on 3 SGX server machines with CPU: Intel(R) Core(TM) i9-9900K each with 8 cores (16 HT), memory: 64 GiB, caches: 32 KiB (L1 data and code), 256 KiB (L2) and 16 MiB (L3). TREATY nodes are connected over a 40GbE QSFP+ network switch. Clients generate workload on 3 machines and are connected with TREATY over a secondary 1Gb/s NIC.

Benchmarks/workloads. We evaluate TREATY's 2PC w/o any underlying storage (§ VIII-B). For the distributed (§ VIII-C) and single-node (§ VIII-D) Txs evaluation, we use YCSB [38] and TPC-C [37]. We configure TPC-C with 10 Warehouses, as in [75]. For distributed Txs, we also run a TPC-C workload with 100 Warehouses. Lastly, we evaluate the network stack (§ VIII-E) by stress-testing the network using: *(i)* iPerf [76] (implemented w/ kernel-sockets), and *(ii)* our own server/client application, build with eRPC [36], that implements iPerf. Unless stated otherwise, we refer to overheads for throughput (tps).

B. TREATY's 2PC Protocol

We evaluate TREATY's 2PC protocol designed over eRPC with the YCSB workload (50 %R-50 %W). 2PC runs without any underlying storage to isolate the protocol's overheads. We compare two Secure (w/ SCONE) versions of TREATY 2PC with and w/o Enc(ryption) against two Native executions of the protocol with and w/o Enc(ryption) respectively. All four versions "saturate" with 300 clients, each of which executes a YCSB workload (10 Ops/Tx, 1000 B value size).

Figure 4 shows the slowdown in the throughput of 3 versions of TREATY's 2PC protocol (Native 2PC w/ Enc, Secure 2PC w/o Enc, Secure 2PC w/ Enc) normalized to a native, non-secure version of 2PC. Some Tx's operations might be served by the coordinator node; therefore not all operations are sent thought the network to participants and thus, be en-/decrypted. Our evaluation shows minimal encryption overhead in the native case. Further, TREATY's secure 2PC w/o Enc experiences 1.8× slowdown w.r.t. a native execution while encryption (Secure 2PC w/ Enc) increases the overheads leading to a 2× slowdown in comparison with native 2PC.

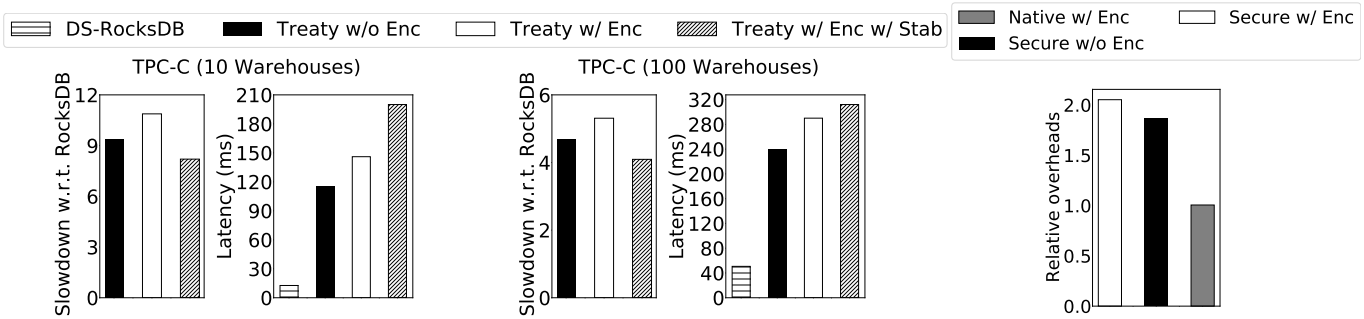

Figure 3: Performance evaluation of distributed transactions under two TPC-C workloads with 10W and 100W respectively.

Figure 4: Throughput slowdown of three versions w.r.t. Native 2PC.

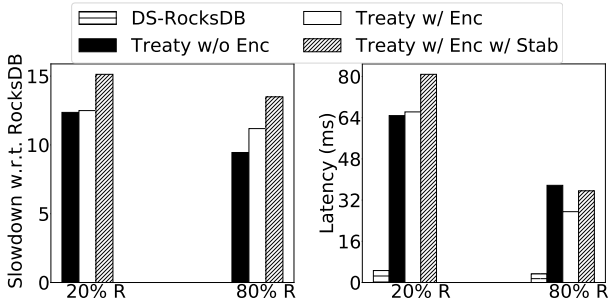

Figure 5: Performance evaluation of distributed Txs under a W-heavy (20 %R) and a R-heavy (80 %R) YCSB workload.

C. Distributed Transactions

Baselines and setup. We evaluate the performance of distributed Txs under two TPC-C workloads, with 10 and 100 Warehouses, and two YCSB workloads: read-heavy (80 %R) and write-heavy (20 %R). We show the overheads of TREATY's throughput normalized w.r.t. a native execution of 2PC with RocksDB as the underlying storage (DS-RocksDB). We study the performance behavior of three systems: *(i)* TREATY w/o Enc, *(ii)* TREATY w/ Enc and *(iii)* TREATY w/ Stab(ility) w/ Enc. All three versions run with SCONE and our TREATY's secure storage system.

Results. YCSB. Figure 5 (left) shows the throughput slowdown of the three systems w.r.t. DS-RocksDB. TREATY's performance is 9×—15× worse compared to DS-RocksDB where SCONE overheads fast dominate the performance (TREATY runs w/ and w/o Enc have little differences). For the W-heavy workload, DS-RocksDB achieves 18.5 ktps. All four systems are saturated with 96 clients equally divided across all three machines (each serving 32 clients). Distributed Txs require both participants and coordinator to stabilize their entries and therefore, TREATY rollback protection increases latency further for write-heavy Txs, as shown in Figure 5 (right).

For the R-heavy workload, TREATY w/ Enc slows down the execution 11× while the un-encrypted version of the system shows a slowdown of 9.5×, both compared to native DS-RocksDB (24 ktps). Encryption overheads are reasonable; reading from SSTables requires integrity checks as well as proving the freshness of the entry. All four systems present different scaling capabilities. DS-RocksDB and TREATY w/o Enc scale up to 92 clients while encrypted versions cannot scale more than 60 clients. Therefore, TREATY is over saturated in the benchmark, explaining the higher latency values. TPC-C (10W). Figure 3 (left) shows the throughput overheads and the latencies of three versions of TREATY (all run in SCONE) w.r.t DS-RocksDB under TPC-C with 10 Warehouses. TREATY is 8×—11× slower compared to the native, non-secure DS-RocksDB. This configuration presents heavy W-W conflicts; DS-RocksDB achieves 780 tps. Consequently, DS-RocksDB, TREATY w/o Enc and TREATY w/ Enc cannot scale for more than 10 clients. However, TREATY w/ Enc w/ Stab scales up to 16 clients as the stabilization period (where locks are released) allows the system to serve more requests.
TPC-C (100W). Figure 3 (right) shows the throughput overheads and the latencies of three versions of TREATY (all run w/ SCONE) w.r.t DS-RocksDB under TPC-C with 100 Warehouses (total worksize equals to 10GB divided equally to all 3 nodes). This configuration presents less conflicts than the previous case; DS-RocksDB achieves 1200 tps. Our evaluation shows reasonable overeheads (4×-6×) and similar behavior for TREATY w/ Enc and Stab; while all the three other systems (DS-RocksDB, TREATY w/ Enc, TREATY w/o Enc) are saturated with 60 clients, TREATY w/ Enc w/ Stab is saturated with 84 clients.

D. Single-node Transactions

Baselines and setup. We evaluate the performance of pessimistic and optimistic single-node Txs with TPC-C and YCSB. TPC-C is configured with 10 Warehouses as in [75] and YCSB with: 10 ops/Tx, value size to be equal to 1000 B, uniform distribution with 10 k unique keys. For the pessimistic Txs, we measure the performance against read-heavy (80 %R-20 %W) and write-heavy (20 %R-80 %W) workloads, while for the optimistic Txs we use the read-heavy workload. Our experiments stress-test EPC usage since both TREATY and RocksDB do not support in-place updates. We evaluate the throughput (tps) and latency for 6 versions of the single-node TREATY; *(i)* RocksDB, *(ii)* Native TREATY, *(iii)* Native TREATY w/ Enc, *(iv)* TREATY w/o Enc (SCONE), *(v)* TREATY w/ Enc (SCONE) and *(vi)* TREATY w/ Enc w/ Stab (SCONE).

Results. Pessimistic Txs. Figure 6 shows the throughput and latency of the TPC-C for the pessimistic Txs. TREATY executed natively (Native TREATY) performs equivalently to RocksDB. Additionally, we deduce that Native TREATY w/

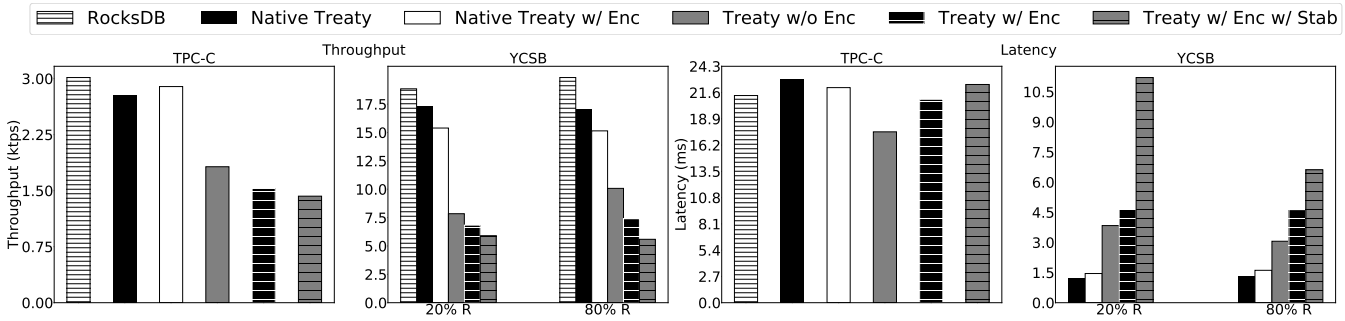

Figure 6: Performance evaluation of pessimistic single-node transactions under TPC-C and YCSB benchmarks. YCSB performance is evaluated with a write heavy (20 % reads) and a read heavy (80 % reads) workload.

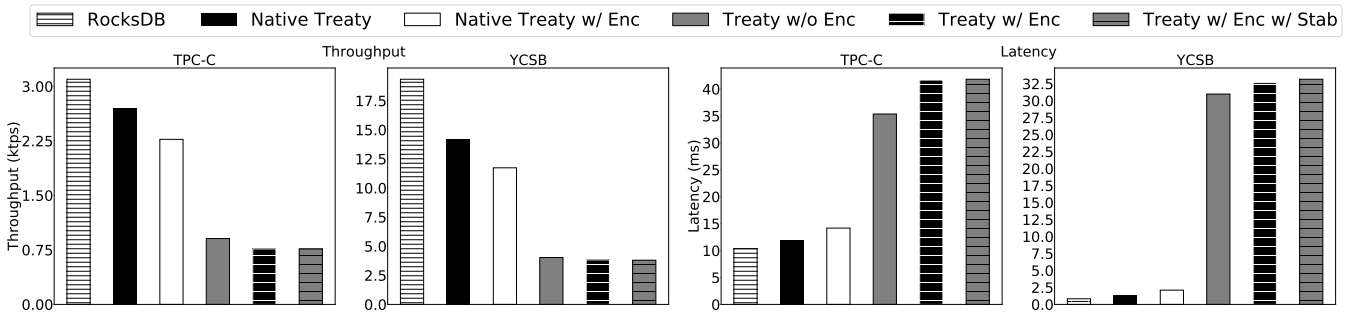

Figure 7: Performance evaluation of optimistic single-node transaction under TPC-C and YCSB benchmarks. YCSB performance is evaluated with a write heavy (20 % reads) and a read heavy (80 % reads) workload.

Enc adds minimal overhead compared to the non-encrypted versions. Further, SCONE's overheads are reasonable. TREATY w/o Enc has roughly $1.6\times$ slowdown compared to RocksDB while TREATY w/ Enc has $2\times$ slowdown. Lastly, the stabilization period seems not to have great impact on the overall throughput. We experience a $2.1\times$ slowdown compared to RocksDB. Regarding the latency, we see that all TREATY SCONE versions do not scale as good as the native execution. However, the latency of SCONE systems is equivalent or smaller to the natively executed versions. This behavior is reasonable since the native versions are "saturated" to 64 clients while the SCONE versions to 32 clients.

Additionally, Figure 6 shows the throughput and latency of all 6 systems for the two YCSB workloads. YCSB's configuration, in contrast to TPC-C, present little conflicts. That said, for the read-heavy workload, encryption, adds a throughput overhead of $1.3\times$ and $2.7\times$ compared to native and SCONE versions respectively while the respective overheads for latency are $1.6\times$ and $4.6\times$. For the write-heavy workload, we have $1.2\times$ and $2.8\times$ slowdown to native and SCONE versions compared with RocksDB. The latency overheads are $1.5\times$ and $4.7\times$ respectively. Similarly to TPC-C, TREATY's stabilization function does not impact performance dramatically. We experience $3.5\times$ slowdown for the read-heavy workload and $3.2\times$ slowdown with respect to RocksDB for the write-heavy workload compared to when de-activating the stabilization mechanism. Further, especially for the read-heavy workload, we find out that TREATY w/ Enc w/ Stab takes advantage of the "idle" (stabilization) time to improve the scalability; TREATY w/ Enc w/ Stab becomes saturated in 64 clients while the otherversions are saturated in 32 clients.

Optimistic Txs. Figure 7 shows that TREATY w/ Enc w/ Stab performs $5\times$ and $4\times$ worse compared to the native RocksDB for TPC-C and YCSB, respectively. We see that TREATY's stabilization does not incur extra throughput overhead compared to the TREATY w/ Enc as the system, thanks to our userspace fiber scheduler, continues to process requests. TREATY w/ Enc w/ Stab's compared to TREATY w/ Enc experiences roughly 10 % latency overhead. Further, we notice that TREATY w/ Enc w/ Stab's saturation point under YCSB is 128 clients while RocksDB's one is 32. TREATY shows similar overheads as SPEICHER [31] which is the most related system.

E. Network Library for Txs

We evaluate the performance of TREATY's networking library using iPerf against six baselines: eRPC (SCONE), eRPC (native), iPerf-UDP (native), iPerf-UDP (SCONE), iPerf-TCP (native), and iPerf-TCP (SCONE). All native (eRPC and iPerf) versions do not provide any security. Additionally, SCONE (eRPC and iPerf) versions do not secure network layer; we only use the secure message format for TREATY-networking. Note that iPerf build with SCONE is optimized w.r.t to SGX since SCONE uses the async syscalls [26] for performance.

For the sockets (native and SCONE), we use iPerf to measure the throughput. For the eRPC versions and TREATY-networking, we implement a client-server model with eRPC to implement iPerf. Our experiments saturate network bandwidth where we compare the performance with different packet sizes. iPerf supports TCP and UDP, eRPC supports only UDP.

Figure 8 shows the throughput in network bandwidth for all seven systems discussed (TREATY networking, eRPC (SCONE), eRPC (native), iPerf-UDP (native), iPerf-UDP

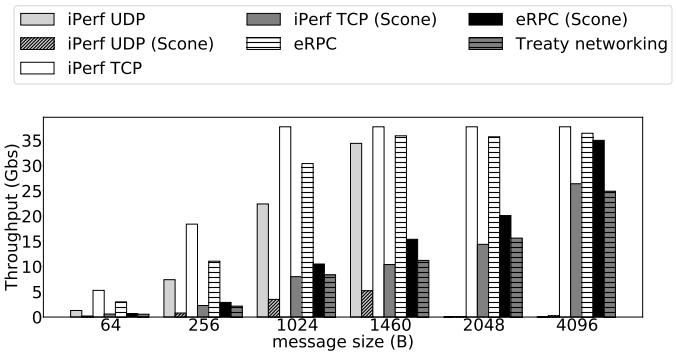

Figure 8: Throughput in network bandwidth of TREATY-networking, eRPC (native and SCONE), iPerf-TCP and iPerf-UDP (native and SCONE).

(SCONE), iPerf-TCP (native), and iPerf-TCP (SCONE)). We see that eRPC is comparable to iPerf-TCP while iPerf-UDP performs poorly. Especially for large messages ($>$ MTU), UDP throughput equals zero as many messages are dropped. In contrast to UDP, TCP performs equivalently and better than eRPC. We deduct this to the fact that TCP is optimized for high speed bulk transfers and, additionally, the entire TCP/IP stack processing is frequently offloaded to the network controller. For small and medium packets sizes that are still smaller than the MTU ($1460\,\mathrm{B}$), we observe performance differences between eRPC and iPerf-TCP. Especially, for packet sizes of $256\,\mathrm{B}$ and $1024\,\mathrm{B}$, eRPC shows roughly $30\,\%$ and $22\,\%$ slowdown respectively compared to iPerf-TCP. For larger messages, both eRPC and iPerf-TCP perform almost equivalently.

We deduce two core conclusions: (a) SCONE's overhead is significant—SCONE deteriorates up to $8\times$ iPerf-TCP (SCONE) while up to $4\times$ eRPC; and (b), due to the amount of syscalls, eRPC in SCONE performs up to $1.5\times$ faster than iPerf-TCP (SCONE). As discussed, syscalls execution in the enclave incurs heavy overheads. Note the bigger the packet size is, the worse the performance becomes. Lastly, we see that TREATY network stack which also fully secures the network and includes the encryption overheads performs equivalently to iPerf-TCP (SCONE) that do not provide any security. As a result, iPerf-TCP (SCONE) is an inappropriate design.

F. Recovery Protocol

We next evaluate the overheads of TREATY recovery w/ and w/o Enc compared with native recovery. We construct logs of 800K entries each that lead to log sizes of $69\,\mathrm{MiB}$ and $91\,\mathrm{MiB}$ for the non-encrypted and encrypted entries respectively. In this experiment we use relatively small log entries (e.g $100\,\mathrm{B}$ per log entry) which is the worse case for TREATY as: *(i)* we have more syscalls, and *(ii)* we have more decryption calls.

Table I shows that TREATY recovery without decryption costs incurs roughly $1.5\times$ slowdown compared to the native recovery. Further, encryption increases the overheads by up to $2\times$ slower than the native recovery.

Version	Slowdown	Version	Slowdown
TREATY w/o Enc	$1.5\times$	TREATY	$2.0\times$

Table I: Recovery overheads w.r.t. native recovery.

IX. RELATED WORK

Confidential computing frameworks [23], [25], [47], [77] use TEEs to build secure systems [31], [78]–[85]. TREATY leverages SCONE to build the first secure distributed transactional KV storage system with TEEs.

Secure systems for cloud computing [75], [80], [86]–[95] offer different security properties, interfaces, threat model, and security enforcement mechanisms. EnclaveDB [80] is the most related work. In contrast to TREATY, it (1) is a single-node in-memory system (w/o persistence and distribution), (2) runs in emulated h/w and, (3) assumes unlimited enclaves. TREATY targets a distributed storage system, where we extend the security properties to storage and network and overcome the limitations of TEEs. Other storage systems vary on hardware, security guarantees and interfaces: KV APIs [31], [78], [96] and filesystems [97]–[99]. Precursor [94] combines SGX with RDMA offloading the cryptographic operations to clients. In contrast, TREATY provides distribution, persistency and Txs.

Secure distributed storage systems [100]–[102] provide consistency, durability, availability and integrity. CloudProof [102], as TREATY, distrusts the cloud provider but it requires (1) clients to guarantee these security properties and (2) a trusted proxy which limits scalability. TREATY leverages TEEs to avoid such limitations.

Other distributed systems [2], [27]–[29] deploy RDMA as TREATY. However, we target security which is more challenging; DMA connections for direct I/O are not allowed by TEEs. ShieldBox [83] uses DPDK to overcome this limitation, but it targets only layer 2 in the OSI model which is limiting for distributed systems. SPEICHER [31] uses SPDK [71] for direct I/O to the SSDs. rkt-io [49] provides a library OS in the enclave including a full network stack. We build on these advancements to build a secure direct network I/O mechanism for TEEs with which we design a 2PC protocol.

X. CONCLUSION

In this paper, we present TREATY, a secure distributed transactional KV store for untrusted cloud environments. TREATY offers high-performance serializable Txs with strong security properties.We achieve these design goals by building on hardware-assisted secure Txs with SGX and designing a distributed 2PC protocol with a direct I/O network library based on eRPC. Further, we design a stabilization protocol for Txs using an asynchronous trusted counter interface along with a distributed attestation service. We implement an end-to-end secure Tx processing system from the ground-up based on RocksDB/SPEICHER as the underlying storage engine. Our evaluation with the YCSB and TPC-C shows reasonable overheads for TREATY, while it provides strong security properties.

Software artifact. TREATY is publicly available: https://github.com/TUM-DSE/Treaty.

Acknowledgements. We thank our shepherd, Prof. Fernando Pedone. We also thank Dr. Le Quoc Do, Dimitris Stavrakakis and Prof. Jana Giceva for their helpful comments. This work was supported in parts by a Microsoft Research PhD Fellowship and Huawei Research, UK RISE and BaCaTeC Grants.

REFERENCES

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review (SIGOPS)*, 2007.

[2] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[3] "RocksDB, A persistent key-value store," https://rocksdb.org/, last accessed: Dec, 2018.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," 2013.

[5] Amazon, "Amazon S3 Cloud Object Storage," https://aws.amazon.com/s3, last accessed: Dec, 2018.

[6] Microsoft, "Azure Blob Storage," https://azure.microsoft.com/en-us/services/storage/blobs, last accessed: Dec, 2018.

[7] Google, "Cloud Storage," http://www.cloud.google.com/storage, 2017, last accessed: Dec, 2018. [Online]. Available: https://cloud.google.com/storage/

[8] Dell, "Elastic Cloud Storage," https://www.dellemc.com/en-us/storage/ecs/, 2017, last accessed: Dec, 2018. [Online]. Available: https://www.dellemc.com/en-us/storage/ecs/index.htm

[9] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards Trusted Cloud Computing," in *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[10] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[11] CRN, "The ten biggest cloud outages of 2013," https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, 2013, last accessed: Dec, 2018. [Online]. Available: https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm

[12] N. Santos, R. Rodrigues, and B. Ford, "Enhancing the os against security threats in system administration," in *Proceedings of the 13th International Middleware Conference (Middleware)*, 2012.

[13] G. Goodson and B. Schroeder, "An analysis of data corruption in the storage stack," in *6th USENIX Conference on File and Storage Technologies (FAST 08)*. San Jose, CA: USENIX Association, Feb. 2008. [Online]. Available: https://www.usenix.org/conference/fast-08/analysis-data-corruption-storage-stack

[14] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901339

[15] ARM, "Building a secure system using trustzone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, last accessed: Jan, 2021.

[16] "Arm Confidential Compute Architecture," https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture, last accessed: May 2021.

[17] "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx, last accessed: Jan, 2021.

[18] AMD, "AMD Secure Encrypted Virtualization (SEV)," https://developer.amd.com/sev/, last accessed: Jan, 2021. [Online]. Available: https://developer.amd.com/sev/

[19] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[20] A. Cloud, " Alibaba Cloud's Next-Generation Security Makes Gartner's Report," https://www.alibabacloud.com/blog/alibaba-clouds-next-generation-security-makes-gartners-report_595367, last accessed: Jan, 2021.

[21] Microsoft Azure, "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/, last accessed: Jan, 2021.

[22] "Introducing Google Cloud Confidential Computing with Confidential VMs," https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms, last accessed: Jan, 2021. [Online]. Available: https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms

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

[24] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "PANOPLY: Low-TCB Linux Applications with SGX Enclaves," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

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

[26] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-less System Calls," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[27] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[28] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro, "Fast General Distributed Transactions with Opacity," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, 2019.

[29] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, *Fast In-Memory Transaction Processing Using RDMA and HTM*, 2015.

[30] "Intel, "SGX documentation: sgx create monotonic counter"," https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter/, last accessed: Dec, 2018.

[31] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani, "SPEICHER: Securing lsm-based key-value stores using shielded execution," in *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[32] F. Gregor, W. Ozga, S. Vaucher, R. Pires, D. L. Quoc, S. Arnautov, A. Martin, V. Schiavoni, P. Felber, and C. Fetzer, "Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2020)*, 2020.

[33] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, p. 631–653, Oct. 1979. [Online]. Available: https://doi.org/10.1145/322154.322158

[34] "How big is rocksdb adoption?" https://rocksdb.org/docs/support/faq.html, last accessed: May 2021.

[35] "CockroachDB," https://www.cockroachlabs.com/, last accessed: May 2021.

[36] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[37] "TPC-C," http://www.tpc.org/tpcc/, April 4, 2022.

[38] "YCSB," https://github.com/brianfrankcooper/YCSB, last accessed: Jan, 2021.

[39] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *Proceedings of the 28th ACM Symposium on Principles of distributed computing (PODC)*. ACM, 2009.

[40] "MongoDB," https://www.mongodb.com/, last accessed: May 2021.

[41] "Couchbase," https://www.couchbase.com/, last accessed: May 2021.

[42] L. Bindschaedler, A. Goel, and W. Zwaenepoel, "Hailstorm: Disaggregated compute and storage for distributed lsm-based databases," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[43] "LevelDB," http://leveldb.org/, last accessed: Dec, 2018.

[44] "Apache HBase," https://hbase.apache.org/, last accessed: May 2021.

[45] "Apache AsterixDB," https://asterixdb.apache.org/, last accessed: May 2021.

[46] V. Costan and S. Devadas, "Intel SGX Explained," 2016.

[47] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[48] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," 2019.

[49] J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, and P. Pietzuch, "Rkt-io: A direct i/o stack for shielded execution," in *Proceedings of the Sixteenth European Conference on Computer Systems (ACM EuroSys 21)*, 2021.

[50] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein, "Eleos: ExitLess OS services for SGX enclaves," in *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*, 2017.

[51] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementationi (NSDI)*, 2014.

[52] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[53] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[54] V. Vasudevan, D. Andersen, and M. Kaminsky, "The Case for VOS: The Vector Operating System," in *13th Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[55] "How long does it take to make a context switch?" https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, last accessed: Jan, 2021.

[56] "Intel DPDK," http://dpdk.org/, last accessed: Jan, 2021.

[57] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxeattack.com/, 2020.

[58] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions," 2020.

[59] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[60] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[61] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[62] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[63] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[64] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

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

[66] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[67] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: Memory Safety for Shielded Execution," in *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[68] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[69] "Intel Corporation. Attestation Service for Intel Software GuardExtensions (Intel SGX): API Documentation." https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf, last accessed: Jan, 2021.

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

[71] "Intel Storage Performance Development Kit," http://www.spdk.io, last accessed: Dec, 2018. [Online]. Available: http://www.spdk.io

[72] M. K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)*, 1994.

[73] "OpenSSL library," https://openssl.org, last accessed: Jan, 2021. [Online]. Available: https://openssl.org

[74] "boost: C++ libraries," https://www.boost.org/, last accessed: Aug, 2020. [Online]. Available: https://www.boost.org/

[75] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi, "Obladi: Oblivious Serializable Transactions in the Cloud," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

[76] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/, last accessed: Aug, 2020. [Online]. Available: https://iperf.fr/

[77] "Asylo: An open and flexible framework for enclave applications," https://asylo.dev/, last accessed: Jan, 2021. [Online]. Available: https://asylo.dev/

[78] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: Policy enhanced secure object store," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[79] F. Schuster, M. Costa, C. Gkantsidis, M. Peinado, G. Mainar-ruiz, and M. Russinovich, "VC3 : Trustworthy Data Analytics in the Cloud using SGX," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[80] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database using SGX (S&P)," in *IEEE Symposium on Security and Privacy*, 2018.

[81] B. Trach, R. Faqeh, O. Oleksenko, W. Ozga, P. Bhatotia, and C. Fetzer, "T-lease: A trusted lease primitive for distributed systems," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[82] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer, "Clemmys: Towards secure remote execution in faas," in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[83] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "ShieldBox: Secure Middleboxes using Shielded Execution," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[84] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer, "Securetf: A secure tensorflow framework," in *Proceedings of the 21st International Middleware Conference (Middleware)*, 2020.

[85] M. Bailleu, D. Dragoti, P. Bhatotia, and C. Fetzer, "Tee-perf: A profiler for trusted execution environments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[86] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[87] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[88] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the 39th international conference on Very Large Data Bases (VLDB)*, 2013.

[89] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious Query Processing for Secure Databases," in *Proceedings of the VLDB Endowment (VLDB)*, 2019.

[90] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "Orthogonal security with cipherbase," in *Proc. of the 6th CIDR*, 2013.

[91] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[92] S. Bajaj and R. Sion, "Trusteddb: a trusted hardware based database with privacy and data confidentiality," in *In Proceedings of the 2011 international conference on Management of data*. ACM, 2011, pp. 205–216.

[93] U. Maheshwari, R. Vingralek, and W. Shapiro, "How to build a trusted database system on untrusted storage," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000.

[94] I. Messadi, S. Neumann, N. Weichbrodt, L. Almstedt, M. Mahhouk, and R. Kapitza, "Precursor: A fast, client-centric and trusted key-value store using rdma and intel sgx," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–13. [Online]. Available: https://doi.org/10.1145/3464298.3476129

[95] M. Bailleu, D. Giantsidi, V. Gavrielatos, D. L. Quoc, V. Nagarajan, and P. Bhatotia, "Avocado: A secure in-memory distributed storage system," in *2021 USENIX Annual Technical Conference (ATC'21)*, 2021.

[96] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "ShieldStore: Shielded In-Memory Key-Value Storage with SGX," in *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*, 2019.

[97] D. Garg and F. Pfenning, "A proof-carrying file system," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[98] C. Weinhold and H. Härtig, "jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[99] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post, "Guardat: Enforcing data policies at the storage layer," in *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*, 2015.

[100] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud Storage with Minimal Trust," in *ACM Transactions on Computer Systems*, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2063509.2063512

[101] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the salus scalable block store," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[102] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling security in cloud storage slas with cloudproof," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2011.