STANlite – a database engine for secure data processing at rack-scale level

Vasily A. Sartakov*, Nico Weichbrodt*, Sebastian Krieter†‡, Thomas Leich‡, Rüdiger Kapitza*
* *IBR DS – TU Braunschweig*, Braunschweig, Germany
{sartakov,weichbr,rrkapitz}@ibr.cs.tu-bs.de
† *Otto-von-Guericke-University of Magdeburg*, Magdeburg, Germany
sebastian.krieter@ovgu.de
‡ *Harz University of Applied Sciences*, Wernigerode, Germany
tleich@hs-harz.de

***Abstract*—Intel's novel Software Guard eXtensions (SGX) enable secure and trusted execution of services, thereby paving the way to outsource sensitive data processing to external data centers. While SGX promises trusted execution close to native speed, frequent I/O operations and memory usage beyond a hardware-dependent threshold of currently 92 MiB result in substantial performance degradation. For memory-intensive workloads such as key-value stores and databases these penalties can be prohibitively high.**

We present STANlite – an in-memory database engine for SGX-enabled secure data processing in rack-scale environments. STANlite performs efficient user-level paging, whenever a database workload requires more space than the performance-friendly in-memory state size. Furthermore, STANlite smartly combines the properties of Remote Direct Memory Access (RDMA) and SGX to reduce the overhead of network-based I/O operations. While SGX usually provides confidentiality and integrity at the same time, STANlite enables a purely integrity preserving data management mode for additional performance. Finally, STANlite features a small trusted computing base and is memory-efficient, as it extends SQLite, a database for embedded use. We evaluated STANlite in terms of query response time. It outperforms a vanilla SGX-based SQLite version by 1.79× for microbenchmarks and 2.44× for TPC-C.

***Index Terms*—Memory encryption, database, SGX**

I. Introduction

Outsourcing all kinds of data processing applications to external infrastructures – in particular cloud computing environments – has become best practice. However, associated externalized data is often privacy or security sensitive, and thus, requires additional protection during storing and processing. As a result, customers either need to trust the provider including all its infrastructure and personnel, apply costly custom protection mechanisms ([1], [2]), or simply avoid outsourcing in the first place. Recent hardware platforms promise to substantially change this situation by enabling trusted execution in otherwise untrustworthy environments.

In particular, Intel recently introduced the Software Guard eXtensions (SGX) [3], which enable trusted execution based on encrypted and integrity-protected user-space memory regions named *enclaves*. Using SGX, the CPU performs encryption/decryption of enclave pages inside the memory controller, as a consequence enclave data never leaves the CPU as plain text. Because of the on-the-fly memory encryption/decryption the performance of enclave assigned computation is slower compared to commodity execution, but provides reasonable performance considering the provided security guaranties. However, there are two further sources for possible performance degradation: i) memory usage beyond a certain machine-dependent threshold and ii) mode transitions between trusted and commodity execution.

The hardware-dependent threshold is defined by the Enclave Page Cache (EPC) size, which is shared by all SGX enclaves, and for current platforms is limited to 92 MiB. If more memory is demanded this cannot be addressed by the EPC and an operating system supported paging mechanism is performed [4]. However, this process is expensive due to costly transitions between trusted execution in user space and privileged execution performed inside the kernel. On top, the exchange of pages between EPC and plain main memory requires the encryption and decryption of all evicted pages. While for workloads with a small memory footprint the performance degradation of SGX is typically low – for data-intensive applications such as in-memory databases the performance decrease due to frequent paging can be prohibitively high.

Frequent transitions between trusted and commodity execution are another reasons for severe performance loss. As enclaves are user-space entities, a frequent source are system calls and signals, i.e., due to I/O operations, which lead to mode transitions. So far this issue has been addressed via reducing the number of system calls by implementing them directly inside the enclave itself ([5]–[7]) and the support of asynchronous system calls, which effectively prevents an enclave to exit ([6], [8]). While the former is only possible for a limited number of system calls and is not an option for handling I/O, the latter requires an additional kernel module and synchronization between enclave and kernel threads [6].

Beyond this, it remains an open question how to further reduce the overhead of SGX-secured network-intensive applications such as in-memory databases. In rack-scale environments RDMA is on the rise and has successfully been utilized to speed up key-value stores and in-memory databases ([9]–[11]). However, so far it has not been explored, how this technology can be efficiently combined with trusted execution and especially SGX.

Accepted for publication in: IEEE International Conference on Cloud Engineering (IC2E 2018, http://conferences.computer.org/IC2E/2018)

In this work, we present STANlite – an in-memory database engine for SGX-enabled secure data processing in rack-scale environments. In summary, we make the following contributions:

- STANlite extends SQLite[1], which was originally designed for embedded use and as such features a small code base and has a small idle memory footprint. While a compact code base is essential to expose only a small attack surface, the small memory footprint is crucial to efficiently use the size-limited EPC for data processing.
- STANlite avoids costly operating system supported memory paging by featuring a scalable Virtual Memory Engine (VME) inside the enclave. This avoids costly mode transitions and gives STANlite full control over its memory. As such STANlite offers a purely integrity preserving data management mode for additional performance in cases where confidentiality is not a concern. In a simple microbenchmark, the VME of STANlite outperforms a vanilla port of SQLite to SGX, which performs hardware-based paging, by up to 4.44 times. For the SQLite standard *speedtest1* benchmarks STANlite is up to 1.79 times faster.
- STANlite features an RDMA-based, transition-free and zero-copy communication layer enabling fast remote database access in rack-scale environments. We evaluated STANlite with TPC-C and on average it outperforms a vanilla port of SQLite to SGX by 2.44 times.

The remainder of the paper is organized as follows: At the beginning, we give a brief introduction to SGX and analyze its restrictions. In Section III, we outline the overall design of STANlite, including its VME and communication layer. In Section IV, we detail relevant implementation aspects. In Section V, we present the results of our performed micro- and macrobenchmarks. Finally, we discuss related work in Section VI and conclude the paper in VII.

II. SGX Essentials

Intel SGX enables the creation of secure compartments called *enclaves* to achieve trusted execution [12]. An enclave is an isolated part of a process with its own code and data. All enclaves are located inside a physical memory region called the Enclave Page Cache (EPC). As the name implies, the EPC is organized in pages. It is transparently encrypted and integrity protected using Intel's Memory Encryption Engine (MME), which is part of the memory controller inside the CPU. EPC pages can only be accessed by the enclave owning the page, access from untrusted memory or other enclaves results in abort page semantics (i.e., *reads* return all binary ones and *writes* are silently discarded). Even privileged software (e.g., the operating system or a hypervisor) cannot access the EPC. Physical access to the memory will only read encrypted data, while writes are detected, because they trigger the integrated integrity protection.

The lifecycle of enclaves is managed by a set of new instructions [13] to create, enter, exit, and destruct enclaves. As these are low level instructions, a Software Development Kit (SDK) is offered by Intel, which abstracts the finer details and makes SGX easily usable. The SDK offers abstractions to call functions inside an enclave, dubbed *ECalls*, and to call untrusted functions outside an enclave, dubbed *OCalls*. As enclaves cannot use system calls, they can only communicate with untrusted code via shared memory. This means that other forms of I/O, e.g., network communication or accessing persistent storage, still involve untrusted code, which calls into or is called from the enclave.

The EPC containing all enclaves has a limited size of 128 MiB, due to hardware constraints. Of these, only 92 MiB are actually available, the remainder is used for integrity protection of the main part. To overcome this limitation, SGX is able to swap out EPC pages into main memory, similar to an operation system swapping out main-memory pages to persistent storage. However, this special page swapping is heavyweight, as it includes re-encryption, hashing, and storage of the hash during swap out. Similarly, during swap in, decryption, hashing, and checking the hash against the stored one is necessary. These steps are crucial to ensure confidentiality and integrity of swapped out pages, however, they cause a high performance overhead.

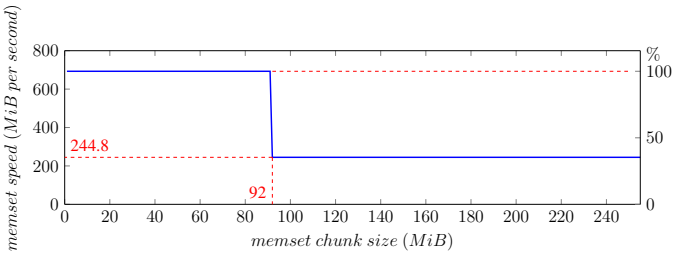

Fig. 1. In-enclave `memset` performance

Fig. 1 shows a microbenchmark of an in-enclave memory write. A simple `memset` is used to fill a buffer of varying size from 1 to 256 MiB of which we measured the bytes filled per second. As one can see, as soon as the EPC limit of 92 MiB was reached, performance degraded by a factor of 2.83 due to page swapping. Similar results can be found in SCONE [6]. Page swapping is done by the SGX kernel driver, whenever a page fault for EPC pages occurs.

We aim to circumvent mode changes from trusted execution to the privileged kernel mode by utilizing our own memory manager inside an enclave that transfers memory between EPC and main memory as outlined in the following section.

III. Design

Fig. 2 shows the general architecture of STANlite. The main part is the SQL engine, which is contained inside an Intel SGX enclave. Communication with this enclavised SQL engine is possible via a traditional TCP/IP socket, as well as using RDMA. The main contribution of STANlite is the custom Virtual Memory Engine (VME). The VME is plugged into the SQL engine and allows us to intercept memory allocation requests so that they can be managed by STANlite. The VME

[1]https://www.sqlite.org

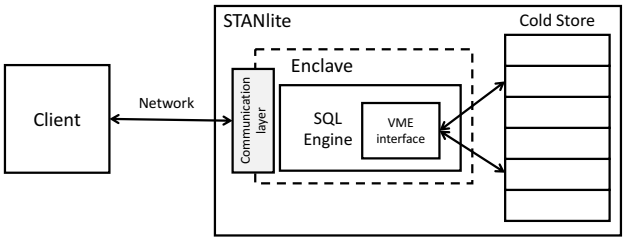

Fig. 2. Architecture of STANlite

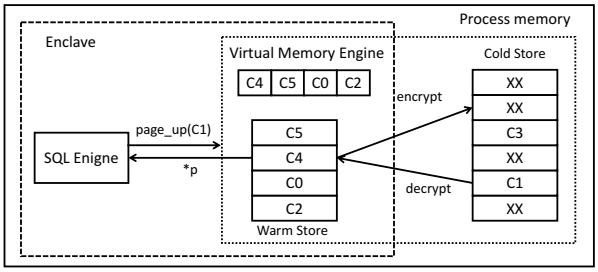

Fig. 3. Virtual Memory Engine

has multiple operation modes, which are described further in this section, but all modes share a so-called *cold store* outside of the enclave.

A. Threat Model

The threat model of STANlite is similar to the one of other SGX-enabled systems. We assume that an attacker has direct and privileged access to hardware and software components of a cloud infrastructure ([14]–[16]). We further assume that the Intel SGX-enabled CPU operates correctly and all instructions operate as they are described in their respective specification. STANlite does not prevent software flaws or exploitation of known and unknown vulnerabilities of software components. Side-channel attacks such as paging-based [17], cache-based attacks [18], [19] or synchronization-based [20] are also beyond the scope of this work.

B. Database Engine Memory Access Analysis

We chose the SQLite embedded database engine as our basis for STANlite. SQLite is typically used in embedded systems or in cases where a full database server is simply not needed. It has no external dependencies excluding the C standard library and does not require special system support. SQLite features a layered design, is customisable and self-contained and is able to work as a purely in-memory database. In STANlite, we used SQLite "as is", without modifications or enhancements. Conceptually, other DB engines could have been used in STANlite because the element of interest for STANlite is the memory access patterns created by SQL engines, not the specific one of SQLite.

In fact, in-memory databases use typically three main kinds of memory. Firstly, there is the static memory for the engine's code as well as data sections. This memory is allocated once and only freed when the database shuts down. Secondly, the database allocates memory to process incoming queries, which is short-lived as queries contain temporary data of requests. Lastly, long-lived memory is allocated for the data store itself. This memory can only be freed, when the content is not needed anymore, e.g. if the database, a table or a row is dropped/deleted.

The ratio of these regions varies with time. While the code/data region is big at the start, it becomes more and more negligible while data is stored inside the database. The data store itself with its long-lived memory allocations occupies the largest fraction of the memory. The share for temporary data memory changes dynamically with the load but is also small compared to the data store.

This classification of memory usage is not only important because of their differences in sizes but also for their differences in access patterns: Code and static data memory has a fixed size and is accessed all the time during execution. Contrary, request memory is allocated dynamically, can vary in size but is short-lived with its allocations not associated with other database components, e.g. the data store. Finally, the data store differs from the previous two. Conceptually, it mimics the content of the database files stored on disk, therefore it exhibits file read/write access patterns. Stored objects in this memory can reference each other via indexes so multiple allocations might be inter-linked.

In principle all discussed memory types can be virtualised. However, only the data store is actually relevant, as it occupies by far the largest amount of memory and is long-lived. Therefore, we focused on virtualising access to the data store memory within STANlite.

C. Virtual Memory Engine (VME)

The VME is providing memory virtualisation in STANlite. Since enclave memory is limited, its main component, the *cold store*, is located outside of the enclave. The cold store is used in all VME configurations. For some configurations a second store, called the *warm store*, is located inside the enclave to cache pages. In any case, the database engine need to be adapted to use the VME. Before access to memory is made, the database engine has to first ask the VME to load it. The VME then returns a pointer to the memory that must not be saved as it will change over time. Based on these initial design considerations, we envision four modes of operation for the VME, which are named based on the feature flags `C`, `F`, `I` and `i`. These flags can be combined, so for example, the mode `C-I` denotes *Caching with Integrity and Confidentiality Protection*. In particular we distinguish the following flags:

1) Integrity Only (`-i`): In this mode, the VME only uses the cold store to store memory outside of the enclave. No encryption/decryption of memory is made, only a hash sum of the memory is held inside the enclave to preserve integrity of the data and to detect rollback attacks. This is the most

performant mode but has the drawback of not guaranteeing confidentiality.

2) Integrity + Confidentiality (`--I`): Additionally to integrity protection, this mode encrypts all data before writing it to the cold store. Subsequently, the data is decrypted while loading it into the enclave. This guarantees confidentiality and integrity of the data in the cold store. This mode shares the same feature flag slot as the integrity-only mode as these mode are mutually exclusive.

3) Caching (`C--`): The previous modes only utilised the cold store outside of the enclave. However, some pages might be accessed more frequently than others so having those cached inside the enclave will increase performance because of less en-/decryption work. This mode adds a warm store to the enclave to cache a definable amount of pages inside the enclave. Pages are swapped in/out of the cache using a custom version of the Least Recently Used (LRU) displacement strategy.

4) Fetching (`-F-`): The fetching mode builds on top of the caching mode and allows the database engine to pin pages to the warm store to prevent them from being swapped out. This mode is named after the `Fetch` and `UnFetch` functions of SQLite to access memory mapped files. In this case, with Fetch, SQLite asks for a pointer to data and assumes this data is always in memory until it is unpinned via UnFetch. This moves the decision of what memory is needed or can be swapped out of the database engine. Access to memory this way is faster, as the database engine can directly work on the data whereas plain Read/Write semantics have the overhead of a memory copy.

D. Utilising RDMA for remote communication

As enclave software is not permitted to use system calls, communication between clients and STANlite need to be implemented by untrusted software. Therefore, requests to the database will be performed using ECalls to deliver the request into the enclave and to start request processing. However, reading from a network socket to untrusted memory and in turn copying the request into the enclave involves many parts of the system, which makes this slow.

We thus looked for alternative methods for network communication and turned to Remote Direct Memory Access (RDMA). This technology bypasses the kernel network stack by offloading the networking to the RDMA device. Data is delivered directly to a programs virtual memory space. In turn, STANlite can save processing time copying buffers and increase the throughput when using RDMA for communication with a client. Sadly, RDMA cannot directly write to enclave memory, however, we can still save a significant amount of time by just eliminating the additional interaction with the kernel.

Furthermore, we can utilise asynchronous calls between the enclave end the untrusted server side. Instead of doing ECalls, a thread stays inside the enclave and polls a flag in shared memory. The untrusted system can set this flag as soon as data is ready to be processed. This removes the slow enclave transition for requests, as the enclave is already executing and can just copy the request from the outside. Gathering the response is done the same way, with the untrusted side polling. This is related to existing approaches to speed up enclave request processing ([6], [8]).

IV. Implementation

Next, we provide details regarding the implementation of STANlite. First, we start with a description of our VME, then we detail how the VME is integrated with the utilized SQL engine, and finally we outline the devised RDMA-based communication layer including how RDMA and SGX can be efficiently combined.

A. Virtual memory engine

We developed the Virtual Memory Engine (VME) of STANlite in accordance with Read/Write access patterns used by databases. Fig. 3 shows a general scheme of the VME. As one can see, the VME consists of two parts located in different areas of a process virtual memory. The first part is located inside the enclave and contains the VME API library and, depending on the mode, a warm store. This part is active, i.e., it performs the actual virtualization of memory. The second part is a cold store that is located outside of the enclave. The cold store is passive and consists of a set of pages swapped out from the enclave.

Both warm and cold store are split into fixed size pages. The warm store is smaller than the cold store and consists of non-encrypted pages. The size of the warm store is defined at compilation time and can be arbitrary, but needs to be smaller than the size of the EPC minus the size of code/data sections of the database. In this case, all allocations are made by the devised VME allocator and costly EPC paging via the mechanisms of the SGX driver can be prevented. Buffer sizes and variables, like page size, warm store size and cold store size are pre-defined, but can be changed at runtime.

All allocations should be performed by the VME API library. This library enables allocations of pages inside the both stores and performs paging. In the case of the `--i/--I` modes, no warm store exists and the VME operates directly on the cold store. For the `C--` mode, the core function of the library is `page_up(c_id)`. This function receives an identifier of a page `c_id` as an argument and returns a pointer inside the warm store where this page is currently located. In other words, when the database engine requests a page, it deals with indexes of pages from the cold store. The warm store is used as cache for cold store pages.

There are three components of the warm store: an array of cached pages, a hash table covering the range of the cold store pages and a LRU queue, elements of which describe pages inside the warm store. The queue is a double linked list, the tail of which is the least used object, while the head is the most used. When the program requests a page, the VME looks for a free slot inside the array of cached pages, decrypts the requested page from the cold store into the warm store and updates the queue – a fresh page should be added to the head of the queue. Then the VME returns a pointer to the warm store slot. If a requested page is located already inside the

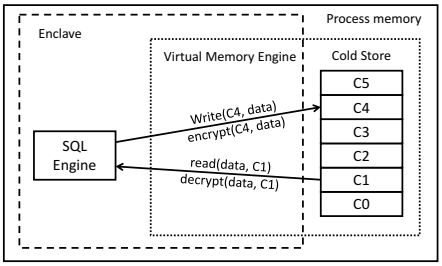

Fig. 4. Direct R/W access

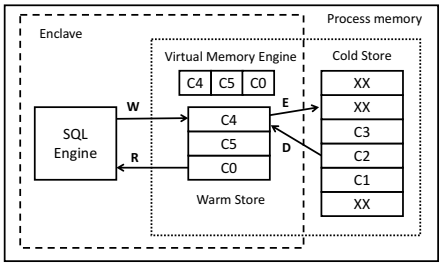

Fig. 5. Cached R/W access

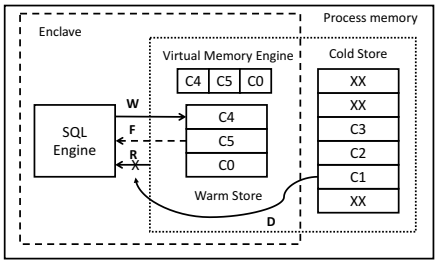

Fig. 6. Cached R/W access with fetch

warm store, then the VME just updates the queue and returns the pointer to the page.

If the program requests a page, and there is no free space inside the warm store, then a virtual page fault takes place (see Fig. 3). The VME takes a page from the tail of the queue (C4), and puts it into the corresponding address inside the cold store. Since the cold store is located outside the enclave, i.e., in an untrusted area, the VME encrypts this page. Then the VME takes a requested page (C1) and decrypts it into a previously freed slot inside the warm store. Furthermore, the integrity of the loaded page is validated. After that, the VME returns a pointer to the the warm store slot.

The VME does not control swapped out pages. These pages can be corrupted, and what is also important, fresh pages can be replaced by old ones. This attack is named *replay attack*, and the VME should prevent it. Before a swapping out a page, the VME computes the SHA-224 hash sum of the page's plain data. When the VME needs to swap in a page, it compares the hash of the loaded data with the previously stored hash value. If the values are different, then the VME stops working and detects an integrity violation.

However, memory encryption/decryption for confidentiality protection requires computation resources. While we are using hardware-accelerated primitives provided by the Intel SGX SDK, these primitives are heavyweight. In some cases, developers of a database system can decide to use an integrity protection only, and evict pages non-encrypted. We support this operation as mode `--i`, but it can be applied only on a database level. In sum, each database can use only one VME, and to enable a combination of different VMEs a developer needs to split a database into multiple databases each featuring the own VMEs.

B. SQLite as a basis for STANlite

STANlite uses the SQLite embedded database as a SQL engine and extended it with two additional layers. The new top layer – the *communication* layer, interacts with a network subsystem and passes messages to/from the SQLite *Core*. The Core processes the requests and consists of a SQL compiler and a virtual machine [21]. The Core utilizes the *OS interface* layer to store database data. For that, the core uses a *B-tree* to maintain on-disk data, and a *Pager*, to cache stored pages. The in-memory form of SQLite uses a page cache, but in contrast to the file-based form, SQLite never drops cached pages.

Before an integration of our VME with SQLite, we analyzed an API of the two lowest layers – the Pager and the OS interface. Conceptually, the OS interface implements access to storage, while the Pager is a store for cached pages. We have chosen the lowest layer, the OS interface, to integrate our VME since it provides a more flexible set of APIs, and provides more freedom for page management.

As described early, the database interacts with a storage layer on a segment granularity. This access assumes read and write patterns of communications. The database prepares a buffer to write and requests the operating system layer to store the content of the buffer at a certain place on a disk. When the database needs to read something from the disk, it prepares a read buffer and asks the operating system layer to fill the buffer with the data from the disk. This data can be cached inside the database, but conceptually, the database only works with data belonging to the current request.

This independent block access pattern enables development of a virtual memory engine supporting an infinite size cold store, because the VME does not use all segments at the same time. The VME can hold frequently used pages inside the warm store, and when the warm store is full evict pages as necessary. This is the general approach to integrate our VME with the SQLite database. However, there are several options, and in the following we consider different cache designs and interaction strategies of the VME and the upper layers.

***Read and write**:* The operating system layer provides a simple communication interface between the database and the storage. A *write* call is used to store data on disk, and a *read* call is used to load data from the storage. The VME thereby can apply different schemes.

*a) None caching mode - **`--I`**:* Fig. 4 shows an interaction of the `--I` operation mode of the VME and the database engine. As one can see, there is no cache at all and *read*/*write* operations are transformed directly into *decryption* and *encryption* operations. With the direct cold store operations and without temporary cache, the database consumes only a little amount of heap memory. Accordingly this mode of operation can be used in strongly memory restrict environments, or utilized for certain kind of workloads when caching of data does not increase performance (i.e., random access of a large database

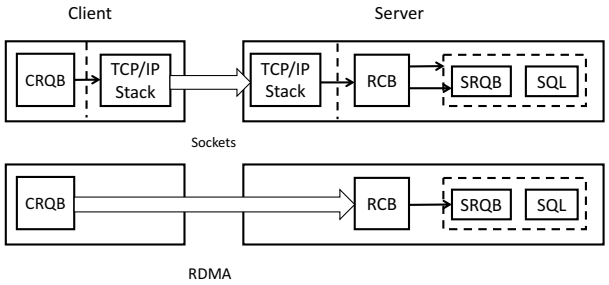

Fig. 7. Difference between TCP/IP and RDMA

that by far exceeds the EPC size).

*b) Caching mode - **C−I**:* Fig. 5 shows an interaction of the C−I operation mode of the VME and SQLite. As one can see, this mode utilizes both warm and cold stores. In this mode, the VME follows the model described in Section IV-A: the cold store consists of an encrypted form of swapped out pages, while the warm store consists of a plain text form of cached data. The LRU cache manages the migration of pages and provides integrity checks for swapped-in pages.

*c) Caching and fetching mode - **CFI**:* Modern versions of SQLite support not only Read and Write access patterns but also *Fetch* and *UnFetch*. These calls were implemented to enable memory-mapped access to a database file. For example, the *Fetch* call requests a pointer to memory where a particular Database (DB) segment is stored. To prevent corruption of data, SQLite assumes that all fetched pages are "pinned", i.e., cannot be swapped out, until the *UnFetch* call is invoked. If the requested page cannot be fetched, SQLite uses Read/Write operations on the same DB segment. Thus, this mode limits freedom of the VME to swap in and swap out pages.

Fetch and *UnFetch* calls were integrated into the VME. Fig. 6 shows a scheme of the CFI operation mode. As one can see, as in the C−I mode, the VME uses cold and warm stores, as well as integrity protection and a LRU queue. But now in addtion, each object in the queue has a *pinned* flag. The VME sets this flag each time SQLite issues a *Fetch* call, and clears it after a *UnFetch* call. With the set *pinned* flag, the page cannot be swapped out, but after a *UnFetch* calling, the VME places it to the tail of the LRU queue. Thus, the page will be swapped out as soon as the next page will require free space inside the warm store.

In this mode, it is possible, that all pages inside the warm store are pinned. In our access pattern, when the warm store is filled, we still can perform Read/Write operations directly on the cold store. Fig. 6 shows this example. All pages [C4, C5, C0] are "pinned" in the warm store. The {Write, C4} operation can be performed because the page [C4] is in the cache already. Also, the {Fetch, C5} operation will be successful since the page [C5] is in the cache. But the VME cannot swap in the page [C1] for the {Read, C1} request, because all pages are pinned. In this case the VME decrypts data from the [C1] segment of the cold store into a request Read buffer, performs an integrity check, and successfully completes the request.

In short, when the cache is overloaded, the CFI mode operates in the same way as −−I mode, until SQLite will release some warm pages via the *UnFetch* call.

C. Client-Server Communication

To enable remote access to STANlite we implemented a small network library based on TCP/IP and RDMA. This library supports exchange of SQL requests and responses with a remote server. Conceptually, the library consists of two components. On the client side, there is a function that marshals requests and sends them to an untrusted server service. This service transfers data into an enclave, reads the response and sends it back. This scheme is similar for both network technologies – RDMA and TCP/IP. However, there are specific implementation differences, which are shown in Fig. 7 and described in the following.

***TCP/IP Sockets**:* The following steps show how a client request is transferred to a STANlite instance: First, a client request is stored inside the request buffer (CRQB) on the client side, which is then copied into a TCP/IP socket. This places the data into the kernel TCP/IP stack. It is then send to the remote site via the network device. On the remote host, the network device delivers the data into the kernel TCP/IP stack of the server. The server software reads data from the TCP/IP stack and copies it into a receive buffer RCB. After that, the server delivers this request to the SQL engine, located inside an enclave. For that, the network service issues an ECall into the enclave and after that, the enclaved part of the network API decrypts the incoming package into the enclave request buffer SRQB.

***RDMA**:* On the contrary, RDMA communication does not require additional copies and can sidestep the kernel TCP/IP stack. Instead, data from the registered buffer CRQB on the client side can be delivered into the registered buffer RCB on the server side. We send this data with a notification (RDMA_WRITE_WITH_IMM) and store inside the notification the actual size of a transferred data. After the delivery of the data, the untrusted server side notifies the enclave thread, which in turn decrypts the request into the enclave buffer SRQB and starts processing. In sum, the RDMA version uses neither additional memory copies, nor ECalls but consumes additional CPU cycles due to polling.

***Integrity**:* Communication between clients and STANlite should be protected against intrusions and eavesdropping. To prevent this, we implemented traffic encryption on top of the communication layer. To enable a unified layer for TCP/IP, as well as RDMA-based communications, we implemented the encryption of traffic as an independent library, and added it to the server and client API libraries. The library operates on top of TCP/IP and RDMA packets and performs encryption and decryption of packet payload. The payload is either a SQL request or a database response. The server side performs encryption/decryption of requests inside the enclave, while the client side performs encryption/decryption directly inside the plain process.

Each request consists of four components: the payload of the request, the size of the payload, the sequence number of the

request and a checksum of the request. The sequence number is necessary to prevent a replay attack. Both the server and clients have counters for the request sequence numbers. Clients and the server increment their corresponding sequence number each time a request is sent (for the client) or received (for the server). If the server or the client receives a request with a previously used sequence number it drops it and detects a possible replay attack. The per-request checksum is an integrity protection technique, which prevents modification of requests. We utilized SHA-224 as our hashing algorithm. And finally, all requests are encrypted by the AES-CBC encryption algorithm. Keys can be generated easily by a DiffieHellman key exchange procedure [22]. As an alternative TLS termination inside the enclave, as offered by TaLoS[2] could be used.

D. Persistency

The structure of the cold store mimics the layout of a database file. The cold store can be allocated via the *posix_memalign()* call or can be a file mapped into a virtual memory via the *mmap* system call. In the second case the DB operates as a persistent database, and for this operation mode we added functions for warm store purging and syncing. However, supporting persistent storage additionally requires protection against rollback attacks as for example proposed by LCM [23] and ROTE [24], which is out of scope of this work. As consequence, all measurements and experiments in this work were made on top of an in-memory form of the DB.

E. Dynamic reconfiguration

Most of basic SQL request like `SELECT`, `INSERT` or `DELETE`, do not require a lot of heap memory for data processing. However, with complex requests like `CREATE INDEX`, SQLite core consumes a significant amount of memory, comparable with the current database size. If the database size fits into the remaining EPC memory, then these requests can still be performed inside the EPC without involvement of hardware-based paging. But to do this, memory consumed by the warm store should be reassigned to gain additional heap memory. For these cases we added a `PRAGMA` command that purges the warm store, limits the size of the warm store to a requested value and switches the current VME mode to `--I`.

V. Evaluation

We evaluate STANlite in terms of query response time using three different benchmarks, a custom *microbenchmark*, *Speedtest1*, and *TPC-C*. Our *microbenchmark* is a simple `SELECT` statement, characterized by intensive read memory accesses. The second benchmark is a full-fledged macrobenchmark based on Speedtest1[3], which is a benchmark suite to measure SQLite performance. As a third benchmark, we use the complex TPC-C test suite [25], which implements an abstract billing system of an industry service with multiple users.

We are interested in measuring STANlite's performance with different VMEs configurations and communication layers, as well as the impact of different database sizes. We compare STANlite against two versions of SQLite, an in-memory version, ported to SGX and a native, non-modified version. In summary, we compare the following six implementations (cf. Section IV-B):

- STANlite in `CFI` mode (`CFI`)
- STANlite in `C-I` mode (`C-I`)
- STANlite in `--I` mode (`--I`)
- SGX SQLite (vanilla, or `VNL`)
- Non-modified SQLite (native, or `NTV`)

Using our most complex benchmark, TPC-C, we measure the response time of each implementation using either TCP/IP or RDMA as communication layer. In addition, we investigate the potential performance benefits of STANlite's `C-i` mode, which does not encrypt evicted pages, against its encrypted counterpart `C-I`.

A. Setup

As the performance of STANlite is influenced by numerous factors other than VME type, DB size, and communication layer, we keep all parameters that we are not investigating consistent between experiments.

Evaluation System: We use identical hardware platforms for all server and client systems: Intel Xeon CPU E3-1230v5 (3.40 GHz, 4 cores, 8 hyper-threads), equipped with 32 GiB of RAM and Mellanox MT27520 RoCE RDMA controller (10 Gibit/s). We use identical network cards for RDMA and socket-based communications. Regarding software, we run Ubuntu 16.04.3 with the kernel version 4.4.0 as an operating system and use RDMA libraries from Mellanox OpenFabrics Enterprise Distribution version 4.1-1.0.2.0, Intel SGX SDK version 1.8, and SQLite version 3.18.2. For encryption/decryption, we use the primitive functions from Intel's IPP library provided in the corresponding SGX SDK. We built all software components using the "-O2" flag.

As for all enclaved applications, STANlite's Trusted Computing Base (TCB) includes only software located inside an enclave. In case of STANlite, its TCB includes SQLite as an SQL engine, our implementation of VME and RDMA, and the Intel SGX SDK libraries for E- and OCalls, encryption and decryption, and basic memory allocation. Additionally, for some of our experiments, we had to include a load generator into the enclave. The memory usage of code and data sections of STANlite inside the enclave consists of approximately 1 MiB.

Memory Layout: Two important parameters are the page size of SQLite and the segment size of the VME. Using different values for both parameters can have a high impact on performance for different queries. In our evaluation, we use a page and segment size of 4,096 bytes for all experiments, which is the default value of SQLite.

Another parameter is the cache size of the VME. All STANlite modes that employ a caching VME (i.e. `CFI`, `C-I`, and `C-i`) use a warm store of 80 MiB for microbenchmark and TPC-C and a warm store of 70 MiB for Speedtest1. As `--I` does not use caching the size is 0 for all benchmarks. Both

[2]https://github.com/lsds/TaLoS

[3]http://www.sqlite.org/src/finfo?name=test/speedtest1.c

SQLite versions, VNL and NTV use their own implementation of in-memory stores.

For all STANlite modes (i.e. CFI, C-I, C-i, and --I) the heap size reserved for dynamic memory allocations is limited to 16 MiB for TPC-C and microbenchmark and to 300 MiB for Speedtest1. The heap size of VNL and NTV is limited to 2 GiB for all benchmarks.

B. Benchmarks

In the following, we describe each benchmark in more detail.

Microbenchmark: In our microbenchmark, we perform random select queries on different database sizes. One experiment in the microbenchmark consists of 10 random select queries on a given database for each implementation. The used database consists of a single table with a primary key and a symbol field[4]. We fill the table using multiple insert queries[5] that each add a record with 1 KiB of random data to the table. The table we use in the first experiment consists of 10,000 records (i.e. 10 MiB). For each following experiment we increment this value by 10,000 up to a size of 500,000 records (i.e. 500 MiB), which results in a total number of 50 experiments. In each experiment we execute 10 random select queries[6] and measure the total response time of these queries.

Speedtest1: Speedtest1 is a benchmark provided by SQLite, which features a wide range of sequentially executed requests. During execution, this test suite creates and fills several tables with random test data, and performs multiple operations, ranging from a simple SELECT to complex subqueries and four-way JOINs. We use Speedtest1 with the default payload of SQL records and a size value of 2,000. We perform a local execution of the Speedtest1. In this mode, the load generator is located in the same environment as the DB.

TPC-C: TPC-C is a well-known online transaction processing benchmark. It simulates DB transactions of a wholesale supplier that operates a number of warehouses. The benchmark is designed to scale with the number of warehouses. Thus, we generated a set of queries for different numbers of *warehouses*, each of which increases the DB size by 110 MiB on average. In sum, we generated 19 experiments, incrementally increasing the number of warehouses from 1 to 19.

As different DBs have a different query syntax and architectural features, there exist many different implementations of the TPC-C benchmark. Instead of implementing our own TPC-C, we emulate its workload by executing a set of queries generated by the open-source PY-TPCC engine[7].

C. Results

In the following, we present our findings for all benchmarks.

Microbenchmark: In Fig. 8, we show the throughput for different databases sizes. For all implementations that use caching (i.e., VNL, C-I, and CFI), we can see two distinct behaviors

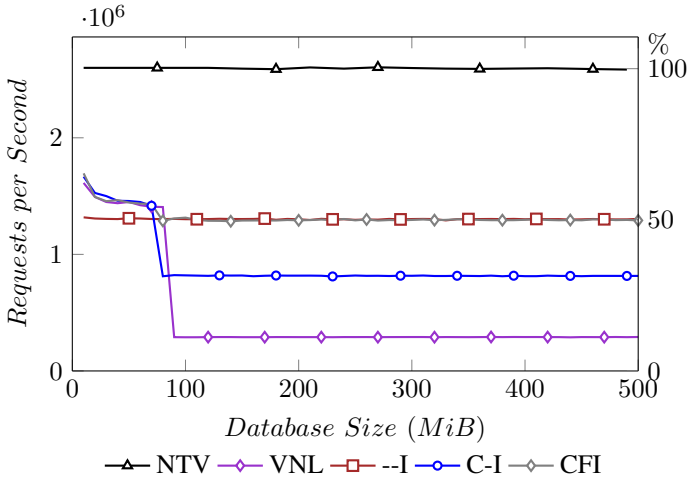

Fig. 8. Comparison of performance in Microbenchmark

depending on the DB size. When the DB size is smaller than the EPC or warm store size limit, all caching implementations demonstrate nearly the same performance. For larger DB sizes VNL is outperformed by all other implementations, followed by C-I and then CFI.

Another noteworthy result of this benchmark is the behavior of the non-caching --I mode. Its performance stays constant across all DB sizes. When the DB size is smaller than the EPC size, --I shows the worst performance of all evaluated implementations. However, for larger DB sizes, --I shows the same performance as CFI and outperforms VNL and C-I. Contrary to other implementations, --I does not use a cache and performs encryption/decryption on each read/write request. However, for a DB larger than the EPC size limit, caching becomes useless for random SELECT requests, as cached data is unlikely to be used again. In this case, C-I performs a combination of two operations: an encryption of warm data into the cold store with following decryption of cold data into the warm store, and copying of the decrypted data into the SQL engine. In contrast, --I only performs encryption/decryption without copying data. Thus, the performance of --I does not degrade. CFI has the same behavior as --I for larger DBs, since the VME cannot swap out fetched pages and, thus, effectively works as being in --I mode.

Comparing the encrypted implementations with NTV shows that encryption has a substantial impact on performance. Before reaching the EPC size all implementations show a performance of just about 50% of NTV. Using larger DBs, the performance of VNL degrades to approximately 11.2%.

Speedtest1: In Fig. 9, we use bars to represent the execution time of every query of Speedtest1 for each implementation, in the order of execution. Additionally, we use lines to show the growing size of the DB during the benchmark's execution for VNL and STANlite with VME.

Regarding the memory usage of the DB, we can see that STANlite consumes less memory than VNL. The difference

[4] `CREATE TABLE stest(ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, BODY CHAR;`

[5] `INSERT INTO stest (BODY) VALUES('<..>'))`

[6] `SELECT * FROM stest ORDER BY RANDOM() LIMIT 1`

[7] https://github.com/apavlo/py-tpcc

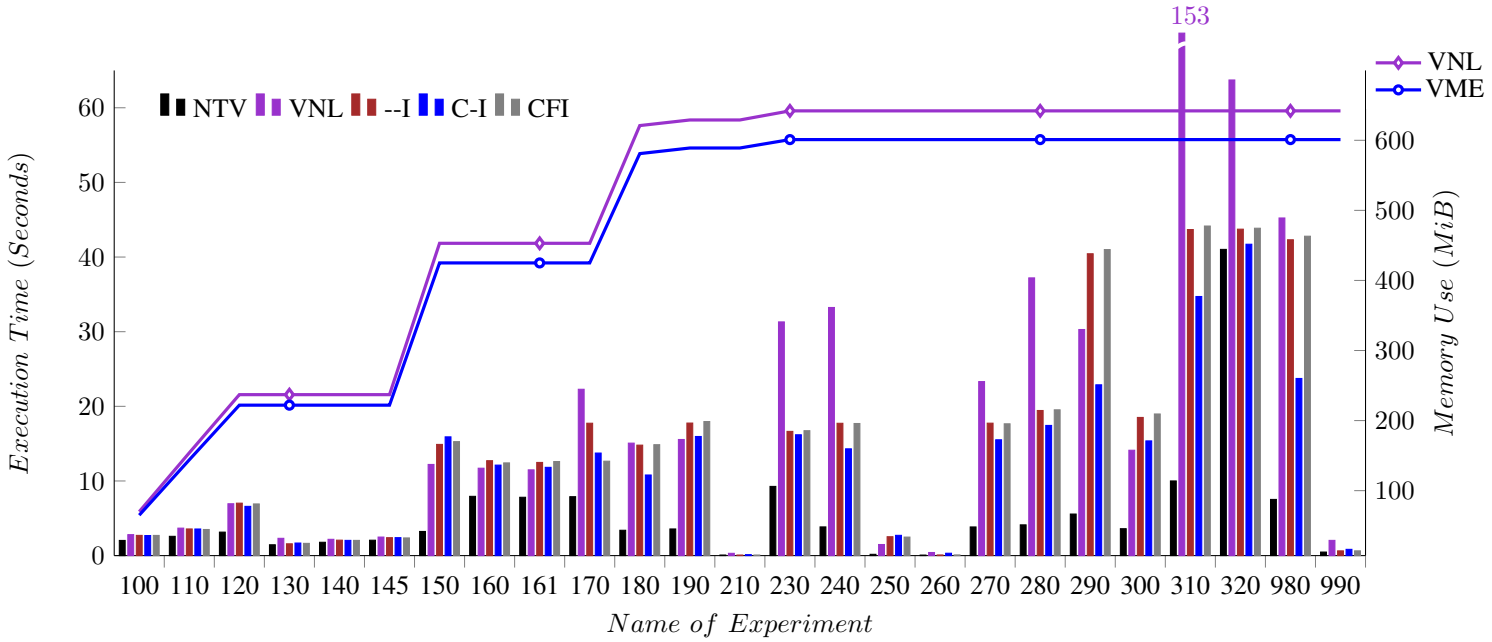

Fig. 9. Comparison of performance in Speedtest1 (local execution)

has a constant factor of approximately 6.8%.

Surprisingly, there are multiple experiments, in which VNL outperforms CFI and --I (e.g., 150 (creation of indexes for tables) and 300 (refilling with different conditions)). Analyzing these queries, we identified that SQLite requires a lot of heap memory to process this kind of queries. As mentioned above, for this benchmark, we limited the heap size to 300 MiB, however, these requests consume hundreds of megabytes. Since we do not virtualize heap memory, we see a corresponding performance gap resulting from hardware-based paging.

There is a set of experiments, where all engines show close performance (e.g., 100-145, 160, 161). However, for several experiments VNL shows dramatical performance degradation. Especially experiment 310 (four-ways JOINs) and 320 (subquery in a result set). These experiments can be characterized by an intensive memory use caused by active read/write accesses. In contrast, experiments, in which VNL shows its best results can be characterized by operations on indexes. In most experiments, C-I performs better than CFI and --I, while CFI and --I often show similar performance.

TABLE I
TOTAL TIME REQUIRED BY SPEEDTEST1 (IN SECONDS)

DB Size	NTV	VNL	--I	C-I	CFI
51 MiB	6.9	8.6	24.1	13.3	13.4
601 MiB	136.5	545.0	373.3	305.2	370.6

Tab. I shows the total time of Speedtest1 for small (i.e., 51 MiB) and large (i.e., 601 MiB) DBs. For small DB sizes VNL outperforms all STANlite modes. As expected, --I is substantially slower than C-I and CFI for a small DB, while C-I and CFI show similar performance. As we did focus on optimizing STANlite for small memory usage, we expected a performance degradation due to the additional layer of indirection. Regarding larger DBs, we can conclude that for Speedtest1 C-I shows the best performance, excluding NTV. The total time for C-I is about 2.2 times slower than NTV and 1.79 times faster than VNL.

As mentioned in Section III-C, VME is able to evict non-encrypted pages. Thus, we compare the performance of VME with and without confidentiality protection. In Speedtest1, C-i shows a better performance than C-I, as C-I on average is 1.2 times slower than C-i.

TPC-C: In Fig. 10, we show the results of the TPC-C benchmark. For all implementations, we use RDMA as well as TCP/IP as communication layer. We compare Transactions per Second (TpS), as it is the primary metric of TPC-C.

The general results of TPC-C are similar to the previous benchmarks. As expected, VNL degrades substantially when increasing the DB size beyond the EPC limit, while otherwise it outperforms all STANlite modes. While the constant performance of --I is lower than CFI and C-I for small-sized DBs, it outperforms both for DBs with sizes larger than the EPC size limit.

Furthermore, our results indicate that the selection of a particular communication layer has a great impact on the performance of STANlite. For all STANlite modes, there is a constant performance improvement of approximately 45% when using RDMA-based networking. NTV's performance is improved by approximately 36%. The performance improvement of VNL

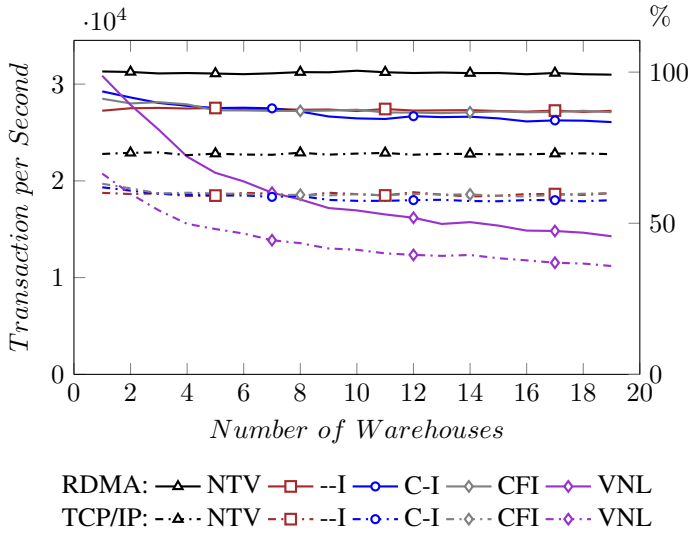

Fig. 10. Comparison of performance in TPC-C

decreases with larger DB sizes from 49% (1 warehouse) to 27% (19 warehouses). Compared to the VNL with TCP/IP communication layer, STANlite is faster by 2.44 times for a database 2 GiB in size.

D. Threats to Validity

In the following, we want to address some issues that might affect the validity and generalizability of our results. We implemented the microbenchmark by ourselves. This may bias our results in favor of STANlite. In addition, we used randomness, which may also bias our results. We tried to mitigate a potential bias by repeating our experiments multiple times and also evaluating STANlite with the help other benchmarks, such as Speedtest1 and TPC-C that are provided by others.

VI. Related works

Orenbach et al. introduced Eleos, which features enclave-level software-based paging for C++ based programs [26]. Key abstraction of their design are *spointers*, a specific instance of smart pointers that can determine if referenced data is inside or outside the EPC. As a consequence data can be paged into the enclave without mode transitions. In principle STANlite shares the general direction with Eleos but focuses on custom paging support for an in-memory database and enables fast remote interaction using RDMA in combination with the use of SGX.

Panoply [7], Graphene-SGX [5] and SCONE [6] offer self-contained system library or libOS infrastructures for enclaves and provide a general purpose trusted execution environment for legacy programs. Thus, in principle, an in-memory database such as STANlite can be hosted by these systems, but none of them offer scalable, enclave-based paging support, but instead they all rely on the Intel SDK provided functionality or do not specifically address memory usage beyond the EPC size.

Glamdring [27] is a source-level partitioning framework, which enables the use of SGX enclaves to protect security-sensitive data and functions of complex C programs. The SQL engine of STANlite is insignificant compared to the EPC size, thus there is no need for partitioning and would likely results in performance degradation due to additional mode changes.

There are a couple of systems that offer encryption and secure data processing at the database level. MrCrypt [2] uses homomorphic encryption and processes queries in an encrypted form. CryptDB [1] also provides query-based homomorphic encryption and operates as a proxy which encrypts sensitive information at the request level. Working on encrypted data either reduces query expressiveness or substantially impacts performance.

Recently there have been multiple works to secure a specific service [16], [28] or a type of middleware [29], [30]. Neither of them addresses a in-memory database, specifically memory management for complex query processing nor the use of novel communication techniques as offered by RDMA.

TrustedDB [31] and Cipherbase [32] use specialized hardware to accelerate secure query processing. STANlite has similar goals but only relies on commodity hardware and secures processed data and code.

Finally, there are related projects that utilized RDMA to speed up remote communication. Pilaf [9], MICA [10] and HERD [11] are key-value stores that utilize RDMA. While these projects provide high-performance storages, they do not have any mechanisms for data protection, neither during data transfer nor while performing data processing. Lightbox [33] is an SGX-based system that explored how trusted execution and TCP/IP can be efficiently combined. Contrarily, STANlite combines the use of RDMA with trusted execution.

VII. Conclusion

In this work we present STANlite – an in-memory database engine for SGX-enabled secure data processing in rack-scale environments. While SGX promises trusted execution that is close to native speed, frequent mode transitions (e.g., due to I/O operations) and memory usage beyond the EPC size of currently 92 MiB result in severe performance degradation. With STANlite we address this performance issue by employing a custom VME, which features customizable memory management polices, thereby avoiding mode transitions, and using an RDMA-based communication layer to improve remote access speeds.

Acknowledgement

This work was partly supported by the German Research Foundation (DFG) under priority program SPP2037 grant no. KA 3171/6-1 and LE 3382/3-1.

References

[1] 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 '11)*, 2011, pp. 85–100.

[2] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "MrCrypt: Static analysis for secure cloud computations," *ACM Sigplan Notices*, vol. 48, no. 10, pp. 271–286, 2013.

[3] Intel, "Architecture instruction set extensions programming reference," 2014. [Online]. Available: https://software.intel.com/en-us/isa-extensions

[4] Intel. (2017) Intel software guard extensions for linux os. [Online]. Available: https://01.org/intel-softwareguard-extensions

[5] C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 645–658.

[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX." in *OSDI*, 2016, pp. 689–703.

[7] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "PANOPLY: Low-TCB Linux Applications With SGX Enclaves," in *Proc. of the Annual Network and Distributed System Security Symp.(NDSS)*, 2017.

[8] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 81–93.

[9] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *USENIX Annual Technical Conference*, 2013, pp. 103–114.

[10] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 429–444.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 295–306.

[12] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, p. 11.

[13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 10:1–10:1.

[14] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015, pp. 38–54.

[15] N. Balakrishnan, L. Carata, T. Bytheway, R. Sohan, and A. Hopper, "Non-repudiable disk I/O in untrusted kernels," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017, p. 24.

[16] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX." in *Middleware*, 2016, p. 14.

[17] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015, pp. 640–656.

[18] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 557–574.

[19] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[20] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *European Symposium on Research in Computer Security*, 2016, pp. 440–457.

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

[22] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[23] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory," in *Proceedings of the 47th International Conference on Dependable Systems and Networks*, ser. DSN'17, 2017, pp. 157–168.

[24] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1289–1306.

[25] Council, Transaction Processing Performance, "TPC benchmark C (standard specification, revision 5.11), 2010," *URL: http://www.tpc.org/tpcc*.

[26] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *EuroSys*, 2017, pp. 238–253.

[27] J. Lind, C. Priebe, D. Muthukumaran, D. OKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for Intel SGX," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017, p. 24.

[28] S. B. Mokhtar, A. Boutet, P. Felber, M. Pasin, R. Pires, and V. Schiavoni, "X-search: Revisiting Private Web Search Using Intel SGX," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17, 2017, pp. 198–208.

[29] R. Pires, M. Pasin, P. Felber, and C. Fetzer, "Secure Content-Based Routing Using Intel Software Guard Extensions," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, 2016, pp. 10:1–10:10.

[30] S. Brenner, T. Hundt, G. Mazzeo, and R. Kapitza, "Secure Cloud Micro Services using Intel SGX," in *Proceedings of the 17th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS '17)*, 2017.

[31] S. Bajaj and R. Sion, "Trusteddb: A trusted hardware-based database with privacy and data confidentiality," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 3, pp. 752–765, 2014.

[32] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan, "Secure Database-as-a-service with Cipherbase," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1033–1036.

[33] H. Duan, X. Yuan, and C. Wang, "LightBox: SGX-assisted Secure Network Functions at Near-native Speed," *arXiv preprint arXiv:1706.06261*, 2017.