Precursor: A Fast, Client-Centric and Trusted Key-Value Store using RDMA and Intel SGX

Ines Messadi
messadi@ibr.cs.tu-bs.de
TU Braunschweig
Germany

Shivananda Neumann
TU Braunschweig
Germany

Nico Weichbrodt
TU Braunschweig
Germany

Lennart Almstedt
TU Braunschweig
Germany

Mohammad Mahhouk
TU Braunschweig
Germany

Rüdiger Kapitza
rrkapitz@ibr.cs.tu-bs.de
TU Braunschweig
Germany

ABSTRACT

As offered by the Intel Software Guard Extensions (SGX), trusted execution enables confidentiality and integrity for off-site deployed services. Thereby, securing key-value stores has received particular attention, as they are a building block for many complex applications to speed-up request processing. Initially, the developers' main design challenge has been to address the performance barriers of SGX. Besides, we identified the integration of a SGX-secured key-value store with recent network technologies, especially Remote Direct Memory Access (RDMA), as an essential emerging requirement. RDMA allows fast direct access to remote memory at high bandwidth. As SGX-protected memory cannot be directly accessed over the network, a fast exchange between the main and trusted memory must be enabled. More importantly, SGX-protected services can be expected to be CPU-bound as a result of the vast number of cryptographic operations required to transfer and store data securely.

In this paper, we present Precursor, a new key-value store design that utilizes trusted execution to offer confidentiality and integrity while relying on RDMA for low latency and high bandwidth communication. Precursor offloads cryptographic operations to the client-side where possible to prevent a server-side CPU bottleneck and reduces data movement in and out of the trusted execution environment to the bare minimum. Our evaluation shows that Precursor achieves up to 6-8.5 times higher throughput when compared against similar SGX-secured key-value store approaches.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **Systems security**.

KEYWORDS

Key-value stores, Intel SGX, RDMA

ACM Reference Format:
Ines Messadi, Shivananda Neumann, Nico Weichbrodt, Lennart Almstedt, Mohammad Mahhouk, and Rüdiger Kapitza. 2021. Precursor: A Fast, Client-Centric and Trusted Key-Value Store using RDMA and Intel SGX . In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Québec city, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3476129

1 INTRODUCTION

Outsourcing of services to remote infrastructures as provided by cloud computing has revolutionized the IT landscape. As a barrier to this trend, the loss of control over code and data has been a constant concern. With the widespread availability of trusted execution, as offered by SGX [26, 34], cloud users can retain control over their data and code as even administrative personnel and privileged software cannot access or undetectably modify a protected execution context. Accordingly, a great variety of services and applications have been secured using SGX [8, 13, 50]. A popular class of investigated applications has been key-values stores, as they build a core entity of many complex application deployments [6, 9, 15, 31, 55]. While generic approaches have been proposed to execute legacy binaries inside a trusted execution context [8, 46, 59], tailored variants have been devised as SGX imposes some hardware-based limitations that need to be addressed for providing a fast and scalable solution [6, 9, 31, 55]. First, there is the transition time between normal mode and trusted execution – which can be as costly as approximately 13,100 cycles [63]. Second, memory usage inside the trusted execution context can be an issue. If the active working set of all trusted execution contexts on a machine is larger than 93 MiB (more recently 188 MiB [1]), a custom software-based memory paging process is necessary. However, it severely degrades the performance [8, 13, 63]. In addition to these two barriers, we identified a third challenge. Data centers more and more utilize RDMA, as it enables high bandwidth at low latency [22]. Paired with a high line rate of up to 200 Gbit/s [2], key-value stores are vastly accelerated [27, 28, 36]. While they show great performance, all were implemented under the assumption of a fully trusted environment. At first glance, it seems natural to combine the use of RDMA and trusted execution in data centers to gain low-latency and high bandwidth for trusted services. However, two main obstacles can be identified.

First, the memory of a trusted execution context is not accessible from a remote location. Thus, data cannot be directly transferred from a client machine into an SGX-secured key-value store via RDMA.

Second, services running under the protection of trusted execution typically require heavy use of cryptographic operations. In particular, clients and the server need a secure exchange path. Storing data usually involves a custom server-side encryption scheme to circumvent costly software-based paging, as mentioned earlier [6, 9, 15, 31, 55]. Combined with the vast available bandwidth, the employed cryptographic operations eventually make services like key-value stores CPU-bound.

In this paper, we present Precursor, a key-value store that combines the use of RDMA for fast communication and SGX for increased security. Precursor focuses on the offloading of server-side encryption to the client-side. In particular, payload data is pre-encrypted on the client-side using one-time keys to enable the direct storing of encrypted data in the untrusted memory of Precursor. Overall this reduces cryptographic operations on the server-side. Additionally, Precursor features a design that avoids costly context switches where possible.

The remainder of this paper is organised around Precursor's motivation and contributions:

§2 introduces the main technologies behind Precursor, discusses the threat model, and states the identified problem;

§3 describes Precursor's design, explaining the critical design choice around shifting computation to clients, combining SGX and RDMA while mitigating the hardware limitations related to both technologies;

§4 gives details on how we implemented Precursor;

§5 evaluates Precursor's design in terms of throughput and latency. Using the YCSB benchmark [18], we can show that Precursor outperforms competitors like ShieldStore [31] with 6-8.5 times higher throughput for different workloads.

2 BACKGROUND AND MOTIVATION

In the following, we first explain how SGX secures execution in an untrusted environment. Second, we detail the essentials regarding RDMA that allow remote systems to communicate with sub-microsecond latencies. Finally, we outline the challenges that arise when combining the two technologies.

2.1 Intel Software Guard Extensions

Over the past few years, trusted execution support became prevalent on commodity platforms. Recent initiatives such as the Confidential Computing Consortium concentrate on propelling trusted execution to provide customers with proof of data integrity and confidentiality [19]. Indeed, a trusted execution environment (TEE) isolates security-sensitive code and data from an untrusted surrounding. Examples for implementations of trusted execution are ARM TrustZone [4], AMD SME/SEV [20, 25], and Intel SGX [5]. In this paper, we focus in particular on SGX because of its far-reaching protection goals and its emerging availability in cloud environments,[1] as compared to other trusted platforms.

[1] https://azure.microsoft.com/en-us/solutions/confidential-compute/

SGX enclaves. In late 2015, Intel released processors extended with a set of instructions, which create and control TEEs called SGX enclaves. Enclaves are isolated regions of the virtual address space protected from any untrusted entities and privileged software, including Direct Memory Access (DMA). For SGX-capable processors, the isolated code and data reside in an encrypted region of the physical memory known as the enclave page cache (EPC), typically limited to a size of 128 MiB. Yet, an application can only use ≈93 MiB, whereas the rest is reserved for security metadata [63]. Latest CPUs generations, i.e., Intel Ice Lake, have doubled this size to 256 MiB of EPC memory. In this case, ≈188 MiB can be directly utilized for hosting enclaves.

SGX hardware limitations. Despite its far-reaching protection, SGX has some limitations. If enclaves consume more memory than the actual EPC size, the operating system evicts EPC pages to the untrusted main memory, which incurs an overhead, roughly estimated to be 20K CPU cycles until the enclave execution can be continued [8]. Enclave transitions also introduce a substantial overhead compared to a regular application flow. Because invoking all kinds of system calls is prevented, the SGX SDK comprises special calls to the untrusted region. These are commonly known as *ocalls*, for a call gate to the untrusted environment, and *ecalls* to enter an enclave. For each defined function, the SDK marshals data between unprotected memory and the enclave. However, these transitional calls imply an overhead of ≈13K CPU cycles, for context switching, security checks, and TLB flushing [63]. When the data is persistently saved to the disk, SGX provides trusted time and monotonic counters to detect state rollback attacks and forking. In this regard, previous works propose different prevention techniques, which can be integrated into our design [9, 11].

2.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a technology that allows direct data exchange between networked machines, without interrupting remote CPUs.

With this, RDMA applications achieve noticeable networked performance increase and CPU load reduction, which is indeed meaningful for latency-sensitive services [29, 30, 33, 37]. Latest RDMA NICs have 200 Gbps of bandwidth, reach 2μs round-trip latency, and can process millions of messages per second [2]. RDMA's programming model works as follows: with each operation, the application starts with preparing buffers and gives access to a particular memory region; the operating system tags the region with an identifier and registers it with the NIC. As long as a process (remote or local) acquires the right permissions (read/write) and memory regions identifiers, it can access these regions. Following this, it can perform two modes of operations, which gives the developer the flexibility to select the most convenient method. The CPU bypassing mode, usually called *one-sided*, can be a remote read/write or atomic operation. The second mode, *two-sided*, called send/receive operation, resembles socket programming.

RDMA endpoints communicate by posting operations to asynchronous queue pairs. A send/receive queue to store requests and a completion queue to optionally notify about the I/O's final status operations. A work request represents a buffer location from where the NIC places data (DMA *write*) or reads (*DMA read*). A typical in-memory key-value store benefits the most from one-sided primitives [29, 37, 57, 62], which is why we choose them for Precursor.

2.3 Threat Model

Cloud environments are prone to security breaches and unauthorized disclosures [17, 42]. Indeed, the cloud is vulnerable to threats from a rogue administrator that might conduct illegal activities or even external attackers. We assume such adversaries can tamper with the Precursor server state in memory. To detect integrity violations and maintain confidentiality, we rely on Intel SGX enclaves and standard cryptographic operations. In principle, we trust the implementation of these mechanisms. Side-channel attacks [32, 51, 61, 65] that disrupt Intel SGX services are not prevented, but a defense technique, e.g., ASLR, can be applied to SGX applications including Precursor [12, 39, 52, 53]. We assume clients to be part of a secure and trusted environment in the cloud domain, i.e., they are also guarded by SGX or another TEE technology as in previous works (e.g., NeXUS) [21]. Availability threats such as crashing an SGX application are not of interest, as the hosting OS can stop enclave execution arbitrarily and at any time. The same applies to RDMA — denial of service attacks [43, 48, 54] are considered out of scope.

2.4 Problem Statement

Most early works that seek to ensure security properties—integrity, authenticity, and confidentiality—of clients' data processed at a remote data center relied heavily on cryptographic approaches (e.g., homomorphic encryption). These techniques introduce high computational complexity, which leads to a substantial overhead [45, 68]. With the ubiquity of hardware TEEs, current research uncovers more efficient ways of securing applications state.

When using TEEs, the straightforward idea is to put the entire application state into the trusted environment. With SGX, this incurs some complexity and additional efforts; enclave restrictions (i.e., due to prohibited system calls) obligate developers to rethink their systems' architecture. The alternative is running unmodified applications on top of a library operating system or similar system layers inside the TEEs [8, 10, 46, 59].

Although this clears out the dilemma of redesigning systems, the library OS approach approach has two downsides. First, it results in a sizeable trusted computing base (TCB), which increases the likelihood an attacker can exploit an enclave. Also, a large TCB risks moderate performance due to SGX restrictions, especially when we consider the capacity limit of the EPC [31]. Second, this encapsulation cannot be conveniently applied to applications utilizing RDMA because SGX prevents DMA access to enclave memory.

For better performance, SGX-tailored systems have been proposed. These systems tend to place encrypted data in the untrusted memory, while enclaves mainly serve to preserve integrity. Following this design path, key-value stores like ShieldStore [31] and EnclaveCache [15] but also the coordination service SecureKeeper [13] apply the following steps. First, transport encryption methods encrypt the data for transit protection between the client and a TEE on the server-side. Next, received requests get entirely copied inside the enclave. There, the payload gets decrypted, and its integrity is validated. To address the limited EPC, and avoid the performance penalty of overstepping the limit, the payload is re-encrypted and copied again to the untrusted memory.

However, this additional encryption step for storing the data outside of the enclave is costly. In general, such schemes use secure authenticated encryption methods, e.g., AES-GCM, to protect confidentiality, authenticity, and integrity of transport communication and data storage.

We conventionally denote this design as a *server encryption scheme*, because the server actively preserves and verifies payload integrity.

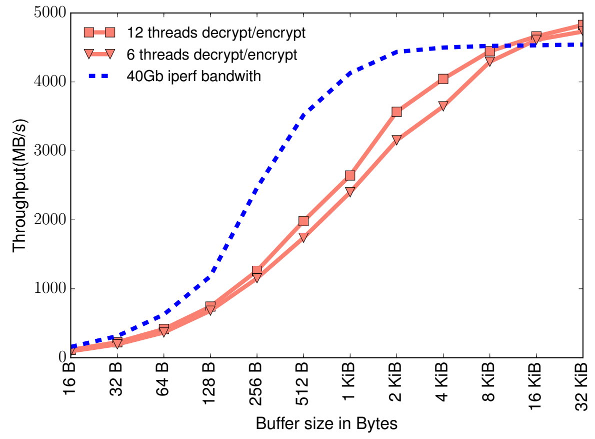


Figure 1: Throughput comparison of essential cryptographic operations needed for the server-side computation with the raw bandwidth of RDMA for 40 Gbit.

Figure 1 emphasizes this problem by approximating the necessary cryptographic operations of a key-value store that features a *server encryption scheme*. The method is as follows: within the enclave, a buffer—with variable size from 16 byte rising to 32 KiB—gets decrypted and then encrypted again. This behavior resembles how the encrypted transport payload gets decrypted for verification, and then encrypted again for storage as implemented by ShieldStore [31], EnclaveCache [15] and SecureKeeper [13].
We used a machine that has an Intel Xeon E3-1230 v5 CPU, equipped with 6 cores (12 hyper-threads) and the hardware accelerated functions AES-GCM of the SGX software development kit (SDK) version 2.9. We compare this measurement to the RDMA bandwidth limit of one-sided WRITEs using Mellanox Perftest [35] for a 40 Gbit/s link. We measured the cryptographic operations multi-threaded with 6 and 12 threads to reach the maximum throughput of the machine, with each thread assigned to a core and with hyper-threading enabled.

As can be seen for small packets (up to 1 KiB), the cryptographic operations cause 36% less throughput than the raw RDMA bandwidth. In a modern data center, speeds of 100 Gbit/s are already available and up to 200 Gbit/s have been announced [38]. Also, it has to be considered that the measured cryptographic operations are only a part of the performed processing. E.g., data might have to be additionally copied, and in the case of a design utilizing a Merkle tree, cryptographic hashes have to be computed frequently.

As a consequence of this result, we aim to reduce the cost of cryptographic computations, which are inherent to the *server encryption scheme*.

3 DESIGN OF PRECURSOR

3.1 Objectives

We propose a design for a key-value store that aims to benefit from the high bandwidth offered by network technologies such as RDMA. We specify here the properties that the Precursor design aims to satisfy, as well as the issues that we want to overcome:

*R*1 *Security and privacy.* A key-value store meant to run in the cloud must ensure the confidentiality and integrity of customers' data. Besides, Precursor adds little code to the enclave's TCB compared to library OS solutions to reduce exploitable vulnerabilities.

*R*2 *Mitigating SGX constraints.* Due to the restricted EPC, we design Precursor to mitigate the paging overhead by keeping a small memory footprint (§5.4). Furthermore, costly enclave transitions should be avoided where possible.

*R*3 *Reduce server-side cryptographic load.* Precursor targets to minimize server-side CPU demand caused by cryptographic operations by offloading them to the clients.

*R*4 *Use of data center network technology.* While previous SGX-secured key-value store implementations go through additional network processing in the traditional network stack, we aim to efficiently combine trusted execution and novel data center network support via RDMA.

3.2 Precursor in a Nutshell

Precursor ensures confidentiality and integrity through standard cryptographic algorithms and a server TEE instance but redistributes a critical share of the demanded computational workload. For the currently predominant *server encryption scheme*, we found out that cryptographic operations on the server-side can be a bottleneck, especially if high bandwidth technology such as RDMA is used (Section 2.4). In the proposed design, each client takes over more responsibility for its generated workload and the overall load is better distributed.

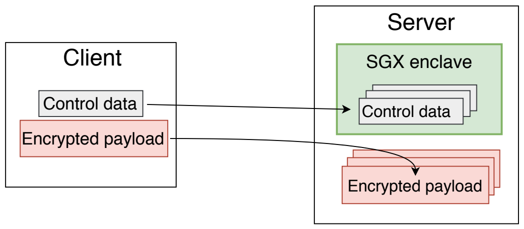


Figure 2: Basic approach of split transfer between control data and payload.

As highlighted in Figure 2, the core principle of Precursor is to split each request to the key-value store in two parts: the *control data* and the encrypted *payload data*. The control data is securely transferred using a transport encryption scheme. The secure endpoint of this connection is inside the enclave of the key-value store, preventing the untrusted world from learning sensitive information about the request. The payload data is separately encrypted and transferred. It never enters the enclave but is only placed in the untrusted memory of the server.

Starting with a `put()`-operation, the payload data, in our case the value, is encrypted on the client-side using a freshly generated one-time key. This cryptographic key is appended to the control data, and sent directly to the enclave with transport encryption. This approach has two advantages: It saves on transport encrypted payload that otherwise has to be transferred to the enclave and decrypted there. It eliminates re-encryption costs on the server-side as the payload data is already encrypted with a one-time key and does not enter the enclave.

In the case of a `get()`-operation, the actual request is sent as control data over the transport encrypted connection. The server-side provides an answer by sending the necessary control data over the encrypted connection, including the one-time key. The encrypted payload data stored in untrusted memory is transferred to the client as-is. Using both segments, the client can validate the freshness and integrity of the data.

In summary, the design of Precursor features the following two main benefits:

- The main share of the data that is managed by a key-value store is *only* encrypted (`put()`) and decrypted (`get()`) by the client side.
- Payload data never enters the sever side enclave, which prevents costly copy steps in and out of the TEE.

3.3 Critical Design Choices

Reducing SGX related overhead. Current SGX-secured key-value stores generate control information for preserving the integrity of the managed data on the server side and inside the TEE to enable the storing of as much data as possible outside the enclave. This follows the clear idea that the EPC is scarce and that the paging process is costly in terms of performance. Some systems even keep a part of the data inside the TEE for caching reasons, which might even increase the risk of paging. For Precursor, we decided to keep the payload data out of the TEE at all times. In particular, this avoids copying payload data in/out the enclave and the need to support any form of in-enclave caching for performance reasons.

Clients as Precursors and omitting cryptographic key management. In a *server encryption scheme* (§2.4), the server-side protects a unique key in its trusted context to encrypt the full payload. This also allows multi-tenancy and traditional access control schemes inside the server-side TEE.

To circumvent costly encryption on the server-side that can result in a bottleneck when combined with the high bandwidth of RDMA, Precursor encrypts the payload on the client-side; the server then only needs to ensure the integrity of the data from a security point of view. A possible design choice to address the proposed approach is to provide a single key for encrypting all clients' data. It seems efficient as the generation of one-time keys could be avoided and the keys could be excluded from `put()` and `get()` requests. However, we refrained from this shortcut. Indeed, it would put every client into the position to access all the data while

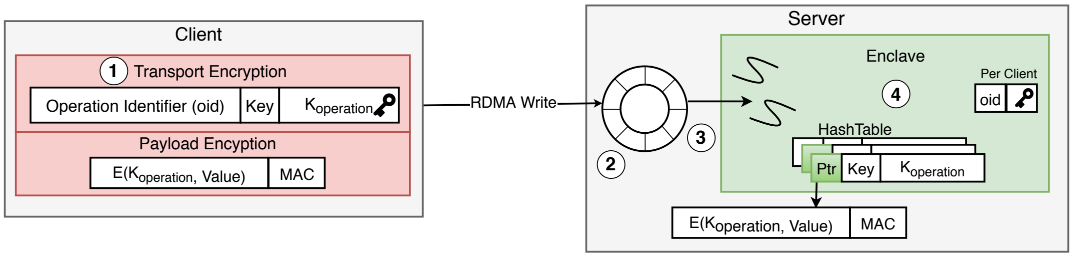


Figure 3: Precursor system architecture. The figure shows the idea of clients precursors and the server-side state

being authorized to use the service and even afterwards, unless all payload data is re-encrypted with a fresh key that needs to be shared with all clients still having access to the service. Accordingly, for Precursor, we decided to use an encryption key per key-value pair. It allows multi-tenancy and traditional access control schemes on top of Precursor as clients only learn about data and one-time keys submitted to the key-value store themselves or explicitly passed to them by the service. Finally, for the encrypted payload data transfer, one does not have to implement additional measures to prevent traffic analysis as a different encryption key is used for each key-value pair.

Integration of RDMA primitives with SGX. The low-level APIs of RDMA enforce a clear memory management strategy and establish an application-level flow-control policy. Authorized clients have direct write access to the server memory with the exception of the TEE. As an essential pre-requisite, clients must not overwrite each others' data or their own data unless it has already been processed by the server.

The server-side can be subdivided into two parts the trusted and the untrusted environment. Client request processing and data exchange are split over both parts, making buffer management and flow control challenging. To achieve high performance on the RDMA side and avoid the costly SGX context switches between trusted and untrusted execution, we tightly combine one-sided RDMA primitives and avoid synchronous operations for SGX. The core design choice is to use a separate ring buffer for incoming and outgoing requests per client. Inside the TEE, a worker thread updates these buffers.

3.4 Precursor Protocol Overview

Following the established design principles, we illustrate the general architecture in Figure 3. (1) It presents the state of a client that first completes a payload encryption procedure and then a transport encryption to protect against network intruders. (2) RDMA permits to directly write requests to a pre-allocated circular buffer in the untrusted memory, which then gets divided on the server side into two segments.
(3) The segment containing the security metadata goes into the enclave, while the payload remains in the untrusted region. (4) The enclave hosts a hash table that stores the correspondent security-metadata and pointers that link to the respective payload data.

3.5 Security and Encryption Notions

We introduce the cryptographic notions that we use to achieve integrity, confidentiality, and authenticity. To authenticate clients, we rely on message authentication codes (MACs) as they efficiently protect a message's integrity and authenticity.
We write `auth-encrypt(K, M)` to represent authenticated encryption of a message M with a cryptographic key K, respectively `auth-decrypt(K, M)` indicates authenticated decryption. We use two variants of cryptographic keys:

- $K_{session}$, a symmetric key shared between the client and the TEE on the server side to enable transport encryption.
- $K_{operation}$, a unique one-time key generated, and updated with each `put()`-operation by the client. It is used to encrypt the payload data and allows clients to decrypt and validate the replies.

3.6 Connection State and Preconditions

Once a client connects to a Precursor key-value store instance, it first performs remote attestation to verify a genuine, trusted execution capable CPU runs an enclave that hosts the expected binary on the remote side. Intel SGX provides the necessary remote attestation mechanisms to gain such a cryptographic proof that certifies the enclave's initial code and data and the expected hardware [5]. During this process, a shared secret key can be established that is further used for transport encryption between the client and the Precursor instance. Furthermore, to bootstrap the RDMA communication, the server shares a registered buffer memory window via its address and the associated memory identifier with the client.

3.7 Client Requests

After establishing a connection, the client now has the authorization to write its requests into the server memory and can compute the available space in its pre-allocated buffer on the server-side. For each operation, the client assigns a unique sequence number `oid` which is authenticated along with the message so that the Precursor server detects potential replay attacks (Algorithm 1, line 5).

Precursor provides standard key-value store functionalities extended with security-metadata as the client is carrying out the cryptographic computations for the payload data.

Add data. For a `put()` request, the client first encrypts the value using $K_{operation}$, and generates a MAC over its ciphertext (Algorithm 1, line 2-4), which corresponds to the payload data that is later transferred to the server. Following this, it prepares the control data that is authenticated and consists of `oid` and key item $K_{operation}$ (Algorithm 1, line 7-8). A newly generated initialization vector (IV) is necessary to prevent an eavesdropper from attempting a block replay and is excluded from the listing for brevity.

Algorithm 1: Secure `put()` client-side

```
State:
Operation identifier: oid;
Encryption key: $K_{operation}$;
Session key: $K_{session}$;
Encrypted value: *v;
ptr: pointer to the untrusted item;
Function Send_request(v, key):
    $K_{operation}$ ← KeyGen();   //generate key
    *v← E($K_{operation}$, v);
    mac ← MAC($K_{operation}$,*v ) ;
    oid=oid +1;
    control_data ← ($K_{operation}$, key, oid);
    buf ← $auth-encrypt(K_{session}, control\_data)$;
    request ← (buf, mac, *v) ;
    send request to the server ;
    return;
```

Algorithm 2: Secure `put()` server-side

```
upon receiving request do
    (buf, payload_data) ← request;
    control_data ← $auth-encrypt(K_{session}$,buf);
    ($K_{operation}$, key, oid) ← control_data;
    if $oid = C_i.oid$ then
        $C_i.oid$ ← oid +1;
        ptr ← store_to_untrusted (payload_data);
        putIntoHashtable(key, $K_{operation}$, ptr);
    else
        // Error handling
end event
```

When a message is received, the payload data remains in the untrusted memory, and only the control data is copied into the enclave. Thereby, the control data preserves the confidentiality and integrity of the payload data. The server decrypts the control data using the $K_{session}$ to verify the authenticity of the message and the client (Algorithm 2, line 2).

To detect replay attacks, it keeps an array indexed by a client identifier, where each entry holds the most recent `oid` operation and compares it with the new `oid` (Algorithm 2, line 4-5). If an attacker tries to send a message with the same number, the server detects it and discards the request.

Upon reception, a new entry is added, or an existing one is updated in the hash table (Algorithm 2, line 8). The entry consists of the key item and a value pair composed of the $K_{operation}$ and an associated pointer `ptr` to the payload data stored in the untrusted memory. If the key already exists, the server updates the entries in its memory: $K_{operation}$, the encrypted value and its MAC to support revocation.

Query data. When a client invokes `get()` to query a value, the Precursor server sends the untrusted payload data, and the trusted control data to the client. In doing so, Precursor shifts requests verification to the client. The Precursor server's task is thus reduced. It fetches the corresponding entry in the hash table using the request key item, encrypts it for transport via the client-specific $K_{session}$, binds it to the payload data then sends it to the client. To verify the integrity of the data, clients re-compute the received MAC over the encrypted value using the secret $K_{operation}$ and compare both the newly computed MAC and the received one. Using the unique $K_{operation}$ that originate from the enclave, clients can verify data integrity and detect state manipulation.

3.8 Client Management and Parallelism

To limit enclave transitions, Precursor runs a collection of threads equal to the number of CPU cores: trusted threads in the enclave and worker threads in the untrusted region.

A trusted thread has sequential tasks. It detects new client requests by polling a subset of circular buffers, then verifies transport confidentiality and integrity, and finally handles the request. Periodically, these threads update clients about the newly available buffer slots using one-sided writes.

When handling clients' requests, the proposed splitting approach demands slot allocation in the untrusted region for storing the payload data. Instead of performing frequent *ocalls* to allocate space from within the enclave, Precursor pre-allocates a memory pool and issues an *ocall* only when needed, i.e., to add extra space and reduce enclave transitions.

Eventually, the trusted threads write request replies into an untrusted queue. The worker threads send these messages using RDMA.

3.9 Security Discussion

We discuss the security properties of Precursor according to our threat model (§2.3).

Precursor guarantees. Precursor ensures the confidentiality and integrity of exchanged and stored data via established cryptographic operations. The use of one-time keys for the payload is robust and preserves forward secrecy. There is no need for pre-deploying keys or re-encryption once a client has been excluded from accessing the service.

During `put()` and `get()` operations, the client might have learned a number of one-time keys, but this knowledge cannot be exploited for uncovering additional information from the key-value store as each time a value is updated, a new one-time key is used. Nevertheless, if an excluded client would have unrestricted access to the network, she can change `get()` responses for her known keys. With access to the server's untrusted memory, she could in principle, modify values that were previously accessed by the excluded client. If such a scenario needs to be prevented, the MAC included in the payload data needs to be stored inside the enclave and has to be added to the transport encryption. This way, the former client cannot replace payload information as this would be detected via the securely handled MAC.

Precursor provides integrity with MAC verification and ensures rollback attack detection at the communication-level; thus, a malicious adversary cannot re-insert previous key-value items as each operation is appended with an increasing sequence number checked in the enclave. The clients' authenticity is securely checked in the enclave; the cloud provider cannot mimic or impersonate a client. However, as mentioned, the attacker can identify encrypted values in the network traffic; thus, she can examine and deduce patterns from similar transmitted requests and replies.

A different scenario is an attacker who targets the client; however, we assume that client-side operations and data are protected, for example, via an SGX enclave.

RDMA specific attacks. Precursor is exposed to the security consequences of RDMA protocols such as the denial of service and side-channel attacks. RDMA memory keys that allow an entity to write or read a remote machine's memory are predictable and not secured. While this pre-exchange of keys can be encrypted, if an adversary guesses this critical information, she can issue read or write requests, which negatively impacts Precursor performance; or overwrite the untrusted memory data. Also, an attacker might open many connections to Precursor, reaching the RNIC cache limit and thereby preventing further connections by other clients. Rothenberger et al. [48] propose mitigations to these attacks using existing hardware counters in RNICs, and other defense techniques which can be implemented on top of Precursor. Similar issues occur with rogue clients. Clients could deviate from the flow control and overwrite their request before being read by the server or write to an incorrect point in the buffer, producing garbage data [54]. Precursor can revoke access to corrupted clients using RDMA queue pair states transition [44]. Finally, Tsai et al. [60] trace major side-channel attacks in RNICs that allow a client to learn other client's accesses to a remote server. However, as mentioned in our threat assumptions, protecting against denial of service, side-channel attacks, and availability is beyond this paper's scope.

4 IMPLEMENTATION

We implemented Precursor in C, with about 5000 lines of code (LOC), where the enclave comprises 580 LOC. It uses the Intel SGX SDK 2.9 for Linux.

It exposes three ecalls that i.) initialize the hash table, ii.) start polling the circular buffer, and iii.) add a new client. For RDMA APIs, we use the core user-space `libibverbs` library. For the internal hash table in the enclave, we used the state of the art Robin hood hash table which resolves conflicts using open addressing [14]. This choice is driven by the fact that it provides a compromise between speed and memory usage. Unlike a chaining strategy, it does not require linked lists or extra pointers which slows down the hash table lookups. Besides, it is not influenced by TLB misses making it more suitable for in-enclave insertions [40].

Security functions. Client in-memory encryption is implemented using the crypto primitives from the *Libsodium* library[2]. It encrypts the value using the Salsa20 stream cipher, which generates a 256-bit secret key.
Then, we generate a CMAC hash over the encrypted value using `sgx_rijndael128_cmac_msg` from the Intel SGX SDK library. To verify clients' identity and transport encryption, packets are protected with an authenticated encryption, AES128 in GCM mode. In addition to the cryptographic metadata, a request includes an `opcode`, a `start_sign` and an `end_sign` operand to detect the start and end of a request.

The enclave hosts security metadata that consists of a 256-bit secret key, 1B `oid`, 4B client identifier that is stored in the hash table. This implies that the in-enclave data is kept small to prevent EPC paging. To allow concurrent accesses, the internal enclave hash table is read-write locked with a completely in-enclave mechanism. The server uses a single *ocall* function (called periodically to limit frequent transitions) to enlarge the pre-allocated untrusted list that stores the payload data.

RDMA optimizations. We use existing RDMA optimizations that already proved to improve throughput and latency [29]. Waiting for completion events with each send request adds extra overhead. *Selective signaling* is an optimization that reduces this overhead. It allows pushing a single completion after a batch of requests. The rest of the operations do not get an explicit notification.

The second feature that we use is *inline*. It allows copying small messages into the work request to send instead of the adapter retrieving it via a DMA read. It reduces the latency for small payloads (up to 912 B in our setup machines). We refer the reader to Kalia et al.'s design guidelines for a detailed explanation [29, 30].

5 EVALUATION

To evaluate the system performance of Precursor, we present an experimental comparison against the SGX-secured ShieldStore [31] in terms of throughput, latency, and EPC working set analysis. The evaluation sets out to answer the following three questions:

- How does Precursor compare to an existing SGX-guarded key-value store in terms of throughput and latency for various workloads and payload sizes? (§5.2 and §5.3)
- How does the offloading of cryptographic operations to the client improve the server-side performance?(§5.3)
- What is the impact of EPC paging on Precursor? (§5.4 and §5.3)

5.1 Experimental Settings

Testbed. In all our experiments, we use one server machine and several client machines as follows:

- Server machine: Intel Xeon E-2176G CPU (3.70 GHz, 6 cores, 12 hyper-threads) equipped with 32 GB of memory and a 40 Gbps Mellanox ConnectX-3 RoCE NIC.
- Client machines: five identical machines with an Intel Xeon E3-1230 CPU (3.40 GHz, 4 cores, 8 hyper-threads), 32 GB of memory, connected with a 10Gbps ConnectX-3 RoCE NIC. The sixth machine has an AMD EPYC 7281 16-Core Processor, 128 GB memory and a 40 Gbps RDMA NIC, which runs half of the number of clients.

For all experiments, we use the same software stack: Ubuntu 18.04 with Linux 4.15.0-62, Mellanox OFED 4.2 driver, SGX driver and

[2]https://download.libsodium.org/doc/

SDK version 2.9 [3]. We also apply the Intel microcode updates to mitigate the Foreshadow and Meltdown attacks.

Baseline Systems. We use ShieldStore as a baseline system as it is a recently published key-value store that was especially designed for throughput and addresses the aforementioned EPC limitation. ShieldStore clients and server interact through socket-based primitives. In a nutshell, it holds the encrypted key-value entries in the unprotected memory, each chained to a MAC entry. The entries create a Merkle tree to guarantee integrity-protection. The top of the tree corresponds to the hashes over a bucket list of MACs. In sum, ShieldStore provides an adequate state-of-the-art baseline as it corresponds to a server-side computation scheme. It is also currently also the only open-sourced SGX-tailored key-value store we are aware of.

Since ShieldStore does not feature the use of RDMA, we created a second baseline. We compare the proposed Precursor client-encryption with a Precursor server-encryption variant. Clients and the server rely on RDMA primitives. However, the full payload is transported encrypted and copied into the enclave, where its integrity and authenticity are checked. Next, we re-encrypt the payload and store it in the untrusted memory. In this case, the server handles the cryptographic verification and computation, i.e., we do not offload the encryption and integrity verification to the clients.

We use the YCSB benchmark [18] to evaluate both Precursor and ShieldStore. We concentrate on the *uniform* YCSB workload, where all items are equally likely to be accessed. We explore the effect of varying workloads (read-write ratio), value sizes, and client numbers.

5.2 Throughput Measurement

In all throughput experiments, we load 600k entries during the warmup phase and take an average of 8 repetitions which provides stable results.

We distribute 50 clients over the six machines and run the benchmark for 4 minutes to reach a constant throughput. When the performance is stabilized, we measure the server-side throughput. We run ShieldStore and both variants of Precursor with 12 server threads to reach the maximum throughput.

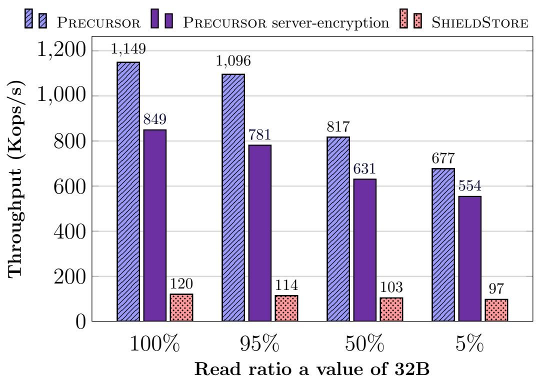


Figure 4: Throughput comparison varying workloads with 50 clients.

Effect of varying workloads. To have a complete view of both system performance, we use the following workloads:

- *Update-heavy* workload, 50% read (50% update), corresponds to YCSB workload A.
- *Update-mostly* workload, 5% read (95% update).
- *Read-mostly* workload, 95% read (5% update), corresponds to YCSB workload B.
- *Read-only* workload, 100% read, corresponds to YCSB workload C.

Figure 4 shows the throughput comparison between ShieldStore, Precursor, and Precursor server-encryption. The experiment uses a fixed value of 32B for a comparison with real-world workloads as in MemC3 [23]. With read-heavy workloads (*read-only* and *read-mostly*), Precursor performs 8.5× better than ShieldStore. With a decreased read ratio, Precursor performs 6.9× better than ShieldStore for mixed ratio workload, and 5.9× better for an *update-mostly* workload. The overhead of ShieldStore is due to TCP networking and the increased cryptographic verification. Besides, we can see that the traditional server encryption approach reduces the throughput compared to client encryption. Indeed, Precursor client-encryption increases the throughput by up to 40% than its server-encryption variant with read-heavy workloads. With update-heavy workloads, it achieves up to 29% more throughput.

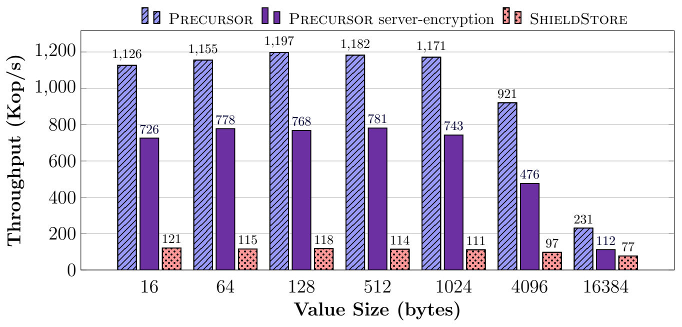
When the value is smaller than the size of the control data (≈ 56B), one could as an alternative store the value directly inside the trusted memory. This saves CPU cycles at the client-side and omits the need to read from the untrusted memory. We consider this as a future extension, where the key-value store switches to this optimization for small values. Nevertheless, Yang et al.'s analysis of in-memory caches at Twitter [67] shows that 50% of the values are bigger than 230B and 35% of the clusters are write-heavy workloads. Thus, we vary value sizes from 16 B to 16 KiB.

Effect of varying value sizes.

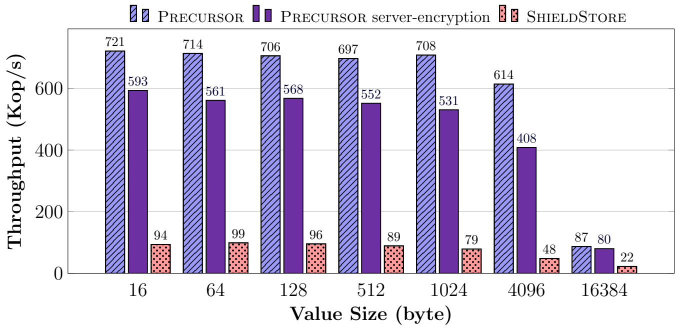
We focus on a *read-only* and *update-mostly* workload where we can see the difference between read and write operations. Figure 5 illustrates the results of this experiment. For an *update-mostly* workload, we observe that Precursor can perform up to 721k operations, while ShieldStore reaches a maximum of 99k.

Indeed, in a `put()` operation, the ShieldStore server copies the full request into the enclave. Next, it computes a MAC over the entry. As ShieldStore relies on a Merkle tree approach, it is necessary to update the root hash, which requires reading all MACs in a bucket and update the hash.

For a *read-only* workload, we observe that Precursor can perform up to one million operations, while ShieldStore reaches a maximum of 121k operations. Although ShieldStore is optimized and cleverly uses a Merkle tree design, the system needs to decrypt all entries in a bucket, search for the corresponding key, then verify its integrity. For this, it reads the bucket MAC lists, recomputes a hash over it, then compares it with the root tree. This overhead is unavoidable due to the design of ShieldStore and becomes even more apparent with bigger payload sizes. However, in the case of Precursor, the number of decrypted bytes remains constant as the payload is pre-encrypted on the client-side. The Precursor server only copies the control data, decrypts it once, and it remains in the



(a) Read-only workload



(b) Update-mostly workload

Figure 5: Average throughput comparison of Precursor against ShieldStore with various value sizes, for a *read-only* workload and an *update-mostly* workload with 50 clients.

enclave, while the encrypted value never enters the enclave. Hence, Precursor performs less copy steps while clients can still verify the integrity of the data.

The Precursor server encryption variant shows the expected overhead and results. Following the traditional approach for verifying integrity decreases throughput with an average of 49% for bigger request sizes (4Kib and 16Kib), and an average of 34% for smaller request sizes in case of a *read-only* workload. In case of an *update-mostly* workload, the throughput decreases by 27% on average.

Effect of increasing clients numbers. We simulated in this experiment a larger number of clients to evaluate Precursor's scalability. We used the six client machines, splitting the number of clients processes for both systems. This experiment uses 32B values. Figure 6 shows the results for a read-only workload. Precursor delivers its maximum throughput at 55 clients; with more clients, the throughput starts to decrease. The decline is due to the resource contention and cache misses in the RNIC [16]. Another possible limit is the necessary polling in the enclave. With more client processes, this might incur much CPU overhead.

As ShieldStore and Precursor use different networking technologies, we investigate in the following section, the latency of the full systems as well as separating networking from server-side processing.

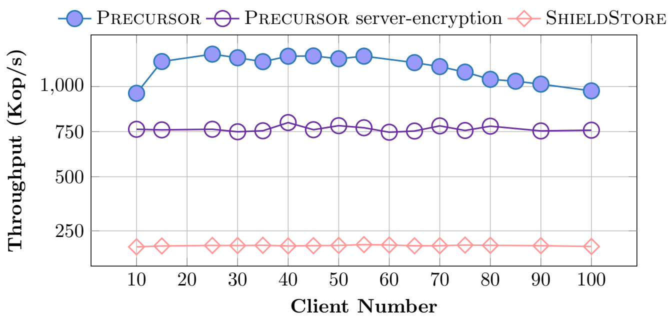


Figure 6: Comparison of ShieldStore and Precursor while varying the number of clients for a *read-only* workload.

5.3 Latency Analysis and Tail Latency

We now look at the tail latencies of a `get()` operation for a read-only workload (99th percentile), which is an important metric in datacenters. Figure 7 shows the Cumulative Distribution Function (CDF) of latency for requests with 32B, 512B and 1024B value for 1 million requests. Precursor has lower `get()` tail latencies. This can be explained by the usage of RDMA for communication, as well as offloading computations to clients as compared to state-of-the-art SGX-based key-value stores. We can see from the results that Precursor latency remains steady until 95% ($\approx 8\mu$s) , with a 99th percentile at 21μs. Besides, we see that with bigger values, Precursor tail-latency remains good and does not increase. ShieldStore seemed to have more outliers due to scheduling, kernel processing and TCP buffering. Indeed, using TCP and interrupts adds latency to network requests. We then loaded 3 million entries that would make Precursor trigger EPC paging because it exceeded the upper EPC limit. Till 90th percentile, Precursor latency is still 77% lower than ShieldStore, starting from 95% percentile the EPC impact becomes more apparent. ShieldStore is not affected by the EPC paging in this case because of its Merkle Tree-based design.

To further understand the performance differences between both designs and show the impact of different networking technologies and security protection techniques, we break down the `get()` latency in Figure 8. This experiment analyzes the average time spent performing `get()` requests, splitting networking and server-side. We vary the value size since it changes the amount of data that has to en-/decrypted for each request. The goal is to separate the time that is spent in the network processing and transmission and the time in the server while processing requests. It is especially to see the benefit of our design regardless of using RDMA. The results show that the server processing of ShieldStore is 1.34× slower than Precursor, due to the decryption/encryption of the full payload, the copy between the enclave and untrusted memory and the server-side integrity verification. When increasing the buffer size, the in-enclave latency of the Precursor server remains the same, while for ShieldStore it keeps increasing, 2.15× higher for large sizes.

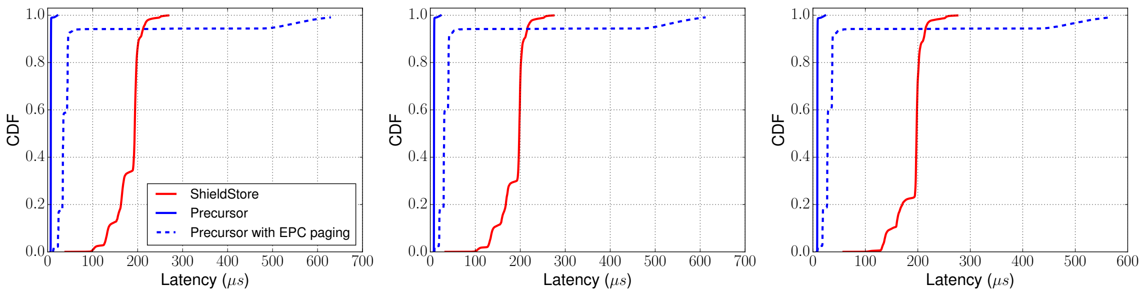


Figure 7: CDF of different latencies for 32B, 512B and 1024B for a *read-only* workload. The dashed line shows the impact of EPC paging on Precursor.

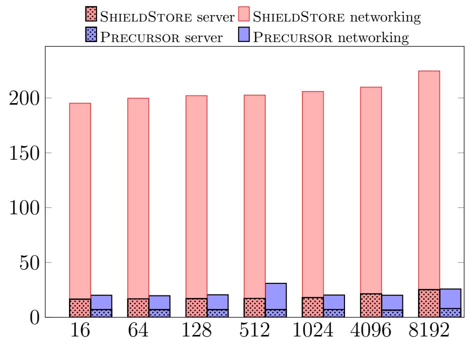


Figure 8: Average latency analysis of Precursor and ShieldStore under a *read-only* workload.

The different mechanisms we used in the paper all contribute to the good performance of Precursor. Using the right networking technology reduces the latency of the service by 26× and improves the server responsiveness time compared to a traditional technology. Avoiding data copy where possible, limiting the en/decryption improves the server latency compared to a traditional server-encryption scheme.

5.4 EPC State Analysis

Our design principle keeps the enclave's data sufficiently small to fit into the EPC by splitting the workload and maintaining a minimal data set in the enclave. This approach defers EPC paging and does not suffer from a tradeoff between computation and EPC overhead. ShieldStore's limitations results from the overhead of the Merkle tree verification steps for each request, which increases with a reduced number of MAC hashes cached in the enclave. ShieldStore design needs to reduce the number of MAC hashes in the enclave to sidestep the EPC paging. Thus, we show here the EPC analysis for Precursor and ShieldStore. To give an idea of how the EPC increases, we measure the enclave's working set size using the sgx-perf tool [64]. sgx-perf is a performance analysis tool for sgx-based applications. It traces which parts of the enclave are used and gets information about the working set including the number of pages and the size. We use a fixed value size of 32B and increase the number of `insert()` operations.

Table 1 shows the result of running the tool on both systems with a various number of inserts. At start time, ShieldStore allocates the full data structure with a statically fixed number of in-enclave hashes, and thus, its initial working set is 17,392 pages (≈67.9 MiB). Precursor only initializes a subset of the hash table in the enclave, which increases within a threshold and only needs 52 pages (≈ 0.2 MiB).

In general, these results suggest that Precursor keeps a small enclave memory footprint with an increasing number of keys. According to our results, even with paging, Precursor outperforms ShieldStore, which has the trade-off of decreasing the hashes in the enclave to limit EPC paging or increasing them to limit the number of mac verification steps.

	0 keys/init	*1 key*	*100,000 keys*
Precursor	52 pages (0.2 MiB)	65 pages (0.25 MiB)	2981 pages (11.6 MiB)
ShieldStore	17392 pages (67.9 MiB)	17586 pages (68.6 MiB)	17594 pages (68.7 MiB)

Table 1: EPC size with various key inserts

6 RELATED WORK

We are not aware of prior work which investigated the use of RDMA for an SGX-secured key-value store. Still there has been a set of related works that focused on dedicated aspects also addressed by Precursor. First, we discuss general performance relevant aspects of SGX that influenced the design of Precursor. Second, we analyze key-value stores secured via trusted execution, in particular SGX. Finally, related work considering key-value stores utilizing RDMA is outlined.

Mitigating limitations of SGX. As detailed in the introduction, so far two main performance issues have been identified when securing complex applications via SGX: memory usage beyond the limit of the EPC and excessive switching between trusted and untrusted execution mode. Overstepping the EPC boundary can be addressed by explicit outsourcing of data to the untrusted environment or more transparent approaches such as user-level paging.

The first case has been implemented for example in SecureKeeper [13] but is also performed by most of all data-intensive

SGX-guarded systems [9, 31, 49]. The second case has been proposed through smart pointers in Eleos [41]. STANlite [49] performs efficient user-level paging whenever a database workload requires more space than the in-memory state size. Precursor aims to avoid the transfer of all data into the TEE in the first place, thereby avoiding additional custom memory management. Avoiding performance degradation due to frequent transitions between the trusted and untrusted side has first been proposed by SCONE [8] and later explored in more detail by Weisse et al. [66] as well as integrated into the SGX SDK by Tian et al. [58]. Precursor embraces this approach and combines it with the use of RDMA to retain as much performance benefits of this communication technology as possible.

Secure data stores using Intel SGX. With the advent of SGX, the first impulse was to make its use transparent. Solutions such as Haven [10], SCONE [8] and Graphene [59] proposed to leverage the hardware protection of Intel SGX without the need to re-architecture applications, by placing systems support (e.g., a library OS) and the application binary in the enclave. All three solutions are susceptible to the issue of overstepping the EPC limit. While convenient, it turns out that tailored solutions offer better performance [15, 31]. Tailored approaches on protecting key-value stores in the cloud either utilize Merkle trees or tailor specialized data structures relying on a server-side encryption scheme. In both cases, the aim is to limit the active enclave state to the EPC size if possible. Key-value stores such as EnclaveCache [15] and ShieldStore [31] ensure integrity and confidentiality through server-side verification and rely on the large untrusted memory for storing encrypted data to sidestep EPC swapping overhead. ShieldStore uses a Merkle tree approach for integrity protection of data hosted in the untrusted memory. EnclaveCache uses multiple enclaves to isolate data among tenants. SPEICHER [9] is an LSM-based key-value store that stores the list of keys inside the enclave while maintaining the values in the unprotected memory. For these streams of research, CPU usage can be a limiting factor and they might not take full advantage of the available bandwidth at least when combined with novel communication technology such as RDMA. Sinha et.al proposes a trustworthy proxy that guarantees integrity using a Merkle tree structure. So far, Merkle tree approaches are considered to be computationally intensive and prone to concurrency bottlenecks [7]. Concerto [7] records a hash of the untrusted memory write and reads for integrity verification but does not address the performance-related limitations of SGX. EnclaveDB [47] and STANlite [49] are SGX-enabled databases that offer security with low overhead and a small TCB. None of the discussed systems focus on client-side encryption of the payload data or aims to avoid the transit of all data via the TEE such as proposed by Precursor.

Accelerated key-value stores using RDMA. RDMA has been explored to scale up various distributed key-value stores [22, 29, 36, 56]. Achieving high-performance implies an RDMA-tailored design, due to its programming model and low-level details [24, 30]. Proposed designs tend to rely on the one-sided primitive because it bypasses remote side CPUs. Still, these systems always assumed a fully trusted environment. Pilaf [36] and FARM [22] rely on one-sided reads from the server memory, which requires multiple round trips to fetch the actual result. Our implementation shares similarities with HERD [29], a key-value store that uses one-sided writes to deliver requests, with a defined server-side polling threads. However, we use a different in-memory data structure design as well as RDMA over a reliable connection.

The STANlite [49] database utilizes RDMA and as such combines to a certain extend trusted execution and RDMA. However, it is unlikely that STANlite takes fully advantage of the available bandwidth because it implements a server-side encryption scheme. With Precursor, besides avoiding the transfer of all data into the enclave, a tighter integration of the two technologies is achieved.

7 CONCLUSION

Intel SGX promises strong confidentiality and integrity guarantees for cloud-based environments. However, its current architectural limitations lead to a dilemma between security and performance. In particular, the service state is often encrypted inside the enclave to store it outside the trusted execution environment because the EPC memory is scarce, and overstepping it results in a costly memory paging process. In this paper, we designed Precursor, a secure key-value service that mitigates SGX limitations while maintaining security guarantees and achieves good performance. It outsources necessary data encryption to the client-side to lower the server load and thus benefits from the RDMA's high bandwidth. Our results show that the Precursor approach achieves high throughput and outperforms an existing state-of-the-art key-value store.

ACKNOWLEDGMENT

We thank our anonymous reviewers and our shepherd, Paulo Ferreira, for their helpful comments. This work was supported by the German Research Foundation (DFG) under grant no. KA 3171/9-1.

REFERENCES

[1] [n.d.]. 10th Gen Intel Core Processor Families Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf. Accessed: 2020-05-23.

[2] [n.d.]. ConnectX SmartNIC ICs. https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic. Accessed: 2020-05-23.

[3] [n.d.]. Intel Software Guard Extensions SDK for Linux. https://01.org/intel-softwareguard-extensions/downloads/intel-sgx-linux-2.3-release. Accessed: 2020-05-23.

[4] ARM Ltd. 2019. *Introducing Arm TrustZone.* ARM Ltd. https://developer.arm.com/ip-products/security-ip/trustzone

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. 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 '13).*

[6] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data.* ACM, 251–266.

[7] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17).* ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/3035918.3064030

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[9] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19).* 173–190.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[11] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 157–168.

[12] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. 2019. DR. SGX: automated and adjustable side-channel protection for SGX using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 788–800.

[13] Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Middleware'16: 17th International Middleware Conference Proceedings*. ACM. http://www.ibr.cs.tu-bs.de/users/brenner/papers/2016-middleware-brenner-securekeeper.pdf

[14] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE, 281–288.

[15] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. 2019. EnclaveCache: A Secure and Scalable Key-value Cache in Multi-tenant Clouds Using Intel SGX. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. ACM, New York, NY, USA, 14–27. https://doi.org/10.1145/3361525.3361533

[16] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[17] Te-Shun Chou. 2013. Security threats on cloud computing vulnerabilities. *International Journal of Computer Science & Information Technology* 5, 3 (2013), 79.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. on Cloud Computing (SoCC)*. ACM. https://doi.org/10.1145/1807128.1807152

[19] Intel Corp. 2020. *Confidential Computing Consortium*. https://www.intel.com/content/www/us/en/security/confidential-computing.html

[20] David Kaplan, Jeremy Powell, Tom Woller. 2016. AMD Memory Encryption, White Paper.

[21] Judicael B Djoko, Jack Lange, and Adam J Lee. 2019. Nexus: Practical and secure access control on untrusted storage platforms using client-side sgx. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 401–413.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[23] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 371–384.

[24] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 553–560.

[25] Advanced Micro Devices Inc. [n.d.]. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[26] Intel. 2014. Intel Software Guard Extensions Programming Reference, Revision 2. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[27] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. 2011. Memcached design on high performance rdma capable interconnects. In *2011 International Conference on Parallel Processing*. IEEE, 743–752.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 295–306.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance {RDMA} Systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 437–450.

[31] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 14.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside {SGX} Enclaves with Branch Shadowing. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 557–574.

[33] Patrick MacArthur and Robert D Russell. 2012. A performance study to guide RDMA programming decisions. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, 778–785.

[34] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[35] Mellanox. [n.d.]. *Perftest Package*. https://community.mellanox.com/s/article/perftest-package

[36] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[37] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided {RDMA} Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 103–114.

[38] NVIDIA. 2020. *NVIDIA MELLANOX CONNECTX-6 DX ETHERNET SMARTNIC*. https://www.mellanox.com/files/doc-2020/pb-connectx-6-dx-en-card.pdf

[39] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting {SGX} Enclaves from Practical Side-Channel Attacks. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 227–240.

[40] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 238–253.

[41] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 238–253. https://doi.org/10.1145/3064176.3064219

[42] Siani Pearson and Azzedine Benameur. 2010. Privacy, security and trust issues arising from cloud computing. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 693–702.

[43] Pinkerton and Deleganes. 2007. *Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security*. https://tools.ietf.org/html/rfc5042

[44] Marius Poke and Torsten Hoefler. 2015. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 107–118.

[45] Raluca A Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: Processing queries on an encrypted database. *Commun. ACM* 55, 9 (2012), 103–111.

[46] Christian Priebe et al. 2019. SGX-LKL. https://github.com/lsds/sgx-lkl.

[47] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[48] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. [n.d.]. ReDMArk: Bypassing RDMA Security Mechanisms. ([n. d.]).

[49] Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rüdiger Kapitza. 2018. STANlite–a database engine for secure data processing at rack-scale level. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 23–33. https://www.ibr.cs.tu-bs.de/users/sartakov/papers/sartakov18stanlite.pdf

[50] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*. 38–54. https://doi.org/10.1109/SP.2015.10

[51] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726* (2019).

[52] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *NDSS*.

[53] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.. In *NDSS*.

[54] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. 2020. Securing {RDMA} for High-Performance Datacenter Storage Systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[55] Rohit Sinha and Mihai Christodorescu. [n.d.]. VeritasDB: High Throughput Key-Value Store with Integrity using SGX. ([n. d.]).

[56] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.

[57] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*. 1–15.

[58] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution* (Toronto, Canada) *(SysTEX '18)*. ACM, New York, NY, USA, 22–27. https://doi.

org/10.1145/3268935.3268942

[59] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*.

[60] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. 2019. Pythia: remote oracles for the masses. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 693–710.

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

[62] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 233–251.

[63] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. Sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. ACM, New York, NY, USA, 201–213. https://doi.org/10.1145/3274808.3274824

[64] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference*. ACM, 201–213.

[65] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. Async-Shock: Exploiting Synchronisation Bugs in Intel SGX Enclaves *(ESORICS)*.

[66] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 81–93. https://doi.org/10.1145/3140659.3080208

[67] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 191–208.

[68] Xingliang Yuan, Yu Guo, Xinyu Wang, Cong Wang, Baochun Li, and Xiaohua Jia. 2017. Enckv: An encrypted key-value store with rich queries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 423–435.