Avocado: A Secure In-Memory Distributed Storage System

Maurice Bailleu, Dimitra Giantsidi, and Vasilis Gavrielatos, *University of Edinburgh;* Do Le Quoc, *Huawei Research;* Vijay Nagarajan, *University of Edinburgh;* Pramod Bhatotia, *University of Edinburgh and TU Munich*

https://www.usenix.org/conference/atc21/presentation/bailleu

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Avocado: A Secure In-Memory Distributed Storage System

Maurice Bailleu[1], Dimitra Giantsidi[1]
Vasilis Gavrielatos[1], Do Le Quoc[2]*, Vijay Nagarajan[1], Pramod Bhatotia[1,3]
[1]*University of Edinburgh* [2]*Huawei Research* [3]*TU Munich*

Abstract

We introduce Avocado, a secure in-memory distributed storage system that provides strong security, fault-tolerance, consistency (linearizability) and performance for untrusted cloud environments. Avocado achieves these properties based on TEEs, which, however, are primarily designed for securing limited physical memory (enclave) within a single-node system. Avocado overcomes this limitation by extending the trust of a secure single-node enclave to the distributed environment over an untrusted network, while ensuring that replicas are kept consistent and fault-tolerant in a malicious environment.

To achieve these goals, we design and implement Avocado underpinning on the cross-layer contributions involving the network stack, the replication protocol, scalable trust establishment, and memory management. Avocado is practical: In comparison to BFT, Avocado provides confidentiality with fewer replicas and is significantly faster—4.5× to 65× for YCSB read and write heavy workloads, respectively.

1 Introduction

In-memory distributed key-value stores (KVS) [8, 29–31, 40, 44, 45, 61, 66, 88, 106] have been widely adopted as the underlying storage system infrastructure in the cloud because *(i)* they support latency sensitive applications by keeping data in main memory, and *(ii)* they are able to accommodate large datasets beyond the memory limits of a single server by adopting a scale-out distributed design.

At the same time, the transition to the cloud has increased the risk of security violations in storage systems [77]. In untrusted environments, an attacker can compromise the security properties of the stored data and query operations. In fact, several studies [36,37,83] show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems. Further, a malicious cloud operator or co-located tenant, presents an additional attack vector [78,79].

To address these security threats, hardware-assisted trusted execution environments (TEEs), such as Intel SGX [5], ARM Trustzone [12], RISC-V Keystone [53, 76], and AMD-SEV [10] provide an appealing way to build secure systems. In particular, TEEs provide a hardware-protected secure memory region whose residing code and data are isolated from any layers in the software stack including the OS/hypervisor. Given this promise, TEEs are now commercially offered by major cloud computing providers [23, 34, 60].

Although TEEs provide a promising building block for securing systems against a powerful adversary, they also present significant challenges while designing a *replicated secure distributed storage system*. The fundamental issue is that the TEEs are primarily designed to secure the limited in-memory state of a single-node system, and thus, the security properties of TEEs do not naturally extend to a distributed infrastructure. Therefore we ask the question: *How can we leverage TEEs to design a high-performance, secure, and fault-tolerant in-memory distributed KVS for untrusted cloud environments?*

In this work we introduce Avocado, a secure, distributed in-memory KVS based on Intel SGX [5] as the foundational TEE that achieves the following properties: (a) strong *security*, in particular, *confidentiality*—unauthorized reads are prevented, and *integrity*—unauthorized changes to the data are detected, (b) *fault tolerance*—the service continues uninterrupted in the presence of faults, (c) *consistency*—strong consistency semantics for a replicated store (linearizability), while protecting against roll back and forking attacks and (d) *performance*—achieving all of these without compromising performance.

To achieve the aforementioned properties, we need to address the following four design challenges pertaining to the network stack, the replication protocol, trust establishment, and memory management in TEEs.

Firstly, in-memory distributed KVSs are increasingly build on high-performance network stacks, where they bypass the kernel using direct I/O [4,30,46]. Unfortunately, the prominent I/O mechanism employed by TEE frameworks [13, 65, 72] is based on asynchronous system calls [85], which exhibit significant overheads [104]. On the other hand, the direct I/O mechanism is fundamentally incompatible with TEEs as the data stored within the protected memory of TEEs cannot be directly accessed via the untrusted DMA connection.

To address this challenge, we design a high-performance network stack for TEEs based on eRPC [46]—it supports the complete transport and session layers, while enabling direct I/O within the protected TEE domain. Our network stack outperforms asynchronous syscall by 66 % for `iperf` (§6.1).

Secondly, in-memory distributed KVSs rely on data replication for fault tolerance. To ensure replicas are consistent in the presence of faults and adversary, a secure replication protocol is deployed. While conventional wisdom requires the employment of BFT protocols [20, 52], they are prohibitively

*Do Le Quoc performed this work at TU Dresden.

expensive for practical systems [22].

To overcome the limitation, we design a secure replication protocol, which builds on top of any high-performance non-Byzantine protocol [56, 98]—our key insight is to leverage TEEs to preserve the integrity of protocol execution, which allows to model Byzantine behavior as a normal crash fault. Our replication protocol offers linearizable reads and writes, and outperforms BFT [87] by a factor of 4.5×—65×, while requiring f fewer replicas and stronger security properties (§ 6.2).

Thirdly, a secure distributed system requires a scalable attestation mechanism to establish trust between the servers and clients. Unfortunately, the remote attestation mechanism in TEEs is designed for establishing root of trust for a single node [68] and it does not provide collective trust establishment across the multiple nodes of a distributed system. Moreover, the attestation itself is based on Intel attestation service (IAS) [3, 11], which suffers from scalability and latency issues.

To address this, we design a configuration and attestation service (CAS) that ensures scalability and flexibility in a distributed environment. Further, it provides configuration management, and improved performance of 18.3× compared to Intel's IAS attestation (§ 6.3).

And lastly, an in-memory distributed KVS requires fast access to large amount of main memory on each server for single-node KVS. Unfortunately, TEEs provide a limited secure physical memory, and rely on prohibitively expensive paging mechanism to access data beyond the physical limit.

To address this limitation, we design a novel single-node KVS based on a partitioned skip list data structure, which overcomes the memory limitations of TEEs, while supporting lock-free scalable concurrent updates. Our KVS provides fast lookup speed; 1.5×—9× faster than ShieldStore [48], a state-of-the-art secure KVS for single-node systems (§ 6.4).

Based on these aforementioned four contributions, we build AVOCADO as an end-to-end system from the ground-up, and evaluate it using a real hardware cluster using the YCSB [7, 24] workloads. Our evaluation shows that AVOCADO is scalable and performs similar in read heavy and write heavy workloads: AVOCADO suffers only 50 % slowdown compared to a non-secure distributed KVS (§ 6.5), which is an order of magnitude better than the state-of-the-art secure distributed storage systems providing strong consistency (§ 7).

Limitations: AVOCADO requires a large trusted computing base (TCB) compared to other work using TEE to provided secure replication [18, 25, 26]. While BFT protocols can handle implementation errors, AVOCADO cannot and requires the TCB to be implemented correctly. Further, we do not aim to protect against side-channel attacks and access or network pattern attacks [37, 49, 55, 105]. Protecting against these attacks is outside of the scope of this work.

2 Background

2.1 Trusted computing

Trusted Execution Environments (TEEs) [5, 10, 12, 27, 76] are tamper-resistant processing environments that guarantee the authenticity, the integrity and the confidentiality of their executing code, data and runtime states, e.g. CPU registers, memory and others. Their content remains resistant against all software attacks even from privileged code (OS, hypervisor).

SGX, Intel's version of a TEE, offers the abstraction of an isolated memory called *enclave*. Enclave pages reside in the enclave page cache (EPC) — a specific memory region (up to 128 MiB for v1 and 256 MiB for v2) which is protected by an on-chip memory encryption engine (MEE). To support applications with larger memory footprint SGX implements a *paging* mechanism. However, the EPC paging mechanism incurs high overheads [16, 65].

This isolation prohibits SGX applications from executing outside-of-the-enclave code directly. Thus, enclave threads need to *exit* the trusted environment and further copy all associated data out of the enclave since kernel code cannot access it. Afterwards, threads have to *enter* the enclave again. We refer to this as *world switch*.

Our AVOCADO project leverages the advancements in shielded execution frameworks; in particular, we use SCONE [13] to build a distributed storage system.

2.2 High-performance networking with eRPC

Traditionally the network stack and the I/O are handled inside the OS kernel where conventional applications perform *system calls* to send/receive messages. However, context switches due to system calls present a bottleneck and might sacrifice performance [2, 38, 43, 86, 101]. Consequently, approaches like RDMA and DPDK [4] are widely favorized in high performance networking because they *(i)* can 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 our network stack based on eRPC [46], a state-of-the art general-purpose and asynchronous remote procedure call (RPC) library for high-speed networking for lossy Ethernet or lossless fabrics. eRPC uses a polling-based network I/O along with userspace drivers, eliminating interrupts and system call overheads from the datapath.

Lastly, eRPC provides us with a UDP stack, leverages optimization techniques (e.g. zero-copy reception, congestion control, etc.) while it remains generic; it supports a wide range of transport layers such as RDMA, DPDK, and RoCE.

3 System model

AVOCADO divides the key space into shards. Each shard is replicated over a configurable number of nodes, which are connected over a high speed network. A client issuing a `Put`, `Get`, or `Delete` operation selects the shard associated with the key and chooses a *request coordinator* from the list of nodes. The nodes will coordinate with each other to provide proof for the success of the operation. For Get operations proofs of integrity, authenticity, consistency and non-/existence need to be provided, too.

Data model. AVOCADO provides confidentiality, integrity, authenticity and strong consistency for the stored data. Specifically, a server only acknowledges a request as long as it can prove the following guarantees: 1) an adversary cannot read or manipulate stored data, without the manipulation being detected, 2) the servers can establish trust with each other and the clients and 3) an operation always observes the latest completed operation on the same key e.g., a Get observes the latest Put.

Threat model. AVOCADO targets an extended threat model beyond the conventional model assumed for single-node shielded execution [17]. In line with the default threat model of SGX, we assume that an adversary has full control over the hardware and software stack of the provided system, including OS and hypervisor. Further, the adversary has the ability to gain full control over the network infrastructure and can drop, delay, or manipulate network traffic. In contrast to BFT protocols, we assume that adversary cannot take advantage of faults in the implementation of SGX or KVS. Moreover, our work does not protect against side-channel attacks [42, 49, 55, 62, 81, 97, 99, 100]. AVOCADO also does not provide mechanisms against access pattern attacks [37, 105]. Lastly, we also do not protect against memory safety vulnerabilities in our implementation [51, 63].

Fault model. We assume an asynchronous model with network and crash-stop failures. The network can be manipulated by the attacker, thus, we assume that message transmission delays can be unbounded, network packets can be reordered, lost or duplicated. We do not assume the existence of synchronized clocks. Individual processes might fail by crashing, but do not operate in a Byzantine manner (because of trusted execution in the nodes). Since the network is controlled by the attacker, AVOCADO cannot provide any availability guarantees. However, as long as there is not a denial-of-service attack on the network, AVOCADO will remain available while a majority of processes remain alive (tolerating f failures).

4 Design

AVOCADO, as a distributed KVS, runs on a set of nodes, each of which has to continuously guarantee the confidentiality, integrity and authenticity of the stored data as well as the sent/received messages. As shown in Figure 1, each node consists of four major components. On the top, a configuration and attestation service (CAS) runs to provide and speed up the trust establishment between the nodes and the clients. Additionally, AVOCADO guarantees fault tolerance as well as consistency between the replicated nodes thanks to an asynchronous replication protocol. We implement this replication protocol using our secure network stack. Further, the network stack securely sends and receives messages, ensuring packet security. Finally, the single-node memory

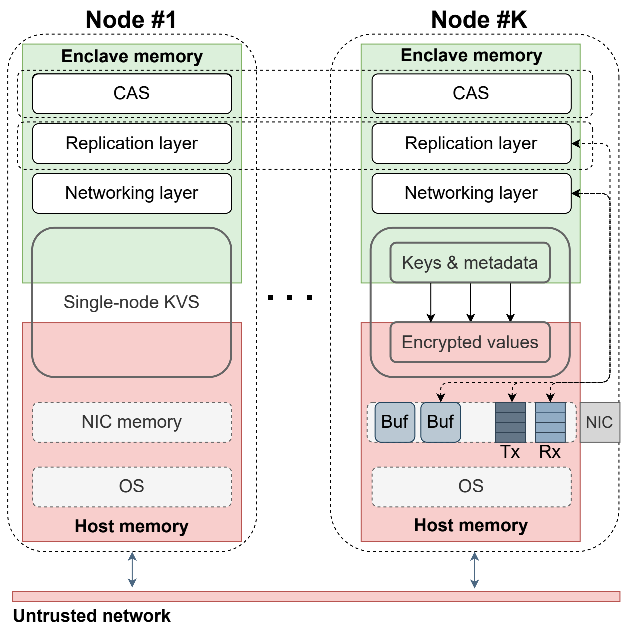

Figure 1: System overview.

KVS stores the dataset, containing all user provided data and ABD's timestamps.

Following, we will discuss the four major components, i.e. network stack, replication protocol, CAS, and in-memory KVS, in more detail.

4.1 Network stack

Problem. High-performance networking based on direct I/O mechanisms (e.g., RDMA and DPDK) is an essential ingredient to design a distributed in-memory KVS. The networking layer is imperative to support high-performance synchronous replication. Unfortunately, mapping devices into TEEs trusted memory is incompatible to the security guarantees, since the device is untrusted. Direct I/O mechanisms, however, depend on DMA.

Additionally, the synchronous socket syscall I/O is limiting as it requires the expensive world switch in the TEEs (the world switch is around 5.5× more expensive than a kernel context switch; 10,170 cycles compared to 1,800 cycles [104]).

To prevent the expensive world switches, asynchronous syscall mechanisms [85] have been adopted by shielded execution frameworks, such as SCONE [13] or Eleos [65]. Although the asynchronous syscall mechanism helps in mitigating the expensive world switch in TEEs, it is not well-suited for AVOCADO since the system call overhead as well as copying data in/out the enclave memory are not avoided. For example, in our evaluation in Section 6.1 we prove that the exit-less asynchronous socket-based networking is poor choice compared to a userspace approach for AVOCADO.

Solution. To overcome this limitation, we opted for a new network stack based on the userspace direct I/O networking approaches (e.g., RDMA and DPDK), offering a secure implementation of the transport and session layer in the OSI model. However, we need to tackle the fact that untrusted resources/memory cannot be mapped into the enclave memory. To address this, our network stack maps the DMA, and message buffers into the untrusted host memory, which is accessible by the enclave.

Shielded network stack. For our shielded network stack, we use eRPC [46] on top of DPDK [4]. To strengthen Avocado's security properties and eliminate world switches, we also map all eRPC's and DPDK's software stack to the enclave address space by leveraging Scone. Therefore, the logic, i.e. code, *lives* completely within the enclave while the networking buffers (e.g. message buffers, network protocol buffers, Tx and Rx queues) remain in host memory since SGX will not allow registering enclave memory to the NIC. As a result, we map untrusted host memory to both NIC and network buffers required by eRPC and we utilise hugepages memory of 2 MB-pages to boost packet processing (e.g. eliminate page walks, exploit data locality, minimize swapping, and increase TLB hit rate).

As shown in Figure 2, the submission and reception of requests and responses mandate the allocation of message buffers. To transmit the message buffer's data, eRPC needs to copy the data to a `rte_mbufs` in the Tx queue which is allocated by DPDK library and also resides in `hugepages` area. However, before that, a header that contains the transport header, and metadata (request handler type, sequence numbers, etc.) is added to the front of the packet. Specifically, eRPC library adds the UDP protocol header while the DPDK library is responsible for the Ethernet protocol header.

Upon a request's reception, a specific handler for the type of the request is invoked. The Rx queue's elements are pointers to the address of the received data. In case the packet is smaller than the `MTU` (1500 B in our case), we perform zero-copy reception by mapping the data address to the message buffer associated for that request. Our networking stack splits big packages (> `MTU`) into a set of ordered `MTU`-size smaller messages and delivers them in order—we guarantee to order by unique monotonic sequence ids. The first (sub)-message contains all the necessary metadata (e.g. the size of the original message). That way, if a message is lost, our library identifies the missing part and only the lost message is re-transmitted. Lastly, note that each user thread owns a separate RPC object which owns distinct Tx and Rx queues allowing that way multithreaded concurrent operations.

Encryption and message format. To sum up, Avocado efficiently eliminates the world switches, establishes a direct communication with the device bypassing the kernel network stack, attacks the limited enclave memory and promotes parallelism. However, by putting these buffers outside the hardware protected area, Avocado has to ensure the integrity and confidentiality for all network data. Towards this direction, we implemented an en-/decryption library (using hardware support for AES-GCM-128). Each call or return from eRPC goes through this en-/decryption layer which also checks the integrity of the transmitted data.

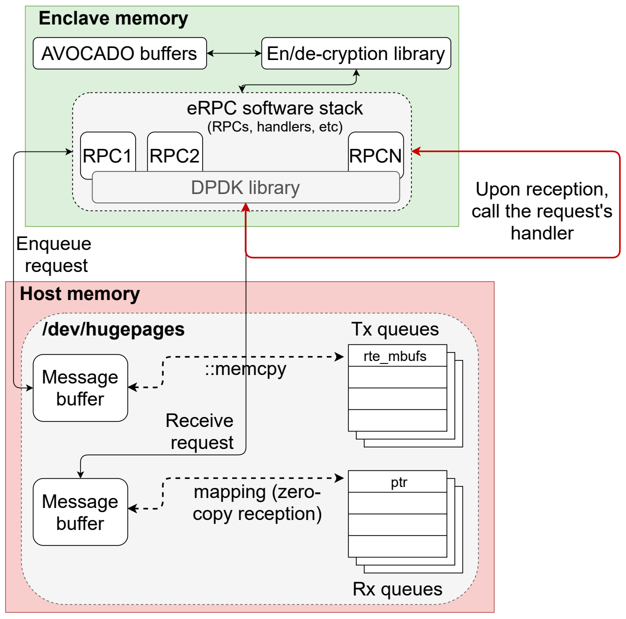

Figure 2: Avocado's network stack.

IV	OP	Key size	Package size	KV & Lamport clock	MAC
12B	8B	8B	8B	....	16B

Figure 3: Avocado's message format.

Figure 3 shows the message format of our Avocado-networking layer. For each transmitted packet, the encryption layer builds a payload, which contains a 12 B IV, 8 B operation identifier, 8 B for the key size, 8 B for the size of the entire package then the KV pair with the Lamport clock (§ 4.2). The generated payload is followed by a 16 B MAC which is necessary to prove the authenticity and integrity upon reception in the remote host. The operation identifier also contains a unique id for the current request/response, allowing to detect resend packages by an attacker. Replicas are trusted and all replicas authenticate each other in the boot up step. Further, they make a key exchange, therefore we can use this together with the unique id to authenticate each message. By encrypting and authenticating our packages we can deal with security concerns raised for user space networking [84, 94].

Result. In Section 6.1 we show that our userspace shielded network stack based on eRPC outperforms the kernel approach based on sockets up to 1.66×.

4.2 Replication protocol

Problem. Distributed systems enforce consistency in the face of faults through replication protocols that establish an order of operations in a replicated environment, preventing data corruption and loss. We strive for linearizability [39], the strongest guarantee from a programmability perspective, which mandates that each request appears to take effect globally and instantaneously at some point between its invocation and completion. Additionally, we strive to provide two often contradictory properties: security and

Protocols	Linearizability	Integrity	Confidentiality	Replication factor	Max compromised nodes
Non-Byzantine	✗	✗	✗	$2f+1$	0
BFT	✓	✓	✗	$3f+1$	f
BFT + TEEs	✓	✓	✓	$3f+1$	All
AVOCADO	✓	✓	✓	$2f+1$	All

Table 1: The landscape of replication protocols in the untrusted environment. This table compares theoretical systems with different protocols against AVOCADO. Thereby, all the systems utilize a secure single-node KVS, however only the execution of BFT + TEE is protected. We assume f compromised nodes for linearizability, integrity and confidentiality columns.

performance. Conventional wisdom suggests the use of BFT protocols [20, 52] since they provide a secure consensus protocol in a malicious environment. However, their performance suffers from their overly pessimistic assumptions. On the other hand, non-Byzantine replication protocols, such as ABD [14, 56], chain replication [98] or Raft [64], perform better than BFT, but cannot tolerate a malicious environment.

Solution. Since AVOCADO assumes a malicious environment, BFT [20, 52] protocols could be deployed to deal with malicious responses. Prior work uses trusted components to increase the performance of BFT protocols by detecting equivocation [18, 54]. However, our assumption of the system differs from BFT. In contrast to BFT, we assume that enclaves will respond correctly, preventing equivocation. Furthermore the TEE is able to preserve the integrity of the protocol execution. This allows us to model a Byzantine behavior as a normal crash fault. As a result, we can adopt a non-BFT replication protocol, which deals with crash faults. Thereby, our design greatly increases the performance by avoiding the additional broadcast rounds required by BFT, while also reducing the required nodes to tolerate f failures. In Table 1 we compare security guarantees of different protocols with and without TEEs.

In AVOCADO we build our replication protocol on the well established multi-writer ABD [56] protocol. (From now on "ABD".) By choosing ABD, we can also guarantee protection against forking and rollback attacks. ABD requires a majority of nodes to acknowledge each operation, guaranteeing that at least one replica involved in the operation has observed the most recent operation on the same key. This further guarantees liveness in case of network partitioning as long as a majority of nodes are in the same partition. While we do not change the replication-related behavior of the original ABD protocol, we design a secure replication protocol based on our network stack (§ 4.1). In the following we describe the important operations of AVOCADO.

#I: Put. In a `Put` operation the client will determine, by hashing the key and looking up the nodes, the set of nodes responsible for the key. They, then, send the `Put` to a randomly selected replica, which will act as the `Put`'s coordinator.

The chosen request's coordinator will prepare it's own KVS by preparing the local put operation, however it will not make the local put visible for other operations until the replicated `Put` operation is completed. This reduces EPC pressure, since the value doesn't have to be cached in enclave memory before it can be inserted into the nodes KVS. An example of the AVOCADO's `Put` request is shown in Figure 4. The coordinator, first, executes the first of two broadcast rounds. All replicas store the key-value along with its Lamport clock to determine an order of operations. The Lamport clock consists of a logical counter and a machine id. This id guarantees that only one machine can generate a specific clock value. In the first broadcast round, the coordinator requests the timestamps that are stored in the replicas for that key. All replicas lookup the key in their in-memory KVS, to find their stored timestamp. Crucially, the replicas do not have to make an authenticity and integrity check on the timestamp, as the Lamport clock is stored as part of the metadata in enclave memory. Upon receiving a majority of the remote timestamps (including its own locally stored timestamp), the coordinator creates the timestamp of the new `Put`, by incrementing the highest of the received timestamps and concatenating its own node-id. Finally, it broadcasts the new KV pair along with its new timestamp to all replicas, which insert the KV pair into their in-memory KVS. Since the put operation does not return the value to the user, and the meta data is protected by the enclave AVOCADO does not have to check the authenticity and integrity of the old value. Upon gathering a majority of acknowledgements it reports completion to the client.

#II: Get. The `Get` operation is similar to `Put`; the client sends its request to a randomly selected server, which coordinates it. The server then looks up the KV-pair in its local store.

The chosen request coordinator executes one broadcast round. In certain cases a second, optional broadcast round is required. Similarly, to the first round of a `Put`, the first round of a `Get` finds out the highest timestamp for that key when the majority of replicas has responded. This action guarantees that the `Get` will observe any completed `Put` (recall that a `Put` only completes if it reaches a majority of replicas). The replicas will respond with their locally stored value and corresponding Lamport timestamp to the coordinator, this involves a lookup in the local KVS and decryption together with integrity and authenticity checks of the value. The `Get` always returns to the client the value that corresponds to the highest timestamp found in its first round. However, the coordinator can reply to the client iff, based on the replies it received on its first round, it can guarantee that a majority of replicas are aware of this value. Otherwise it must perform a second broadcast round.

The second broadcast round is identical to the second write of a `Put`: it shares the KV-pair along with its timestamp

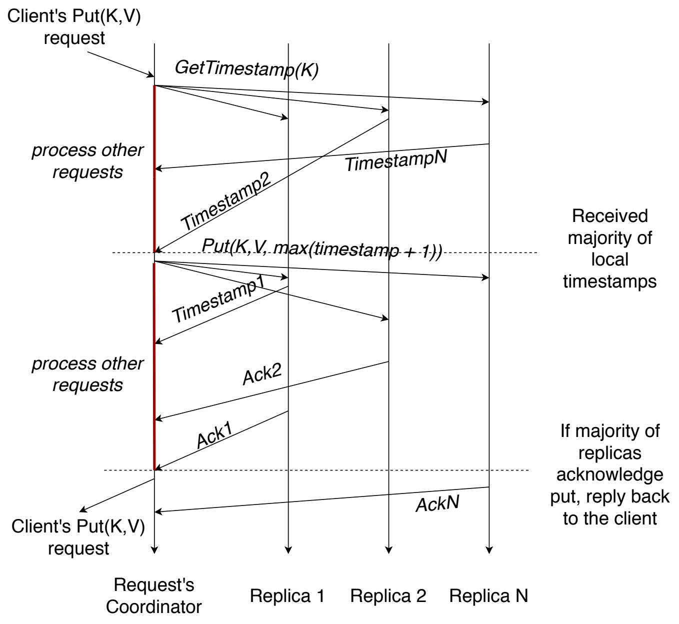
Client's Put(K,V) request
GetTimestamp(K)
process other requests
TimestampN
Timestamp2
Received majority of local timestamps
Put(K,V, max(timestamp + 1))
Timestamp1
process other requests
Ack2
Ack1
If majority of replicas acknowledge put, reply back to the client
AckN
Client's Put(K,V) request
Request's Coordinator
Replica 1
Replica 2
Replica N

Figure 4: Example of `Put` request in Avocado protocol.

with all replicas. Completion of the `Get` is reported to the client only after gathering a majority of acks. The second round of the `Get`, ensures that a `Get` not only observes the latest completed `Put`, but also guaranteeing that the `Put` will be visible to all subsequent `Gets`.

#III Delete. Delete is supported by issuing a `Put` operation with an empty value. This will remove the value from the KVS, but importantly it will not remove the key. We need to keep the key and the corresponding Lamport clock, to be able to establish an order of operation if a future `Put` operation accesses the same key.

#IV: Fault tolerance. Avocado remains highly available in the face of machine failures. However, as nodes fail, new nodes must be added, to ensure that the deployment always includes a majority of live nodes. In order to ensure that machines can safely join the configuration, we deploy a recovery algorithm inspired by Hermes [47].

Specifically, when adding a new node all other live replicas are notified of the new node's intention to join the replica group. The new node starts operating as a *shadow replica* that participates in all Put-related broadcast rounds (of remote replicas), but it cannot yet become the coordinator of a client request. Furthermore, the shadow replica does not take part in the Get quorum. In the meantime, the shadow replica reads chunks (multiple keys) from other replicas to fetch the latest values and reconstruct the KVS. To archive this the shadow replica is using the first broadcast round of ABD, but it never executes the second round, because it does not need to notify other replicas of what it read. After reading the entire KVS, the shadow replica is up-to-date and transitions to operational state, whereby it is able to serve client requests.

Result. We compare Avocado against BFT and Raft in Section 6.2. Our evaluation shows that Avocado is between 4.5 and 65× faster than BFT-Smart [87].

4.3 Configuration and attestation service

Problem. To ensure the integrity of the code and data deployed in the remote hosts with TEEs, TEEs, such as Intel SGX, provide attestation mechanisms. Secrets (e.g. certificates, encryption keys, etc.) are provided only after the attestation. Once an enclave is initialized, an attestation process can be launched to verify the integrity of code and data inside the enclave and proves the enclaves identity to a remote party.

Intel SGX uses an architecture Platform Service Enclave (PSE) called *Quoting Enclave* to sign the report of the loaded enclave [11, 27]. The remote verifier forwards this signed report to the Intel Attestation Service (IAS). Thereafter, IAS confirms or refuses the authenticity of the report to the verifier.

This conventional attestation mechanism using IAS incurs significant overhead in a distributed setting, especially for elastic computing or fault tolerance. The reason is that every time a distributed system (e.g. a distributed KVS) spawns a new enclave, it needs to perform the remote attestation via IAS which is not necessarily hosted in the same data center, incurring high latency. Lastly, and importantly, cloud providers usually do not want to disclosure their hardware or cluster information, as this information might be confidential.

Solution. In Avocado, we overcome this challenge by designing a decentralized configuration and attestation management system (CAS) for distributed SGX-based applications.

By consolidating and expanding the traditional attestation mechanism of Intel to build our CAS, we automatically and transparently perform the attestation for each node. The key idea behind our design is that we replace the Quoting Enclave in the Intel attestation mechanism by the LAS. The CAS first attests the LAS using the Intel attestation mechanism, thereafter the LAS will operate as the root of trust in our remote attestation mechanism. Note, that we can launch as many LAS instances as required for availability. The LAS performs the local attestation for Avocado nodes and provides attestation quotes that can be verified by the CAS. Thus, our mechanism does not need to interact with IAS after the LAS is trusted, this reduces significantly the overhead of the traditional attestation. We achieve the *transparent and automatic properties* by deeply embedding the remote attestation into the Avocado runtime. In addition, our CAS only provisions a configuration and secrets to execute Avocado once it ensures that all nodes were not manipulated. Each node of Avocado can only communicate with others if it can provide a valid certificate provided by our CAS. Therefore, users can just rely on the CAS to control and operate other components of Avocado. They only have to attest our CAS before providing secrets to it. The CAS itself also runs inside an enclave, thus users can use the traditional attestation method of Intel to validate it.

Result. As shown in 6.3 our CAS achieves 18.2× lower end-to-end latency in Avocado when comparing with IAS.

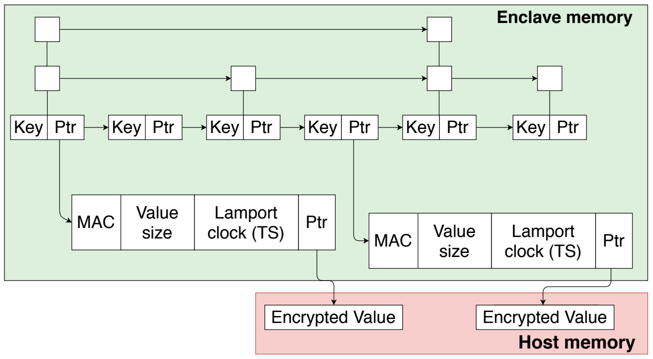

Figure 5: AVOCADO's single-node KVS.

4.4 Single-node KVS

Problem. Enclave's memory limitation is in stark contrast to the requirements of designing an in-memory KVS, which requires fast access to large amounts of in-memory data. Unfortunately, enclaves provide only limited physical memory (94 MiB) and incur high overheads due to the EPC paging mechanism (2 – 2000× [51]) beyond it.

To overcome the limitations of the strawman design, ShieldStore [48], a state-of-the-art secure in-memory KVS for a single-node system, proposed a MerkleTree-like data structure design where the entire KVS resides in the untrusted host memory, except for the security metadata (hash buckets heads). The metadata stored in the enclave memory is used to speed up the look up and to perform authenticity and integrity checks on the KV pairs. However, in our experience, the ShieldStore design suffers from continuous integrity re-calculations. Furthermore, the memory layout prohibits efficient concurrent operations.

Compared to Speicher [16], which also introduced the KV pair separation scheme for enclaves, our KVS is optimized for paging by encountering locality.

Solution. To overcome these limitations, we designed and propose our own in-memory concurrent data structure for the single-node KVS. Our KVS is based on a authenticated and confidentiality-preserving skip list [74] which supports secure and fast updates and lookups. We have chosen skip list as our fundamental data structure because it maintains the best features of a sorted array for searching ($O(log_n)$) and of a linked list-like structure for insertion ($O(log_n)$). Our design carefully partitions the key and value space by placing the keys along with metadata inside the enclave memory, while storing the bulk of data encrypted and integrity and authenticity protected in the untrusted host memory. Our partitioned data structure (keys and values) allow for faster lookups than ShieldStore's hash buckets, while it also reduces the amount of necessary calculations to guarantee the integrity and authenticity. Furthermore, our lock-free data structure supports concurrent operations and it is well-suited for increased parallelism.

As shown in Figure 5 the nodes of the skip list reside inside the enclave and contain the key and a pointer to metadata structure. This structure contains the 16 B MAC, for guaranteeing the integrity and authenticity of the value. Furthermore, the data structure also includes the size of the value, which makes checks on the value easier, since we do not need to read any information from the untrusted host memory, to retrieve how many bytes should be read. AVOCADO's consistency protocol uses logical clocks, i.e. Lamport clocks, to establish an order of operations on each key (§ 4.2). Therefore, we also store the Lamport clock in the corresponding metadata block, to prevent costly decryptions on the timestamp queries. Lastly, the metadata structure also stores the pointer to the value in the untrusted host memory.

Importantly, separating the metadata and the bulk data (i.e. values) from the skip list allows us to update the skip list *lock free*. Further, it also decreases the EPC pressure when doing a lookup, as nodes can be stored more compact and the metadata can be stored on a different page. However, looking up a value mandates an additional indirection due to keys and metadata separation. Nevertheless, we believe that updating the KVS without acquiring any locks is worth this additional indirection as it allows better multi-threaded scalability. Therefore, in contrast to a HashBucket design like ShieldStore, we never need to stall. In contrast to ShieldStore our approach seems to be limited by the enclave memory, however, assuming we have 1 KiB values and 16 B keys, we achieve a space reduction for enclave memory of 92.8 % compared to a naive implementation. Further, SGX provides a paging mechanism significantly increasing the available trusted memory, therefore increasing the possible size of the KVS. While SGX-paging incurs a high overhead, often accessed keys will eventually resided in EPC.

Result. Our evaluation in Section 6.4 confirms that our AVOCADO single-node KVS is scalable and more performant; the speedup of AVOCADO single-node KVS compared to ShieldStore increases from 1.6× in a single threaded benchmark to 5× when utilizing all 8 available CPU threads.

5 Implementation

5.1 System components

AVOCADO network stack. The AVOCADO network stack is based on eRPC [46] and DPDK [4]. In particular, we leverage SCONE to build both eRPC and DPDK. We also assure that the device DMA mappings resides in the host memory. The changes to implement the mappings amount to 154 new LoC and 81 removed LoC.

To run eRPC inside the enclave, we accordingly modify the hugepages allocation mechanism *(a)* to ensure that all network buffers reside in the host memory, *(b)* to fix a bug regarding the hugepages' detection, and *(c)* to alter how the address of the memory region is calculated. We also replace eRPC's allocation algorithm with our own allocator. We notice that the eRPC's native allocation algorithm, which allocates double the space of the previous allocation, quickly reserves all available memory in AVOCADO. Our memory allocator is less aggressive and allows us to use our servers' limited huge page memory more efficiently. In total, 80 LoC are added to eRPC, while 28 LoC are removed.

eRPC provides us with its own implementation of the UDP protocol. To secure the network communication, on top of the layer protocol, we use a modified OpenSSL [6] version. These changes allow us to randomly access the encrypted data. We added 55 LoC to OpenSSL. Further, we added another 287 LoC for a shared en-/decryption layer for the Avocado single-node KVS store and networking. Lastly, we further extend the shared layer to well-fit with the message format. This adds another 205 LoC.

Avocado replication layer. We implement Avocado replication layer in C++ on top of the Avocado network stack (2,743 LoC). We implement the protocol from scratch using the eRPC networking library across Avocado's different layers, i.e., replication and networking layer.

Configuration and attestation service. We implement Avocado CAS in Rust [59] for better memory safety (22,730 LoC). To run the CAS inside the Intel SGX, we use Scone since it transparently supports Rust applications. We make use of an encrypted embedded SQLite [9] to maintain configurations and secrets of Avocado inside Avocado CAS. To setup the configuration and bootstrap process, we provide configurations scripts, in Bash and Python 3, in total these bootstrap scripts require 709 LoC.

Avocado single-node KVS. We implement the Avocado single-node KVS based on a skip list based partitioned data structure. Particularly, Avocado single-node KVS extends Folly's ConcurrentSkiplist [32]. We ported the Folly library to Scone, which resulted to 167 new LoC and 40,394 removed LoC. In addition, the implementation requires another 190 LoC for the integration of the Boost library [19] to Scone. Further, we implement an efficient host memory allocator (388 LoC) for our skip list. We share en-/decryption layer based on OpenSSL [6] with the network stack.

5.2 Optimizations

[O1] Remove duplicated en-/decryptions. In Avocado, we use a shared encryption key between all replicas for the network operations. This allows us to replace some encryption calls with memory copies, as we can send the same packets to all replicas without costly re-encrypting the messages. However, this optimization is an optional trade-off between security and performance since one compromised enclave would compromise the entire system.

[O2] Remove locks. Separating the metadata from the key allows us to make atomic updates to the skip list, avoiding expensive locks. However, it also allows us to retire values earlier to the host memory; thereby reducing the EPC pressure since the metadata can already be written without being visible to other calls. Further, our host memory allocator supports lock-free operations on our skip list by providing similar atomic allocation and de-allocation primitives.

[O3] Limited number of message buffers. We design a rate limiter to allow all current running requests to finish without having to wait for the available resources. While we mostly implement it to prevent eRPC from running out of `hugepages` memory, we also find that it also reduces the stalls between accepting and completing a request.

6 Evaluation

Our evaluation answers the following questions.

- How does the Avocado network stack perform compared to the alternative networking approaches? (§ 6.1)
- How does the Avocado replication layer compare with alternative protocols (Raft [64] & BFT [87])? (§ 6.2)
- What are the performance overheads of Avocado CAS and how it compares with Intel's IAS [3]? (§ 6.3)
- How does the Avocado single-node KVS perform compared to ShieldStore [48]? (§ 6.4)
- What are the overall performance overheads of Avocado KVS? (§ 6.5)
- How does Avocado scale with increasing number of nodes? (§ 6.6)

Testbed. We perform all of our experiments on real hardware using a cluster of 5 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), NIC: Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02). They are connected over a 40GbE QSFP+ network switch.

Benchmarks. For the evaluation, we use the YCSB benchmark [7, 24] with different read/write ratios. Client-server communication over the network is prohibitively expensive from within an enclave (see § 4.1). Therefore, we stress-test the performance of Avocado by generating the workload within the enclave. This is the worst-case scenario for our system, since a client-server setting will show negligible latency/throughput overheads, due to client-server communication being the bottleneck. We configured Avocado to use a shared network key between the replicas (§ 5.2[O1]). For evaluating the network stack, we use `iperf` [41].

6.1 Network stack

Baselines and setup. We evaluate the performance of the Avocado network stack against three competitive baselines: `eRPC-native`, `sockets-native`, and `sockets-SCONE`. Note that Scone uses asynchronous syscalls [85] for performance improvements. Further, note that the native (eRPC and sockets) versions do not provide any security.

For the sockets (native and Scone), we use `iperf` to measure the throughput. For `eRPC-native` and Avocado network stack, we implement a simple server-client model on top of eRPC to simulate `iperf`'s behavior.

In our experiments, we compare the performance with different number of packet sizes, while keeping the number of threads fixed to 4. Note that eRPC supports only UDP while

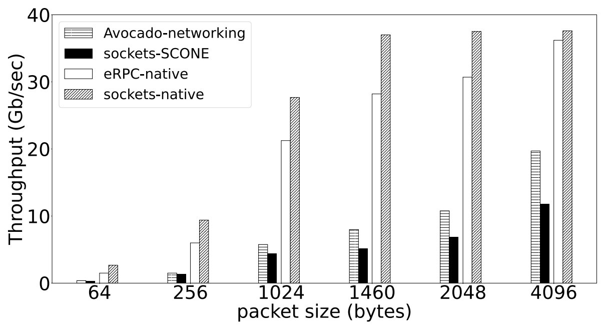

Figure 6: Performance comparison of Avocado network stack, `eRPC-native`, `sockets-native`, and `sockets-SCONE` for different packet sizes.

`iperf` supports both TCP and UDP. In our servers, we found that TCP performs better than UDP, so we report `iperf`'s TCP measurements since a designer could always benchmark both protocols and choose the most performant one.

Results. Figure 6 shows that `eRPC-native` is comparable to `sockets-native`. The reason is that TCP is optimized for high speed bulk transfers while UDP is optimized for low latency in the Linux kernel. This has an impact on buffer sizes and how data is polled and handed over. In addition to this, processing of the entire TCP/IP stack is frequently offloaded to the network controller.

Based on Figure 6 we deduce two core conclusions; (a) Scone's overhead is not negligible—Scone performance degrades ~ 4× and ~ 8× compared to Avocado network stack and `sockets-SCONE`, respectively; and (b), due to the number of system calls the sockets' layer is executing, Avocado network stack in the context of the secure enclave performs up to 1.66× faster than `sockets-SCONE`. As discussed, enclave exits and data copies in and out of the enclave deteriorate sockets' performance. This is further supported by the fact that the bigger the packet size is, the worse the performance becomes. Therefore, `sockets-SCONE` is a poor design choice as far as our requirement is concerned, and it justifies our design of the Avocado network stack.

6.2 Replication protocol

Baselines and setup. We show our system's end-to-end performance in comparison with two state-of-the-art protocols: (a) BFT (BFT-Smart [87]) for the Byzantine setting, and (b) Raft (eRPC-Raft [1]) for the non-Byzantine setting. To the best of our knowledge, there is no secure distributed in-memory KVS; BFT-Smart KVS is the closest baseline in terms of security properties for Byzantine environments, but BFT protocols (or BFT-Smart) still do not preserve confidentiality. Additionally, we compare Avocado against eRPC-Raft since it is also built on top of eRPC. This comparison aims to demonstrate the efficacy of eRPC. We compare them with a native version of Avocado, Avocado-native, which runs without TEEs. We compare Avocado and BFT along three parameters, as shown in Figure 7; *(i)* different read/write ratios, *(ii)* different value sizes and *(iii)* different workload threads per machine. We evaluate using the YCSB benchmark [7, 24]. Similarly, we compare Avocado against Raft implemented with eRPC. Note that eRPC-Raft is limited to only `PUT` requests and 1 workload thread in total.

	kOp/s	**Speedup**
Avocado	96	5.05×
eRPC-Raft	19	

Table 2: Performance comparison between Avocado and eRPC-Raft under a 100% W workload and a single client.

Results. Our evaluation shows that Avocado can achieve 4.5× to 65× more operations per second compared to BFT. Our Avocado presents similar performance to all four workloads, deducting that it is equivalently performant to both read and write heavy workloads. In addition, we notice that striving for the strictest security guarantees can decrease the performance to half compared to a native, unsecure version of Avocado.

Furthermore, we observe that the value size has great impact in the end-to-end performance. For instance, even in a read-heavy workload with value size to be equal to 256 B, the performance of Avocado is 6× higher compared to BFT and 1.83× lower than the native version. However, for value size to be equal to 1024 B , Avocado is 20% slower than BFT and 9× slower than Avocado-native. Similarly, for value size to be equal to 4096 B, Avocado is 1.25× faster than BFT and 3.65× slower than Avocado-native. We discuss the effects of value size on Avocado further in section § 6.5. Lastly, Avocado scales up with the number of threads; Avocado achieves 38% more operations when the number of threads is increased from 4 to 8 threads. Due to the limitation with the amount of threads inside the enclave, Avocado cannot be executed with 16 threads.

Lastly, we compare eRPC-Raft against Avocado. Avocado under the same settings outperforms eRPC-Raft for 4.8× as shown in Table 2. The reason is that eRPC-Raft does process requests asynchronously while in Avocado the time required for the necessary replicas to respond overlaps with processing any outstanding requests.

6.3 Configuration and attestation service

Baseline and setup. To evaluate the advantage of the attestation mechanism using Avocado CAS in comparison to the traditional attestation mechanism of Intel using IAS, we conduct an experiment to measure the end-to-end latency of the attestation process using both mechanisms.

Results. The attestation using Avocado CAS achieves a speedup of 18.2× compared to the traditional mechanism using IAS (see Table 3). The mechanism using Avocado CAS performs the attestation via LAN connections, since Avocado CAS is deployed in the same cluster as Avocado instances. Meanwhile, the mechanism using IAS performs the attestation via WAN connections since it requires to verify the quotes using IAS that is deployed at Intel. Furthermore, Avocado CAS

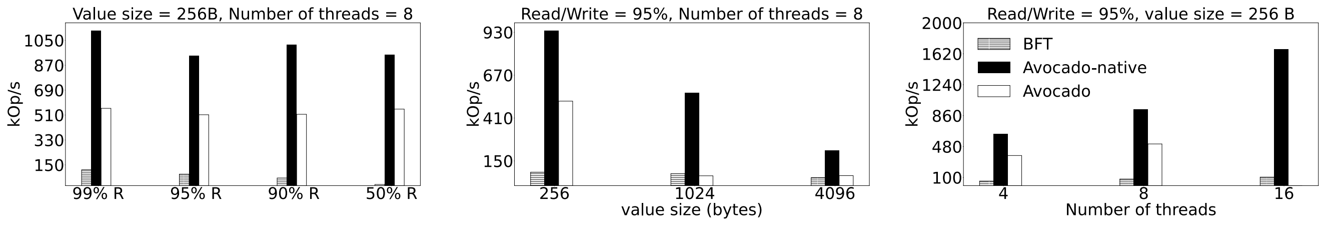

Figure 7: End-to-end performance comparison between AVOCADO, AVOCADO-native and BFT for different types of workloads, value sizes and number of threads.

	Mean / s	SD / s	Speedup
AVOCADO CAS	0.169	0.0195	18.2×
IAS	2.913	0.177	

Table 3: The end-to-end latency comparison between the attestation mechanisms using AVOCADO CAS and IAS.

provides transparently provides configuration management features to operate in an distributed environment.

6.4 Single-node KVS

Baselines and setup. We compare our AVOCADO single-node KVS against ShieldStore [48], a state-of-the-art secure in-memory KVS for a single-node system. For the single-node system evaluation, we use Intel(R) Core(TM) i7-8565U CPU of 8 logical threads and 16 GiB memory. This is due to ShieldStore's dependencies on specific versions of OS, linux-SGX [82] and `tcmalloc` [58], we are not able to run it on our servers. We evaluate AVOCADO single-node KVS and ShieldStore across three dimensions using the YCSB workload: threads (Figure 8), value size and read-write ratios (Figure 9).

Results. Figure 8 shows the scaling capabilities of our AVOCADO single-node KVS vs. ShieldStore for two different value sizes. Our AVOCADO single-node KVS, for all number of threads, is 1.6× to 9.5× faster than ShieldStore. Regarding the value size, we observe that ShieldStore's performance is highly affected when the value size is increased. For example, with 2 threads, ShieldStore presents a performance degradation of 7.3× from 16 B to 1024 B while the same scenario deteriorates AVOCADO single-node KVS's performance only by 1.23×. We deduct this to the fact that Shieldstore searches require decrypting the concatenated entry of the key and the value in each visited bucket. As a result, bigger value sizes increase the cipher text that needs to be decrypted leading to higher performance costs. In contrast, AVOCADO stores keys inside protected area and search time is not affected by value sizes and decryption cost.

We observe similar behavior as the number of threads increases. ShieldStore is designed to avoid synchronization costs between threads that are matched to different KVS's areas. However, to achieve this, ShieldStore requires to sort and distribute (through hashing) requests across threads which adds overheads compared to AVOCADO single-node KVS. Specifically, we show that for value size equal to 16 B AVOCADO single-node KVS is 1.6×, 3× and 5× faster than ShieldStore when using

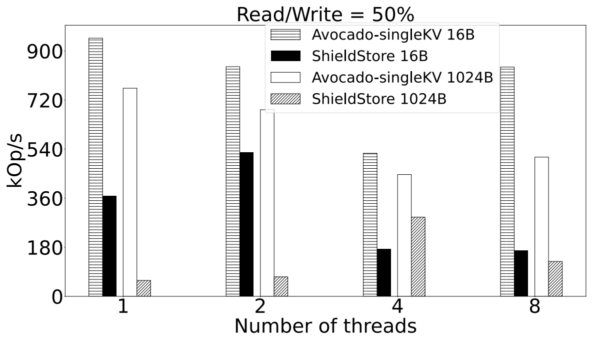

Figure 8: Performance comparison between AVOCADO single-node KVS and ShieldStore under a 50R-50W workload for varying number of threads.

2, 4 and 8 threads, respectively. Additionally, for value size equal to 1024 B, AVOCADO single-node KVS is 9.5×, 1.5× and 4× faster than ShieldStore with 2, 4 and 8 threads, respectively.

However, we find that both AVOCADO single-node KVS and ShieldStore have a performance drop, when the number of threads is increased. This trend is visible until the number of threads exceeds the number of physical cores. We attribute this behavior to two main reasons. Firstly, the CPU we are running this experiment on, is a lower power CPU, with a low base frequency (1.8 GHz) but a relatively high turbo frequency (4.8 GHz). This high turbo frequency cannot be reached with a high number of threads running, giving a performance boost to low thread numbers, compared to higher thread numbers. Secondly, increasing the number of threads results in a higher rate of cache misses, due to other cores having requested different memory, or having to invalidate the cache lines in lower level caches i.e. L1 and L2. This effect is especially pronounced in a write heavy workload, as presented in Figure 8. Increasing the number of threads further, from number of physical cores to logical cores, allows the CPU to schedule a different thread, while it is waiting for a memory access to complete, explaining the increase of performance with 8 threads.

Secondly, we also study how AVOCADO single-node KVS and ShieldStore perform under different value sizes and different read-write ratios. In particular, Figure 9 compares the two Key-Value (KV) stores against three different workloads and varying value sizes, where we fix the number of threads across the experiments to 4. For all three workloads as shown in Figure 9, AVOCADO single-node KVS achieves better performance than ShieldStore; 3.63× to 2.97× faster when value size is equal

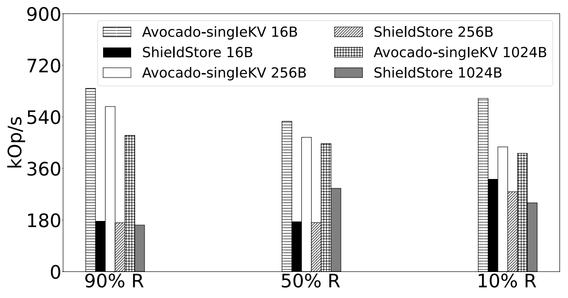

Figure 9: Performance comparison between Avocado single-node KVS and Shieldstore under three different workloads for varying value sizes.

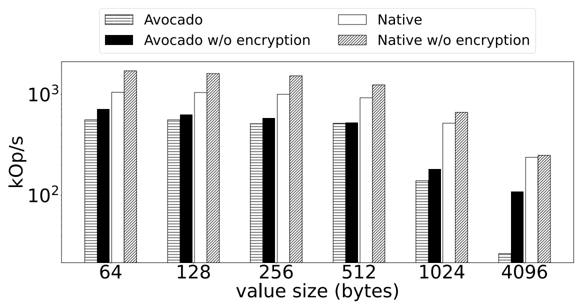

Figure 10: Performance comparison of Avocado with and without network and KVS encryption, inside and outside of the enclave, with different value sizes with 95 % reads and 8 threads per machine.

to 16 B, 3× to 1.53× faster when value size is equal to 256 B and 1.87× to 1.56× faster when value size is equal to 1024 B.

Lastly, we conclude that Avocado single-node KVS is better in read-dominant workload (90% reads) than in write-heavy workload (90% writes), since Avocado single-node KVS achieves ~ 5% to ~ 30% better performance.

6.5 Distributed KVS

Baselines and setup. We evaluate the overhead Avocado incurs from running inside an enclave, against running Avocado natively, i.e. without SGX. Furthermore, we also evaluate the encryption overheads for in-memory KVS and network traffic. Thus, we compare Avocado with encryption activated and deactivated against the native KVS. Both with encryption for the KVS and network enabled and disabled. We run the YCSB benchmark with 95 % reads with 5 machines and 8 threads per machine, with different value sizes.

Results. Figure 10 shows the performance influence of SGX and encryption on Avocado. The value size comparison shows that for small values, i.e. values under <1 KiB, Avocado reaches around half, between 51.2 and 56.0 % of the performance compared to the native KVS. However, with bigger value sizes the difference becomes greater with a slowdown of 3.7× and 9.0×, for 1 or 4 KiB respectively. The sudden drop in performance compared to the native case is mainly due to two reasons: firstly, eRPC has to acquire a lock

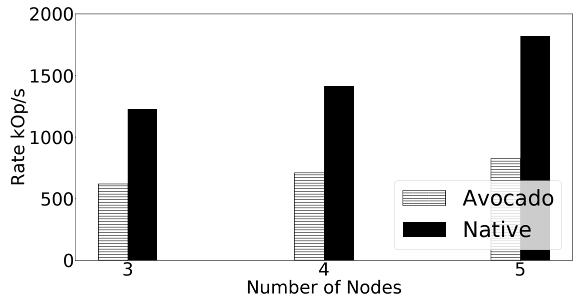

Figure 11: Performance of Avocado inside and outside of the enclave running on different number of nodes with a value size of 256 B, 95% reads and 8 threads per machine.

when allocating DMA for bigger packages size than the MTU, which is configured in our case to 1 KiB. While this penalizes native and Avocado, it is worse for Avocado, since this could result in an enclave exit for yielding. Additionally, with bigger value sizes, it gets more likely that we have to evict a page from the EPC, when inspecting the network traffic. This might be addressed with a memory buffer, which is reused for all data transfer between untrusted host memory and EPC memory. Due to constant accessing of this buffer, it should rarely get paged out the enclave.

The comparison also shows that encrypting the in-memory KVS and network traffic adds up to 62 % overhead for small values and 4.6 % for large values in the native case. However, we observe a different behavior for Avocado. The overhead for small values is more moderate compared to native with around 25 %. However, the overhead does not get smaller with bigger values sizes. In contrast, it peaked with a values size of 4 KiB with 4.1×, which is due to EPC paging.

In these experiment we also observed a mean latency of 66 µs. This latency was reached in a fully stressed system. Due to SGX requiring us to make a syscall for taking a timestamp detailed latency analysis was impractical, as the syscall overhead together with the world switch would have easily dominated the benchmark.

6.6 Scalability

Baselines and setup. We evaluated the scalability of Avocado by running it inside and outside (natively) the enclave on a varying number of nodes. We run the YSCB benchmark with 95 % reads on 3, 4 and 5 machines and 8 threads per machine, with a fixed value size of 256 B.

Results. Figure 11 shows the scalability numbers for different number of nodes. We are limited in our cluster to 5 nodes. The evaluation shows that replicating the KVS over 5 nodes instead of 3 increases the throughput of the native solution by 49 % and 33 % for Avocado.

6.7 Number of keys

Baselines and setup. We measure the performance of Avocado with increasing number of distinct keys. We run the YCSB benchmark with two different distributions, i.e.

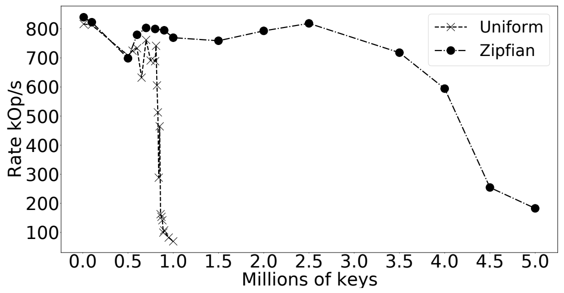

Figure 12: Performance of Avocado inside the enclave running with different number of distinct keys, with a uniform and Zipfian ($\alpha = 1.0$) distribution with a value size of 256 B, 95% reads and 8 threads per machine

uniform and Zipfian, on 5 machines with 8 threads and 95 % reads and a fixed value size of 256 B.

Results. Figure 12 shows the throughput of Avocado with different distribution. Both distributions have a similar performance until $700k$ keys, where the performance of the uniform distribution starts to suffer greatly, due to SGX paging overheads. The uniform distribution prevents efficient caching from SGX in the EPC, since it does not generate any hot keys. In a uniform distribution, Avocado is therefore restricted by the EPC size. However, this could be alleviated with a multi-level lookup, which stores the lower levels in the host memory.

On the other hand, if the data set is not uniformly distributed Avocado can take advantage of the caching of EPC and extend the number of supported keys. In our experiments Avocado throughput was stable until $3.5M$ keys in the Zipfian distribution before it starts to suffer from the paging overheads. Similar to the uniform distribution, a multi-level lookup could reduce the paging overheads.

7 Related work

Shielded execution. With the adoption of TEEs in the cloud, shielded execution frameworks are being adopted to provide strong security properties for unmodified/legacy applications running in the untrusted environment [13, 17, 35, 93]. These frameworks promote portability, programmability and good performance. As a result, they have been used to implement a wide-range of secure systems for storage [16, 50], data analytics [80], databases [73], distributed systems [90], FaaS [92], network functions [91], machine learning [75], profilers [15], etc. Our Avocado project leverages the advancements in shielded execution frameworks; in particular, we use Scone [13] to build a distributed storage system.

Networking for shielded execution. Shielded execution frameworks, like Scone [13] and Eleos [65] provide an asynchronous system call API [85]. However, the asynchronous syscall mechanism incurs high overheads (due to data copies) and latency. ShieldBox [91] alleviates the issue of asynchronous syscalls by using DPDK [4] as polling user mode driver for a secure middlebox. Unfortunately, Shieldbox network stack targets only layer 2 in the OSI model, and therefore, it is incompatible with the RPC layer required for a distributed KVS. rkt-io provides a library OS in the enclave, and can therefore provide a full network stack [89].

Secure storage systems. Secure storage is an important theme for cloud computing systems. A wide-range of systems have been proposed in the community based on different hardware with varying security guarantees and storage interfaces: KVS [16, 48, 50], filesystems [33, 96, 103], databases [28, 67, 70, 73, 95], etc. In contrast to these existing systems, Avocado proposes a scalable distributed in-memory KV store instead of a single-node system.

For distributed storage system design, Depot [57] and Salus [102] propose secure distributed storages, which provide consistency, durability, availability and integrity. A2M [21] is also robust against Byzantine faults. On top of those properties, Avocado also offers confidentiality. CloudProof [69] completely distrusts the cloud provider. However, CloudProof requires the client to guarantee these security properties, which requires major changes to the client, which isn't required by Avocado. Furthermore, since our work leverages hardware-assisted shielded execution as the root of trust, we do not need a trusted proxy, which limits the scalability of the system.

8 Conclusion

We present an approach to build a secure, high-performance in-memory distributed KVS system for untrusted cloud environments using TEEs. Our design includes four core contributions involving TEEs in a distributed environment: (a) the first direct I/O RPC network stack for TEEs based on eRPC with the complete support for transport and session layers; (b) a secure replication protocol based on hardening of a non-Byzantine protocol, where we transform a Byzantine behavior to a faulty behavior using TEEs; (c) a configuration and attestation service to seamlessly extend the trust from a single-node TEE to the distributed environment; (d) a secure in-memory single-node KVS based on a novel partitioned Skiplist data structure, and show that it is well-suited to overcome the memory limitations and support lock-free scalable concurrent updates in the TEEs.

Importantly, we set out to build a practical system without compromising performance—the literature distinctly shows that BFT protocols are typically not adopted in practice due to their high overheads [22, 71]. In contrast to BFT, our system provides stronger security properties (also preserves confidentiality) and improved performance (4.5× to 65×), while using f fewer replicas.

Software artifact. Our project is publicly available: `https://github.com/mbailleu/avocado`.

Acknowledgements. We thank our shepherd, Yu Hua, and the anonymous reviewers for their helpful comments. This work was supported in parts by a UK RISE Grant from NCSC/GCHQ, a Huawei Research Grant, and a Microsoft Research PhD Fellowship.

References

[1] eRPC-Raft. https://github.com/erpc-io/eRPC/tree/master/apps/smr. Last accessed: Jan, 2021.

[2] 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.

[3] 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.

[4] Intel DPDK. http://dpdk.org/. Last accessed: Jan, 2021.

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

[6] OpenSSL library. https://openssl.org. Last accessed: Jan, 2021.

[7] YCSB. https://github.com/brianfrankcooper/YCSB. Last accessed: Jan, 2021.

[8] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):159–174, Oct. 2007.

[9] G. Allen and M. Owens. *The Definitive Guide to SQLite*. Apress, 2010.

[10] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. Last accessed: Jan, 2021.

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

[12] 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.

[13] 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.

[14] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1), Jan. 1995.

[15] 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.

[16] 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.

[17] 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.

[18] J. Behl, T. Distler, and R. Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[19] boost: C++ libraries. https://www.boost.org/. Last accessed: Aug, 2020.

[20] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[21] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[22] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Bft: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.

[23] 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.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010.

[25] M. Correia, N. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, 2004.

[26] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Worm-IT - A Wormhole-Based Intrusion-Tolerant Group Communication System. 2007.

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

[28] 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.

[29] 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.

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

[31] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004.

[32] Folly: Facebook Open-source Library. `https://github.com/facebook/folly`.

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

[34] Introducing Google Cloud Confidential Computing with Confidential VMs. `https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms`.

[35] Asylo: An open and flexible framework for enclave applications. `https://asylo.dev/`.

[36] 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.

[37] 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.

[38] 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.

[39] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transaction of Programming Language and Systems (TOPLAS)*, 1990.

[40] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur-Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[41] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. `https://iperf.fr/`. Last accessed: Aug, 2020.

[42] 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.

[43] 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.

[44] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur-Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid)*, 2012.

[45] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *International Conference on Parallel Processing (ICPP)*, 2011.

[46] 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.

[47] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[53] 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.

[54] D. Levin, J. J. Douceur, J. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[55] 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.

[56] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FtCS)*, 1997.

[57] 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.

[58] TCMalloc. `https://github.com/google/tcmalloc`. Last accessed: Aug, 2020.

[59] N. D. Matsakis and F. S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*, 2014.

[60] Microsoft Azure. Azure confidential computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute/`. Last accessed: Jan, 2021.

[61] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[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] 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.

[64] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014.

[65] 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.

[66] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. 2015.

[67] 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.

[68] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, 2010.

[69] 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.

[70] 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.

[71] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, 2015.

[72] 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.

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

[74] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communication of ACM (CACM)*, 1990.

[75] 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.

[76] RISC-V. Keystone Open-source Secure Hardware Enclave. `https://keystone-enclave.org/`.

[77] 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.

[78] 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.

[79] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services . In *21st USENIX Security Symposium (USENIX Security)*, 2012.

[80] 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.

[81] 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.

[82] Intel Software Guard Extensions SDK for Linux. `https://01.org/intel-softwareguard-extensions`.

[83] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCCS)*, 2016.

[84] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[85] 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.

[86] 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.

[87] J. Sousa and A. Bessani. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *2012 Ninth European Dependable Computing Conference (EDCC)*, 2012.

[88] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and Atomic RDMA-Based Replication. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2018.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] S.-Y. Tsai and Y. Zhang. A Double-Edged Sword: Security Threats and Opportunities in One-Sided Network Communication. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[95] 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.

[96] 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.

[97] 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.

[98] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design &amp; Implementation - Volume 6 (OSDI)*, 2004.

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

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

[101] 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.

[102] 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.

[103] 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.

[104] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *SIGARCH Comput. Archit. News*, 2017.

[105] 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.

[106] S. Yang. SLIK : Scalable Low-Latency Indexes for a Key-Value Store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2014.